Agentry Language Reference

SAP Mobile Platform 3.0

Contents

Agentry Language Reference

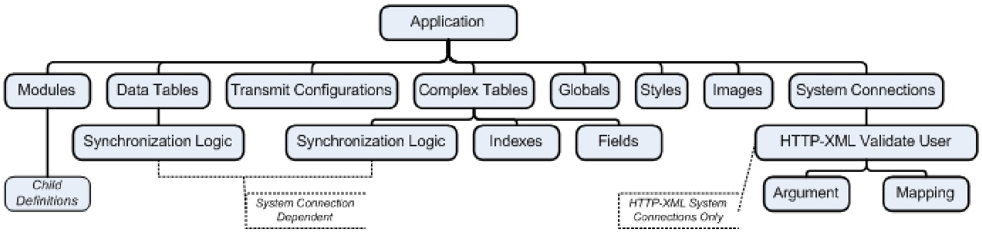
Use the Agentry Language Reference to learn about the following.

Application Level Definitions Overview

Within the application project structure in Agentrythe definitions at the application level are at the top of the hierarchy. These definitions affect the application as a whole. The definitions that are direct children of the application are those that affect communications behavior, globally available constant values used for configuration and other purposes, and also include data storage on the client in the form of tables and records accessible to the entire application. In addition are the definitions that can affect the appearance of the user interface.

The application itself is represented as a definition type within the application project. Within a project there is only one application definition. The child definitions to the application are then referred to as the application-level definitions. Regardless of functionality, most of the application-level definitions will be used in a given application.

Following is the structure of the application level definitions within the application project. For all definitions in this graphic the child definitions are also shown, with the exception of the module. Modules are a robust definition type and the structure of the module is provided with the module-level discussions.



As illustrated in this graphic, the child definitions to data tables and complex tables related to synchronization are dependent on the type of system connection for which those definitions were created. The synchronization logic will be encapsulated in the language or methodology matching that back end system type.

As denoted in the illustration the system connections of type HTTP-XML include child definitions related to user validation. The user validation request is sent according to these definitions, including arguments to that request. Responses from the request are then mapped to the data components of the mobile application.

The module definition type contains numerous child definitions not represented in this graphic. These are illustrated in the sections covering the module-level definition types.

In general when working within the Agentry Editor to either develop an application or to modify an existing one, the application-level definitions dictate and control aspects of the application behavior overall, rather than within a given module or lower-level of granularity.

Application Definition

The application definition type represents the mobile application within the project and all definitions for the application are its descendents. The attributes of the application definition are those that affect application-level behaviors. These include the application name and version, the appearance of built-in Client screens, login and password settings, application-wide screen and user interface behaviors, and other similar items. The application definition is the single root definition in the application hierarchy and as such has no parent definition.

When a new application project is created in the Agentry Editor, an application definition is automatically created. Its attributes are set to defaults that should be reviewed thoroughly early in the development process. These attributes can affect security, appearance, and numerous other behaviors of the application.

General Setting Attributes

The general setting attributes for the application a provide project name, the application's display name, and a version value.

- **Name:** This is the internal name of the application. This value is used for certain checks during publish.
- **Display Name:** This is the name of the application as displayed on the Agentry Client. This value appears in the title bar of the application and in the About dialog displayed from the Agentry Client's Help menu. For any newly created application project this is set to a default of Agentry and should be changed.
- **Version:** This value is also displayed to the user in the About screen. This value is not related in any way to the application's publish version number. This attribute value is provided for branding purposes only and will not impact any aspect of the application's behavior. Typically this reflects the application's release version.

Application Setting Attributes

Table Settings - These attributes affect the behavior of data synchronization related to the two table definition types, data tables and complex tables.

- **Check Data Tables:** Specifies how often the application will check for new or changed data for the application's data tables. The choices are "Every Transmission", "Once per day", and "Once per week". In between the specified intervals, no synchronization components of the application's data tables will be processed during a transmit. A published change to the data table definition will override this attribute, forcing a reload of the data table during the next Agentry Client transmit.
- **Check Complex Tables:** Specifies how often the application will check for new or changed data for the application's complex tables. The choices are "Every Transmission", "Once per day", and "Once per week". In between the specified intervals, no

synchronization components of the application’s complex tables will be processed during a transmit. A published change to the complex table definition will override this attribute, forcing a reload of the complex table during the next Agentry Client transmit.

- **User Request:** This setting specifies whether or not users can explicitly check for changes to the application’s complex tables and data tables. This is a means of providing users with a manual override for the **Check Data Tables** and **Check Complex Tables** attribute settings. When **User Request** is enabled, users will be able to force the synchronization process to include the processing of the data table and complex table definitions’ synchronization components. Users will be able to force this behavior by selecting the menu item “Check for Table Updates” in the Agentry Client’s Off-Line menu. This attribute will have no effect when the **Check Data Tables** and **Check Complex Tables** attributes are set to “Every Transmission”.

Client Settings - Client Settings affect various behaviors of the client application at runtime.

- **When Exiting Client:** This enables a warning message displayed to the user if there are pending transactions stored on the Agentry Client when they exit the application.
- **Prompt on User Change:** This enables a prompt when a user change occurs, informing the user that a synchronization with the Agentry Server must take place to change users and gives the user the option to cancel the user change. If this is disabled, the synchronization will still occur to complete a user change, but no prompt will be displayed.
- **Module Menu Item:** This attribute specifies whether or not the menu item for the current module is enabled or disabled in the Agentry Client’s View menu at runtime. Selecting the current module from the View menu will return the user to the module’s main screen set, regardless of where they may be in the navigation. When disabled, the menu item for the current module is disabled. Users can always select other module items in this menu for applications with multiple modules regardless of this setting.
- **Synchronize Clocks:** This attribute specifies whether or not the system time on the client device will be reset to that of Agentry Server’s host system time during each transmit. Note that this time is not the time of the back end system with which the Agentry Server communicates. It is the system time as reported by the operating system of the Agentry Server’s host system. This is typically disabled in deployments involving multiple time zones.
- **Screen Size:** This setting specifies the size of all screens displayed to the user on the Agentry Client. This attribute will only effect Agentry Client applications running on a Windows PC platform capable of full VGA screen resolution. The screen sizes available for this setting range from 240 x 320 (1/4 VGA) to 1366 x 768. The Screen Size value will override the screen size attribute for all platform definitions. There is also the available setting “Allow Resize”. If any selection other than Allow Resize is made, users will not be able to resize Agentry Client screens. The screen size for all mobile devices, including smart phones tablets, and other devices, is always full resolution of those devices and users can never resize the screens.
- **Battery Status:** For mobile devices, the status of the battery can be displayed on the Agentry Client. This will appear in either the upper or lower portion of the screen,

depending on the device. Note that this setting has no effect on the Windows PC builds of the Agentry Client.

- **WinCE Navigation:** This setting enables support for the arrow keys of a device's hardware keyboard. When a user clicks one of the arrow keys, the focus of the screen will be changed to the next or previous control on that screen. This attribute has no effect on the Windows PC builds of the Agentry Client, where full keyboard navigation is always enabled.
- **Scan Trigger Shortcut Key:** For devices equipped with a scanner, this attribute allows for the specification of a shortcut key to activate the scanner. This key will be universal to the application, and will activate both socket and built-in scanners. This attribute will have no affect on Agentry Clients running on devices not equipped with a scanner.
- **Voice Support:** Enables voice support for devices that support this feature.
- **Title Bar Buttons:** This attribute specifies whether or not the close buttons (either an X or an OK button) are displayed on the title bar of screens within the mobile application. Due to the behavior of Pocket PC devices, it is recommended that these buttons not be displayed and that actions are defined within the Agentry application project to close screen sets within the application, and that users close the application itself using either the **File | Exit** menu item, or through an action containing an Exit Application action step. Note that on Pocket PC devices, screens closed with the title bar's OK button are not destroyed, but rather only moved to the end of the "Z" order, hiding them from view. Applications closed with the X button of the title bar are not actually exited. Any defined behaviors for exiting an application will not be exhibited. Furthermore, the application itself will still be running. The behaviors described here are not present on a Windows PC platform.
- **Theme Selection:** This attribute specifies whether or not the Theme menu item within the Preference menu of the Agentry Client is displayed. When true (checked) the user can change the Agentry Client theme using this menu. When false, the user cannot change the Agentry Client theme and the theme displayed is always the one selected in the **Default Theme** attribute. Allowing user's to select a different theme can have unexpected impact on the UI of the Agentry Client if styles are defined and in use.
- **Default Theme:** The theme selected is the default theme displayed on the Agentry Client at run-time. If **Theme Selection** is disabled, the selected Default Theme is always displayed and the user cannot change the theme selection. If **Theme Selection** is enabled, the selected **Default Theme** will be the initially applied theme on the Agentry Client, but the user can select a different theme at any time.
- **Win32 Buttons - Use large buttons:** This attribute specifies whether or not to use large sized buttons. This attribute affects only the Windows PC platform types. When true, screen buttons are displayed in a large size, generally intended for touch screen support. This setting affects button definitions for list and detail screens. Built-in buttons, such as ellipses buttons, icon buttons, and similar controls are not affected. Note that large buttons are not displayed in the Agentry Editor's layout view or visual screen editor for screen definitions. They are displayed in the Agentry Test Environment when the selected platform is Windows.

Application Styles Attributes

The attributes listed in the Application Styles tab define how styles are to be applied to all components of the application's user interface. These style settings may be overridden at lower levels in the application's structure. The style settings here also impact what styles are applied to the Agentry Client's built in screens and dialogs, such as those for complex table searches, the transmit dialog, and others. For all style attributes, the option "--Default--" will default to the operating system's default font and color options.

Screen Styles

- **Tabs:** The style to apply to the tab controls representing each screen within an object screen set. This attribute has no effect on screens within a transaction or fetch screen set.
- **Buttons:** The style to apply to all button definitions on all application screens. This includes buttons displayed on built-in Agentry Client screens as well as buttons within screen definitions.
- **Focused Buttons:** The style to apply to the button that currently has the focus. This includes buttons displayed on built-in Agentry Client screens as well as buttons within screen definitions.

Detail Screen Styles

- **Screen:** The style to apply to all detail screens defined within the application. This will affect all portions of the screen not displaying a field or button.
- **Fields:** The style to apply to all fields displayed on a detail screen.
- **Focused Fields:** The style to apply to the detail screen field that currently has the focus.
- **Read-Only Fields:** The style to apply to a detail screen field defined to be read-only. If not specified, the Fields style is applied.
- **Hyperlinks:** The style to apply to detail screen field labels defined to be hyperlinks.
- **In Progress Edit Screens:** The style to apply to screens in which changes are currently being made and have not yet been applied. This affects screens displayed in List Tile View and Tile Edit fields.

List Screen Styles

- **Screen:** The style to apply to all list screens as a whole. This will affect all portions of the screen not displaying a list, header label, detail pane, or button.
- **Header Label:** The style to apply to all list screen header labels. If no header label is defined this attribute has no affect on the screen.
- **Column Labels:** The style to apply to the column labels on the screen's list control. This style is applied to the column labels of any screen containing a list control, including both built-in Agentry Client screens as well as list screen definitions, and list view field definitions.
- **Rows:** The style to apply to all rows on a list screen. The **Hyperlinks** optional style will override the **Rows** style for cells with hyperlinks. This style is applied to the list items of

any screen containing a list control, including built-in Agentry Client screens, list screen definitions, and list view field definitions.

- **Alternate Rows:** The style to apply to every other row in a list, beginning with the second row. The **Hyperlinks** optional style will override the **Alternate Rows** style for every other row where there are cells containing hyperlinks.
- **Highlight Rows:** The style to apply to a row for the purposes of drawing attention to that row. This style attribute should not be set at the application level. The platform and list screen definition types both contain a Highlight Rows attribute that should be used.
- **Selected Rows:** The style to apply to the row or rows currently selected by the user in the list control. The optional **Hyperlink** style will be applied to any cells within the selected row containing a hyperlink.
- **Selected No Focus Row:** The style to apply to the row or rows currently selected by the user in the list control, when the input focus is set to some control other than the list control. The optional **Hyperlink** style will be applied to any cells within the selected row containing a hyperlink.
- **Detail Pane:** The style to apply to both the foreground (text) and background of a list screen's detail pane. If no detail pane is defined this attribute has no effect on the screen.

Application Images

- **About Box Dialog Image:** This attribute specifies an image definition to display in the Agentry Clients' about box.
- **Login Dialog Image:** This attribute specifies an image definition to display in the Agentry Clients' login screen.
- **Module Menu Dialog Image:** This attribute specifies an image to display in the Agentry Clients' module selection dialog. This dialog is displayed after users log into Agentry Client applications with multiple modules. Note that within this same module selection dialog, each module may also display an image. The image defined at the application level is separate from the module images. For applications with a single module, this attribute has no effect as the module selection dialog is never displayed.

Application Security Attributes

The attributes in this section control overall security related to items such as failed login attempts, locking the user out of the client application after failing validation, user ID and password rules, and idle timeout settings.

User Settings

- **Login:** This attribute specifies whether or not users are required to perform a transmit every time the Agentry Client application is started and the user logs in. Note that setting this option to true requires an available network connection for the client device and users will be required to perform a successful transmit before the user can use the client application.
- **Login Attempts:** This attribute specifies the maximum number of failed login attempts that may occur before locking the user out of the application. "Unlimited" will never lock

the user out. A failed login will occur if the user enters an incorrect password for the entered User ID when not required to connect to the Server during login; or if the user fails user authentication when the Login attribute is set to true and the number of failed attempts exceeds the value entered here. The resulting behavior of locking out a user is defined in the Lockout Level attribute described below.

- **Lockout Level:** This attribute specifies the action to take when a user is to be locked out of the Agentry Client. This may occur as a result of exceeding the defined **Login Attempts**, or based on failed attempts to authenticate the user against the back end system. The four possible settings for this attribute are:
  - <u>Critical:</u> This lockout level specifies that the entire Agentry Client be reset. This includes the removal of all production data and all application data, as well as removing the stored user ID and password. Users will be required to log into the Agentry Client application and perform a successful transmit with the Agentry Server before being allowed to access the Agentry Client application.
  - <u>Severe:</u> The severe lockout level specifies that all module-level production data, i.e. object instances and pending transactions, be removed from the Agentry Client application. Complex table and data table records will not be removed. The user will be required to perform a successful transmit with the Agentry Server before being allowed to access the client application.
  - <u>Medium:</u> This lockout level specifies the Agentry Client will exit and the user will be required to log into the Agentry Client and perform a successful transmit with the Agentry Server before being allowed to access the client application. No data is removed from the application.
  - <u>None:</u> This setting indicates that no lockout behavior should take place. This setting will result in any lockout events being ignored by the Agentry Client.
- **Client Database will be encrypted:** When true, this attribute specifies the database in which all client data is stored on the client device, including both production data and application data, will be encrypted.

User ID

- **Case:** This attribute specifies the case in which the user ID should be entered and will be stored. The options are mixed case, uppercase, and lowercase. Note that mixed case does not require a mixed case user ID, but rather allows for variable case. User IDs may then be in all upper, all lower, or mixed case.
- **Scan User ID:** This attribute specifies whether or not users can enter user IDs via the device’s barcode scanner. This attribute does not require the ID to be scanned, but only allows for the option. This attribute has no effect on Agentry Clients running on devices not equipped with a barcode scanner.

Password

- **Password Change:** This attribute specifies whether or not users can initiate password changes on the Agentry Client. When true, the users will be able to change the password based on responses from the back end system indicating their passwords are about to expire or have expired. Users are required to enter the old password and the new one to change passwords. Note that enabling this behavior requires the implementation of logic to process a password change for the user in the back end system.

- **Scan Password:** This attribute specifies whether or not users can enter passwords via the device's barcode scanner. This attribute does not require the password to be scanned, but only allows for the option. This attribute has no effect on Agentry Clients running on devices not equipped with a barcode scanner.

Idle Timeout

- **Timeout:** This attribute specifies whether or not to require users to re-enter their user ID and password if the device is left idle for a defined duration of time. The duration is set as a part of this attribute. Also an option is whether or not the user ID should be populated automatically.

Password Rules

- **Minimum Length:** This attribute specifies the minimum number of characters of the password entered on the Agentry Client. The minimum length must be at least 1 to enable the **First Character** attribute. This value must be at least 2 to enable the **Character Mix** attribute. This value must be equal to or less than the **Maximum Length** attribute. The default minimum is none, which does not require a password to be entered on the Agentry Client.
- **Maximum Length:** This attribute specifies the maximum number of characters of the password entered on the Agentry Client. This value must be equal to or greater than the **Minimum Length** attribute, or be set to default, which is no maximum length.
- **Password Case:** This attribute specifies the case in which the user's password is stored on the Agentry Client and will be sent to the Server. This may be set to Mixed Case, Lowercase Only, or Uppercase Only. Note that Mixed Case does not enforce a requirement of a mixed case password. Rather it merely specifies that the case of the password characters will not be changed from how they are entered by the user.
- **Character Mix:** This attribute requires the **Minimum Length** attribute to be set to at least 2. **Character Mix** requires passwords entered on the Agentry Client must contain at least one alphabetical character and one non-alphabetical character. Non-alphabetical characters exclude non-printable characters.
- **First Character:** This attribute requires the **Minimum Length** attribute to be set to at least 1. **First Character** specifies that the first character of the password must be an alphabetical character.
- **New vs. Old:** This attribute specifies that a new password entered by the users on the Agentry Clients must be different from the previous password. A difference is based on the change of at least one character from the previous password to the new one. This attribute may be impacted by the **Password Case** attribute. **Mixed Case** will treat the same letters in the old and new password as different if at least one letter is entered in a different case. For Uppercase Only and Lowercase Only **Password Case** settings, case is ignored and the same letters entered in a different case will not be treated as a different password.

Module

The module definition is a grouping of definitions providing functionality that logically belongs together. The module's attributes and child definitions define the majority of the behavior and functionality exhibited on the Agentry Client at runtime.

The modules of an application contain the functionality related to the user interface on the Agentry Client, data storage and structures, data synchronization, and data capture. The child definitions of a module also have access to all application-level definitions.

An application project must contain at least one module. When multiple modules are defined for an application, users will be required to select which module to work with when logging into the Agentry Client application. They will be able to switch from one module to another using the Agentry Client's View menu, which will list the defined display name for each module within the application.

The module's child definitions are primarily intended to work with other definitions within the same module. Cross-module functionality can be defined using actions within one module that may execute actions of another module within the same application.

Module Child Definitions

- **Action** - An action defines navigation and user interaction for the Agentry Client, bringing the other components of the Client's UI together.
- **Fetch** - A fetch defines how the Agentry Server synchronizes data for a target object collection by referencing the step definitions to perform this task.
- **Object** - An object definition encapsulates a business entity and its related data.
- **Push** - A push defines when it is necessary to push an object in real time from the back end system to the Agentry Client and how that object's data is retrieved.
- **Report** - A report defines a printed tabular report format for the contents of an object collection on the Agentry Client.
- **Rule** - A rule defines evaluation logic processed on the Agentry Client that returns a single value to the caller of the rule.
- **Screen Set** - The screen set is the main Client user interface definition and defines what definition type its child screens display.
- **Service Event** - A service event defines how the Agentry Server synchronizes data between two back end systems, usually based on a change or "event" occurring in one of the systems.
- **Step** - A step defines a piece of processing to be performed by the Agentry Server with a specific back end system.
- **Transaction** - A transaction definition defines what data is captured on the Client, how that data affects a target object instance on the Client, and how the captured data is processed by the Agentry Server.

Module Attributes

- **Name -** This is the unique name of the module. This value must be unique among all modules defined within the application.
- **Display Name** - This is the text displayed to the users on the Agentry Client application at runtime. This value appears in the Agentry Client's Module Selection Screen to represent the module and also appears in the View menu of the Agentry Client as a menu item.
- **Preserve Objects** - This attribute specifies whether or not the objects within the module will be preserved when a new user logs into the Agentry Client on the same device as a previous user. If checked, the objects will be preserved from one user to the next. If left unchecked, a user change will result in the objects being removed prior to synchronizing object data for the new user.
- **Image:** Specifies the image definition to associate with the module definition. This image is then displayed for the module in the Agentry Client's Module Selection Screen displayed after login for multi-module applications.
- **Successful Login Action:** Specifies an action defined within the module to be executed after a user successfully logs into the application. The action executed here targets the module main object. For multi-module applications where more than one module defines a Successful Login Action, the order in which those actions are executed is undefined.
- **Application Exit Action:** Specifies an action defined within the module to be executed just prior to exiting the application. The action executed here targets the module main object. For multi-module applications where more than one module defines an Application Exit Action, the order in which those actions are executed is undefined.

Data Table

A data table definition defines a set of records stored on the Client. Each record consists of two fields containing a key and value. A data table is intended to contain a small number of records (less than 100) that may be displayed to users in drop-down lists and other uses. A data table is defined at the application level and is available to all modules of the application. Its structure also defines how its data is synchronized.

The intended purpose of a data table is to provide short lists of records that can be created quickly and with little overhead related to maintaining the data. A data table has no built-in search support and if searching is necessary it is performed row-by-row (e.g., no binary or other search algorithms are employed).

As a part of its definition, the data table contains the components to synchronize data. This includes determining if new data is needed for the table as well as the processes to retrieve the records for the data table. The definition of a data table requires the selection of an existing system connection. The type of synchronization components a data table contains is based on the type of the selected system connection.

Though the synchronization components will differ in form and structure related to the type of back end system for which they are intended, they are required to always return two general categories of data to the Agentry Server. The first is a date and time value retrieved from the back end that indicates the last time when the data source for the data table was last modified in the back end system. The second is the actual data for the data table’s records.

The date and time value is compared to a date and time value stored internally on the Agentry Client for each data table instance. This internal value is called the data table’s last update value. This last update value indicates when the data table was downloaded to the Agentry Client. When the date and time retrieved from the back end is newer than the Agentry Client’s last update value for the data table it is the indication that the records for the data table must be retrieved. The existing records on the Agentry Client will be deleted and replaced with the new data retrieved for the data table. This is an all-or-none operation and individual records cannot be selectively replaced.

The specifics of how the date and time values for the data tables are retrieved, and how the records are retrieved for the data table are provided in the discussions specific to each of the possible system connection types that may be selected for the data table definition when initially defined.

Data Table Attributes

The following attributes are applicable to all data tables, regardless of the system connection a given data table may be using.

- **Name:** This is the unique name of the data table. This value must be unique among all data tables defined for the application.
- **Display Name:** This is the default text displayed to the user on the Agentry Server for the data table.
- **Connection:** This is the system connection defined for the back end system containing the data source for the data table. This attribute is set when the table is initially created. It cannot be edited for an existing data table definition. The system connection to be used must exist prior to defining the data table.
- **Reload:** This attribute specifies whether or not the records of the data table should be reloaded when a user change occurs on the Agentry Client. When true, all records in the data table are deleted and completely reloaded during the first transmit of the new user. Otherwise the records will remain on the Agentry Client during the user change. This attribute should be set to true when the data table contains records that are user-specific.

SQL Data Table Synchronization Components

When a data table is defined to use a SQL Database system connection, the synchronization components include a Sync Query and a Data Query.

The Sync Query is expected to return a value identified as `LastUpdate`. This value should indicate the date and time the source table in the database was last modified. This value is then compared to the last update value for the data table provided from the Agentry Client. If the date and time value returned by the Sync Query is not newer than the one for the data table, no further processing for the data table occurs.

If the Sync Query `LastUpdate` value is newer than the Agentry Client's last update value for the data table, the Data Query is run. This query is expected to return all records for the data table, whether or not an individual record is different in the database. This query is expected to return two columns identified as `CODE` and `VAL` to the Agentry Server. The value of the `CODE` column must be unique within the return set provided by the Data Query.

Sync Query and Data Query Attributes

- **Sync Query - File:** Specifies the name and location of the text file (`.sql` extension) containing the SQL statement for the Sync Query.
- **Data Query - File:** Specifies the name and location of the text file (`.sql` extension) containing the SQL statement for the Data Query.

HTTP-XML Data Table Synchronization Components

When a data table is defined to use an HTTP-XML system connection, the synchronization component it contains is an HTTP request.

This request is a child definition to the data table. It can be defined to make a request to a specified URL and may use the request methods GET, POST, HEAD, or PUT. The HTTP request itself contains two types of child definitions: Request Arguments and Response Mappings.

The request arguments contain the data values passed as arguments to the back end system as a part of the request being made. The response mappings are defined using XPaths to retrieve data from structured XML return values provided by the back end system as a result of the HTTP request. These mappings can include the back end system's last update value for the data table's data source, the data values for the records to be stored in the data table, and other types of data.

There is a single request made to synchronize the data table, with the value mapped to the `LastUpdate` value determining whether or not the data values returned should be used to replace the data table on the Agentry Client.

Data Table HTTP Request Child Definitions

Following is a list of the child definitions for the HTTP Request within an HTTP-XML data table.

- **Request Arguments:** This definition encapsulates an argument to be passed with the request to the back end system. These arguments can include data contained within the mobile application.
- **Response Mappings:** This definition encapsulates the XML data returned by the HTTP Request. The specific values are extracted from the XML return data using XPaths defined within each response mapping. The mapping "maps" the extracted values to values within the mobile application.

Data Table HTTP Request Attributes

The following attributes pertain the HTTP Data Request of a data table defined to use an HTTP-XML system connection.

- **Name:** The name of the request, set automatically based on the parent data table name. May be modified if desired.
- **URL:** The URL to the specific CGI or other process being called by the HTTP request to synchronize the data table.
- **Method:** The HTTP request method for the request. May be one of GET, POST, HEAD, or PUT.

HTTP Request Argument

The request argument definition encapsulates a data value to be passed from the mobile application to the process being called by the parent HTTP request definition. The request argument specifies the argument type, which may be CGI Argument, Cookie, HTTP Header, or XML Body. The request argument also specifies the data or data source within the mobile application to pass as the argument to the process or service being called by the parent HTTP request definition.

For a data table, the data value may be the user ID, the name of the data table, a fixed string whose value is defined as a constant within the request argument, or markup text. A given parent HTTP request may contain multiple request arguments. The order in which they are passed to the process or service when called is defined in the parent HTTP request's list of request arguments.

HTTP Request Argument Attributes

The attributes of a request argument depend in part on the data type of the argument (Data Type attribute). The following list makes note of those attributes specific to a certain argument data type.

- **Argument Type:** This attribute specifies the type of argument the definition contains. This may be one of CGI, Cookie, HTTP Header, or XML Body.
- **Name:** Alternately displayed as Argument Name, Cookie Name, Header Name, or Name depending on the **Argument Type** selection. This value must be unique among all request arguments defined within the same parent HTTP request definition.
- **Data Type:** Specifies the data value or source for the data value for the request argument. For a data table this may be the Table Name, User ID, Small or Large Markup, or a Fixed String value.
- **String:** This attribute is available only when the **Data Type** attribute is set to Fixed String. String contains the constant string value that is the request argument's data.
- **Markup Text:** This attribute is available only when the **Data Type** attribute is set to Small Markup or Large Markup. **Markup Text** contains the single line (Small Markup) of

markup text or the contents of the Markup File (Large Markup) that is the data for the request argument.

- **Markup File:** This attribute is available only when the **Data Type** attribute is set to Large Markup. **Markup File** contains a reference to the text file containing the multi-line markup text. This file is displayed in the **Markup Text** field directly below the file name in the Editor and can be authored or modified directly in this multi-line field.

HTTP Request Response Mapping

The response mapping definition is a child to an HTTP request definition. This definition maps a data value returned from the process called by the HTTP request to a value within the mobile application. This value may be extracted from structured XML using XPath or XSL. It may also be a Cookie value or the HTTP Header.

For a data table the values may be mapped to the code or value fields in a data table record, an error message, the last update value to be compared against the data table's last update, a local data tag or local XML value, or to the user ID value that may be used in place of the ID entered to log into the Agentry Client.

HTTP Request Response Mapping Attributes

The response mapping attributes are in part dependent on the selection made in the Mapping Type attribute. Those specific to a certain type are denoted in the following list.

- **Mapping Type:** This attribute specifies the mapping type. This may be Cookie, HTTP Header, XPath Expression, or XML Transformation.
- **Base XPath:** This attribute is only available when the **Mapping Type** is set to XPath Expression or XSL Transformation. This attribute is optional and should be used when returning multiple instances of the same data element in the XML content. When a **Base XPath** is defined for a response mapping, the same value will be set by default in the add wizard for subsequent response mappings within the same parent HTTP request definition.
- **XPath:** This attribute is only available when the **Mapping Type** is set to XPath Expression. This attribute contains the XPath markup to extract the desired value from structured XML data returned from the HTTP Request.
- **XSL:** This attribute is only available when the **Mapping Type** is set to XML Transformation. It contains the XSL expression to transform the XML data to be mapped to a value for the request.
- **Cookie Name:** This attribute is only available when the Mapping Type is set to Cookie. It contains the name of the cookie for the response mapping.
- **Header Name:** This attribute is only available when the Mapping Type is set to HTTP Header. It contains the name of the HTTP header for the response mapping.
- **Maps To:** This attribute specifies where the value extracted by the response mapping is stored in the mobile application. This may be one of the following values for a data table:

  - Data Table Key: This selection specifies the value extracted by the mapping contains the key or code field value for each data table record.
  - Data Table Value: This selection specifies the value extracted by the mapping contains the value field value for each data table record.
  - Error Message: This selection will map the data to error text displayed by the mobile application.
  - Last Update: This selection specifies the extracted value is a date and time indicating when the data table's source in the back end system was last modified. This value is compared against the internal last update value for the data table as provided by the Agentry Client.
  - Local String (`<<local>>`): This selection will create a local data tag available to subsequent data mappings in the same parent HTTP Request. The value of this data tag will be the value extracted by the response mapping. When selected, the attribute **String Name** will be available to name the new local data tag. This is the equivalent to calling the SDML function tag `<<local ...>>`.
  - Local XML (`<<localXML>>`): This selection will create a local XML data tag available to subsequent data mappings in the same parent HTTP Request. The value of this data tag will be the value extracted by the response mapping. When selected, the attribute **XML Name** will be available to name the new local data tag. This is the equivalent to calling the SDML function tag `<<localXML ...>>`.
  - User ID: This selection will map the value extracted by the response mapping to the user's ID. This value is the equivalent to the SDML data tag `<<user.id>>`. If a previous response mapping in any HTTP Request processed by the Agentry Server set the user ID, setting it here will override that value. This will then be the value available for all HTTP-XML system connection processing where the data tag `<<user.id>>` is referenced.
- **String Value:** This attribute is available when the map type is set to Local String. It contains the name of the data tag being created by the mapping. This is the name by which the data tag is referenced in subsequent references.
- **XML Name:** This attribute is available the map type is set to Local XML. It contains the name of the data tag being created by the mapping. This is the name by which the data tag is referenced in subsequent references.

Java Virtual Machine Data Table Synchronization Components

When a data table is defined to use a Java Virtual Machine system connection, its synchronization component is a single Java source file. This file contains the skeleton structure for a Java class that extends the Agentry Java API class `DataTable`. The name given to this class matches the name of the data table definition itself and should not be changed.

When the table is defined, the wizard for creating Java classes provided with the Eclipse Java perspective is used. This allows the developer to select the package to which the new class will be added. The source `.java` file created will then be stored according to the configuration of the project and package selected for the new class. Alternately an existing class in a package within the Java perspective may be selected. This class must extend the Agentry Java API class `DataTable`.

This skeleton class declaration includes three methods:

- The Constructor method.
- An override method for the data table `iterator()` method. This method is intended to contain the logic to retrieve the data from the back end system via the Java interface it provides. It is then intended to return an iterator for the data table object.
- An override method of `isOutOfDate()`. This method is expected to return true or false based on whether or not the data for the table is out of date. When true is returned by this method, the iterator() method will be called. When false, processing for the data table by the Agentry Server will be complete.

In versions of the Agentry Mobile Platform prior to 5.1, the source class was stored on the Agentry Server's file system. This behavior is deprecated in versions 5.1 and later. Agentry application projects created prior to this release are still supported and the Java logic will still be processed correctly. New data tables for Java Virtual Machine system connections should use the new procedure for defining the Java synchronization component.

Complex Table

The complex table definition defines a table of records containing multiple fields stored on the Agentry Client in a structured and searchable format. A complex table can contain large amounts of data with records numbering in the thousands. Included in the complex table are the fields for its records and indexes on fields to provide search functionality and structure to the overall data in the table. The complex table definition also defines how its data is synchronized.

The fields and indexes of a complex table define the structure of the records. A complex table must have a minimum of one index definition, which is the primary index. This index is defined for the field containing the unique identifier for each record. This field and index are then used during synchronization to identify records for addition, replacement, or removal.

The synchronization components of a complex table depend on the system connection the table definition uses for its data source. The synchronization components will match the system connection type. Independent of the system connection type, the synchronization logic for a complex table should account for retrieving all records when the table is in a rebuild state, retrieving just new or modified records during normal synchronization, as well as determining which records should be removed from the complex table.

The rebuild state of a complex table is set under various conditions. These include a published modification to the complex table definition, a user change occurring on the Agentry Client, and optionally based on the rebuild state being forced via administrator actions. During synchronization between the Agentry Client and Agentry Server, the Agentry Server will indicate if the complex table is in a rebuild state to the Agentry Client. The Agentry Client will remove all records for the complex table from the client device. The synchronization processing will retrieve all current records for the complex table and send them to the Agentry Client, rebuilding the table. This synchronization processing requires the developer to account for this situation.

When not in a rebuild state, the complex table can be updated selectively. Using an exchange data model for processing, only those records to be added, those records that need to be replaced, or those that need to be removed from the Agentry Client are retrieved by the Agentry Server from the back end system for the complex table. Any unchanged records will be left unmodified.

Complex Table Child Definitions

- **Field:** A complex table field definition defines a single piece of data for a complex table record, including its data type and size.
- **Index:** A complex table index definition orders the table's records by a field, making the table searchable by that field.

Complex Table Attributes

- **Name:** This is the unique name of the complex table. This value must be unique among all complex tables defined within the application.
- **Display Name:** This is the default text displayed to the user on the Agentry Client identifying the complex table.
- **Connection:** This is the system connection used by the complex table's synchronization components to synchronize the records of the complex table on the Agentry Client.
- **Reload:** This attribute specifies whether or not the records of the complex table should be fully reloaded when a user change occurs on the Agentry Client. When true, all records in the complex table are deleted and completely reloaded during the first transmit of a new user. When false, the records downloaded by the previous user are kept. This attribute should be set when the records of the complex table are user-specific.

Complex Table Fields

A complex table field definition defines a field in each record of the table, including the data type of the field and the size of the data the field can store. A record within the table can consist of multiple fields of varying types and sizes.

A Complex Table is made up of records on the Agentry Client. Each record in the table is made up of Fields. Within a Complex Table definition in the Agentry Editor, you define the fields that make up the table's records.

Complex Table Field Settings

- **Name:** This is the name used to uniquely identify the field within the Complex Table.
- **Display Name:** This is the text value displayed on the Agentry Client to for the field. This includes the column headers in a Complex Table Search screen, as well as other places.

- **Type:** This attribute specifies the data type of the field. This is discussed further shortly.
- **No. of Characters:** This attribute is available only for one of the String type of fields and specifies the maximum number of characters the field can hold. Note that this is not necessarily the same as the number of bytes, as is explained in the section of the field data types. When setting this attribute, the value should be large enough to accommodate the strings the records will contain. However, it should not be simply set to an overly large value, as this will waste significant resources on the Client, both in storage and memory.

Field Data Types

There are six data types possible for a Complex Table Field definition. Each controls, first, the type of data that can be stored in the field and, second, how that data is sorted within the table. This last aspect can have a significant impact on how a user can search the complex table on a particular field. Following, each of the data types for a field are listed, along with a description of the impact each type has.

- **ASCII String (case-insensitive):** This field type specifies that the field will contain string characters, each one byte in length. This will support the standard ASCII characters. The case-insensitive portion indicates that, when the table is searched or sorted on this field, the case of the characters is not considered. That is, the lett ‘A’ is treated as equal to the letter ‘a’.
- **ASCII String (case-sensitive):** This field type specifies that the field will contain string characters, each one byte in length. This will support the standard ASCII characters. The case-sensitive portion indicates that, when the table is searched or sorted on this field, the case of the characters is considered. That is, the letter ‘A’ will be sorted after the lett ‘a’.
- **International String (case-insensitive):** This field type specifies that the field will contain string characters in the UNICODE format. This supports the non-english language characters, such as those in Hebrew or Chinese. Note that this also includes characters with an accent mark. This field type will still include the ASCII characters, as well. The case-insensitive portion indicates that, when the table is searched or sorted on this field, the case of the characters is not considered. That is, the letter ‘A’ is treated as equal to the letter ‘a’.
- **International String (case-sensitive):** This field type specifies that the field will contain string characters in the UNICODE format. This supports the non-english language characters, such as those in Hebrew or Chinese. Note that this also characters with an accent mark. This field type will still include the ASCII characters, as well.
- **Number:** This field type specifies that the field will contain numeric values only. These values may be whole numbers or decimal values and may be positive or negative. If you wish to index a field containing numerical values for the purpose of providing search functionality to the user on the Agentry Client, that field type should be string. Currently, Agentry does not contain the control types on the Agentry Client to support searching numerical values in a complex table.
- **Identifier:** This field type specifies that the field will contain numeric values only. These values may only be whole, positive values. Decimal and negative values are not supported. The purpose of this field type is to explicitly support an identifier field for each record.

Note that this is not a requirement, as the other field types can also be used as the identifier value for a record. This is covered in detail in the section in Indexes later in this chapter.

Complex Table Indexes

A complex table index definition orders the records of that table by the field for which the index is created. A field must be indexed to allow for the table to be searched on that field. A complex table can have multiple indexes. Indexes can be defined to have a parent-child relationship to give structure to the table's records.

The Index definition is the most important of the Complex Table. It is this definition type that makes the Complex Table so useful. When an index is defined, you specify the field to be indexed. When the Complex Table is downloaded to the Agentry Client, its records will be sorted by the fields you have indexed. Only those fields that have been indexed can be searched by the user on the Agentry Client.

Additionally, all Complex Tables must contain at least one index. This is the primary index of the table. The field for this index must contain the unique value for each record in the table. Whenever you define the indexes for a Complex Table, the first index defined is the one treated as the primary index. This cannot be changed once set, so be sure to determine which field should contain the Primary index beforehand. Also, any complex table definition that does not contain a primary index cannot be selected for use by any other definition in the application.

Though only those fields which contain an index can be searched on the Agentry Client, do not simply define one index for each field in the table. There is a certain amount of overhead that goes into each index definition. Also, whenever the records in the complex table are changed, each index must be resorted for each new, updated, or deleted record in the table. This also takes a certain amount of time and resources during a transmit. In a table with a large number of records, superfluous indexes can result in an unnecessary delay for users during transmit.

Complex Table Index Attributes

- **Name:** The internal name of the index. This value must be unique among all index definitions within the same parent complex table.
- **Display Name:** The value displayed for the index definition on the Agentry Client's user interface. In most contexts the index is, to the user, the same as the field and it is a common practice to set the display name of the index to match the display name of the field for which it is defined.
- **Field:** The field for which the index is being created and by which the complex table records will be sorted.
- **Parent Index:** The parent index, set to create parent-child indexes within the complex table.
- **Order:** This attribute specifies the order in which records should be sorted; either ascending (default) or descending.

Parent-Child Indexes

In addition to index a field within the complex table, indexes can also be defined to have parent indexes. This can allow you to create a parent-child relationship among the records of a complex table. The Primary index cannot be defined to have a parent index.

This structure can be very useful when the records of the table support this kind of relationship. One example of such data would a complex table containing locations, with each record representing one location within an industrial park. These locations can be structure to have parent-child relationships and the indexes for the complex table can be created to support this. In this case, a parent location could be a building. Within this building there may be five child locations, one for each floor. Within the first floor of the building, there may be 20 child locations, one for each office suite. Within the first suite, there may be 15 child locations as well, one for each room within the suite. Within the Complex Table, each record would contain, among the other fields, one for the location's ID and one for its parent location, named LocationID and ParentID, respectively.

When defining the indexes for the complex table, an index could be defined on the ParentID field, named ParentIDIdx. Then, a second index definition can be defined for the LocationID field, and this index would have a parent index of the ParentIDIdx index. Within the user interface definitions in Agentry, there are the field types used to create a Cascade. If a cascade were defined for the Locations complex table, the user would first be required to select the Parent ID. Then, they would be presented with a list of just those records in the table with a parent ID equal to the one selected. In the Agentry Editor, these controls are defined to use these parent and child indexes.

SQL Complex Table Synchronization Components

When a complex table is defined to use a SQL Database system connection type, the synchronization components consist of three SQL statements: Reload State Query, Deleted Query, and Data Query.

The reload state query can be enabled or disabled based on preference. When enabled, this query is expected to return the text values “true” or “false.” When the query returns true, the complex table will set to its rebuild state. The condition under which this query returns true is completely dependent on the need of the application or implementation. Its intent is to select from the back end system based on some value or condition that an administrator can easily set when it is desirable to force the complex table to be fully reloaded on the Agentry Client. When this query is disabled, it will not be run by the Agentry Server during synchronization for the complex table.

The data query is always run during synchronization and should include two separate select statements. Both statements are expected to return records from the database to the Agentry Server containing the field values for the complex table records. The columns of this return set must be named to match the names of the complex table fields. The difference between the two statements contained in the data query is the logic related to which records they will select. One statement should be written to select all records to be stored in the complex table on the

Agentry Client and under the assumption that the Agentry Client currently contains no records. This statement will then be run for only the rebuild state. The second statement should include logic in support of the exchange data model of synchronization, and should retrieve only new or modified records from the database that will be updated to the records stored on the Agentry Client. To determine if the complex table is in a rebuild state, the SDML data tag `<<rebuild>>` is used. This tag will return true when the rebuild state is set, and false when it is not. The data query will likely check this data tag using the `<<if...>>` function tag, which should then return the appropriate statement.

The deleted query is only run when the complex table is not in a rebuild state. This query is expected to return a single column identified as the key field in the complex table. Any values returned by this query will be sent to the Agentry Client so that the Agentry Client will delete the records with the matching key field value from the complex table.

Reload State, Data, and Deleted Query Attributes

- **Enabled:** This attribute is only found for the reload state query. It specifies whether the reload state query is enabled or disabled. The reload state query is only run during synchronization when it is enabled.
- **File:** All three query components contain the File attribute. It specifies the location of the text file (`.sql` file extension) relative to the Agentry Development Server's installation location.

Java Complex Table Synchronization Components

When a complex table is defined to use a Java Virtual Machine system connection type, its synchronization component consists of a Java source file. This file contains a skeleton class declaration. This class is created specific to the complex table definition and extends the Agentry Java API class `ComplexTable`.

When the table is defined, the wizard for creating Java classes provided with the Eclipse Java perspective is used. This allows the developer to select the package to which the new class will be added. The source `.java` file created will then be stored according to the configuration of the project and package selected for the new class. Alternately an existing class in a package within the Java perspective may be selected. This class must extend the Agentry Java API class `ComplexTable`.

This skeleton class includes the following methods:

- The Constructor method for the class
- An override implementation of `dataIterator()`. This method is intended to contain the logic to retrieve the data from the back end system for the complex table records. It returns an iterator to the data object created to store this returned data. The Agentry Server calls this method during synchronization and uses the returned iterator to extract the data for the records from the array of data objects. Records returned by this method are sent to

the Client to be added to the complex table, or to replace those records with matching key field values.

- An override implementation of `deleteIterator()`. This method is intended to contain the logic to retrieve the key field values from the back end system for the complex table records to be deleted from the client application. It returns an iterator to an array of the data object created to store this returned data. The Agentry Server calls this method during synchronization and uses the returned iterator to extract the key field data for the records from the array of data objects. Records returned by this method are deleted from the Client.
- The method `willRebuildTable()` can be created within the complex table class if needed.This method is called by the Agentry Server after the constructor method has been called. Its logic should check for any administrator defined conditions within the back end system to force a complex table rebuild. This method is expected to return a Boolean value. True will set the rebuild state for the complex table.
- The method `build()` can be created within the complex table class if needed. This method is provided to allow for a single call to the back end system to retrieve new, updated, and deleted records. If a build method is present it will be called by the Agentry Server before the iterator methods. In this scenario, the iterator methods are still expected to provide access to the returned data. However, the build method will have already retrieved it. If two separate calls are needed to retrieve updates to the table and deletions from the table, the `build()` method should not be used. The logic for those separate calls should be contained in the iterator methods, which are always called by the Agentry Server.

In versions of the Agentry Mobile Platform prior to 5.1, the source class was stored on the Agentry Server’s file system. This behavior is deprecated in versions 5.1 and later. Agentry application projects created prior to this release are still supported and the Java logic will still be processed correctly. New complex tables for Java Virtual Machine system connections should use the new procedure for defining the Java synchronization component.

HTTP-XML Complex Table Synchronization Components

When a complex table is defined to use an HTTP-XML system connection, its synchronization components consist of three HTTP request child definitions: **Update Request**, **Rebuild Request**, and **Deleted Request**.

Each request has the same overall structure and attributes, which includes the URL for the request and the request method. Likewise, the request argument and response mapping child definitions also contain the same attributes. The difference between these requests is when they are sent to the back end system, and what the data they are expected to return is used for in relation to the complex table.

The update request definition is sent to the back end during normal synchronization. This request is expected to return data for the complex table representing records to be added or replaced on the Agentry Client. Therefore it should contain one child data mapping definition for each field in the complex table.

The rebuild request is sent to the back end system when the complex table is in a rebuild state. This request is expected to return the data for all records that should be stored in the complex table on the Agentry Client. The rebuild state means the complex table on the Agentry Client is going to be cleared of all records before the request is sent. This request should contain child data mapping definitions for each field in the complex table.

Complex Table HTTP Request Child Definitions

Following is a list of the child definitions for each of the HTTP Requests within an HTTP-XML complex table.

- **Request Arguments:** This definition encapsulates an argument to be passed with the request to the back end system. Includes the ability to use data within the mobile application with the argument.
- **Response Mappings:** This definition encapsulates the XML data returned by the HTTP request. The specific values are extracted from the XML return data using XPaths defined within each response mapping. Attributes are also set to map the extracted values to data structures within the mobile application.

Complex Table HTTP Request Attributes

The following attributes are set in all three HTTP request definitions within an HTTP-XML Complex Table.

- **Name:** The name of the request, set automatically based on the parent complex table name and the request type. May be modified as needed.
- **URL:** The URL to the specific CGI or other process being called by the HTTP request to synchronize the complex table.
- **Method:** The HTTP request method for the request. May be one of GET, POST, HEAD, or PUT.

HTTP Request Argument

The request argument definition encapsulates a data value to be passed from the mobile application to the process being called by the parent HTTP request definition. The request argument specifies the argument type, which may be CGI Argument, Cookie, HTTP Header, or XML Body. The request argument also specifies the data or data source within the mobile application to pass as the argument to the process or service being called by the parent HTTP request definition.

For a complex table, the data value may be the user ID, the name of the complex table, a fixed string whose value is defined as a constant within the request argument, or markup text. A given parent HTTP request may contain multiple request arguments. The order in which they are passed to the process or service when called is defined in the parent HTTP request’s list of request arguments.

HTTP Request Argument Attributes

The attributes of a request argument depend in part on the data type of the argument (Data Type attribute). The following list makes note of those attributes specific to a certain argument data type.

- **Argument Type:** This attribute specifies the type of argument the definition contains. This may be one of CGI, Cookie, HTTP Header, or XML Body.
- **Name:** Alternately displayed as Argument Name, Cookie Name, Header Name, or Name depending on the Argument Type selection. This value must be unique among all request arguments defined within the same parent HTTP request definition.
- **Data Type:** Specifies the data value or source for the data value for the request argument. For a complex table this may be the Table Name, User ID, Small or Large Markup, or Fixed String.
- **String:** This attribute is available only when the **Data Type** attribute is set to Fixed String. String contains the constant string value that is the request argument’s data.
- **Markup Text:** This attribute is available only when the **Data Type** attribute is set to Small Markup or Large Markup. **Markup Text** contains the single line (Small Markup) of markup text or the contents of the Markup File (Large Markup) that is the data for the request argument.
- **Markup File:** This attribute is available only when the **Data Type** attribute is set to Large Markup. Markup File contains a reference to the text file containing the multi-line markup text. This file is displayed in the **Markup Text** field directly below the file name in the Editor.

HTTP Request Response Mapping

The response mapping definition is a child to an HTTP request definition. This definition maps a data value returned from the process called by the HTTP request to a value within the mobile application. This value may be extracted from structured XML using XPaths or XSL. It may also be a Cookie value or the HTTP Header.

For a complex table the values may be mapped to the fields in a complex table record, an error message, the last update value to be compared against the complex table’s last update, a local data tag or local XML value, or to the user ID value that may be used in place of the ID entered to log into the Agentry Client.

HTTP Request Response Mapping Attributes

The response mapping attributes are in part dependent on the selection made in the Mapping Type attribute. Those specific to a certain type are denoted in the following list.

- **Mapping Type:** This attribute specifies the mapping type. This may be one of Cookie, HTTP Header, XPath Expression, or XML Transformation.
- **Base XPath:** This attribute is only available when the **Mapping Type** is set to XPath Expression or XSL Transformation. This attribute is optional and should be used when returning multiple instances of the same data element in the XML content. When a **Base**

XPath is defined for a response mapping, the same value will be set by default in the add wizard for subsequent response mappings within the same parent HTTP request definition.

- **XPath:** This attribute is only available when the **Mapping Type** is set to XPath Expression. This attribute contains the XPath value to extract the desired value from structured XML data returned from the HTTP Request.
- **XSL:** This attribute is only available when the **Mapping Type** is set to XML Transformation. It contains the XSL expression to transform the XML data to be mapped to a value for the request.
- **Cookie Name:** This attribute is only available when the Mapping Type is set to Cookie. It contains the name of the cookie for the response mapping.
- **Header Name:** This attribute is only available when the Mapping Type is set to HTTP Header. It contains the name of the HTTP header for the response mapping.
- **Maps To:** This attribute specifies where the value extracted by the response mapping is stored in the mobile application. This may be one of the following values for a complex table:
    - **Complex Table Field:** This is the default selection and will result in the value being mapped to the selected complex table field in the table records. This enables the field Field Name, where the complex table field to which the return value is mapped.
    - **Error Message:** This selection will map the data to error text display by the mobile application.
    - **Last Update:** This selection specifies the extracted value is a date and time indicating when the complex table's source in the back end system was last modified. This value is mapped to each record. However, the latest date and time value for all records is the one stored with the complex table on the Client.
    - **Local String (`<<local>>`):** This selection will create a local data tag available to subsequent data mappings in the same parent HTTP Request. The value of this data tag will the value extracted by the response mapping. When selected, the attribute String Name will be available to name the local data tag created. This is the equivalent to calling the SDML function tag `<<local ...>>`.
    - **Local XML (`<<localXML>>`):** This selection will create a local XML data tag available to subsequent data mappings in the same parent HTTP Request. The value of this data tag will be the value extracted by the response mapping.
    - **User ID:** This selection will map the value extracted by the response mapping to the user's ID. This value is the equivalent to the SDML data tag `<<user.id>>`. If a previous response mapping in any HTTP Request processed by the Server set the user ID, setting it here will override that value. This will then be the value available for all HTTP-XML system connection processing where the data tag `<<user.id>>` is referenced.
- **Field Name:** This attribute is available when the map type is set to Complex Table Field. This attribute specifies the complex table field to which the values extracted by the mapping is assigned in the complex table records.

- **String Value:** This attribute is available when the map type is set to Local String or Local XML. It contains the name of the data tag being created by the mapping. This is the name by which the data tag is referenced in subsequent references.

Transmit Configuration

The transmit configuration defines how the application on the Agentry Clients can communicate with the Agentry Server. It can define what application-level data definitions to synchronize and the address and port number of an Agentry Server. It also defines whether to log a user out of the Server when a transmit has completed, or to keep them connected to provide real-time communications functionality.

The areas of the communications behavior include:

- The Agentry communications protocol to be used.
- What actions to take, if any, in the event of a communications error.
- Whether or not data tables and complex tables should be synchronized.
- Address and port numbers for the Agentry Server and Midstation.
- Whether or not the client should remain logged in for real-time communications.
- Various aspects of using a modem, such as the Windows Network connection to use, what to do if not currently connected, and other related behaviors.

A transmit configuration is defined for an available communications method on the client device. As an example, if a wireless LAN connection will be available to client devices in the deployment, a transmit configuration for this connection should be defined. If a wireless WAN connection is also available, a transmit configuration should also be defined for this connection type.

Each transmit configuration defined in the application will be listed in the built-in screen called the Transmit Dialog on the Agentry Client. For this reason it is important to consider the proper setting for each transmit dialog's Display Name, as it is this value that will be listed to the user.

Within the transmit configuration's attributes is the Connect Type. This attribute can be set to one of three options:

- Agentry Next Generation Encryption Layer, or ANGEL.
- Midstation
- Unencrypted Network Connection

Each of these connection types perform communications using the TCP/IP protocol. The connection type refers to the type of connection in the context of the application to be used when the transmit configuration is selected by the user on the client application.

Syclo recommends the ANGEL connection for all applications developed going forward. Applications developed using versions of the Agentry Mobile Platform prior to 4.4 being upgraded to the latest platform should be modified so that all transmit configurations use the ANGEL connect type. The Midstation and unencrypted Network Connection types are still available for the purposes of backwards compatibility and will be deprecated in a future release of the platform.

Transmit Configuration Attributes

General Settings:

- **Name:** This is the unique name of the transmit configuration. This value must be unique among all transmit configurations defined for the application.
- **Display Name:** This is the text displayed to the user on the Agentry Client for the transmit configuration in the Client’s Transmit Dialog.
- **Connect Type:** This is the communications protocol the Agentry Client is to use when synchronizing with the Agentry Server The options here are ANGEL or Unencrypted Network Connection. This attribute will not be definable in a future release of Agentry and all transmit configurations will use the ANGEL protocol.
- **Group:** This is the group into which the transmit configuration is organized within the application. This designation is provided for organizational purposes, and there are two options available by default: Fast and Slow. A new group can be created by entering a name in this field. It will then be available in this same drop down for all transmit configurations within the same application project.
- **Failover to:** This attribute can be set to any other transmit configuration within the application. When set the Agentry Client will switch to the selected transmit configuration if it is unable to connect the Agentry Server using the first transmit configuration.
- **Check Data Tables:** This attribute specifies whether or not the data tables within the application should be synchronized when the transmit configuration is used. This is normally unchecked for transmit configurations intended for slower connection types.
- **Check Complex Tables:** This attribute specifies whether or not the complex tables within the application should be synchronized when the transmit configuration is used. This is normally unchecked for transmit configurations intended for slower connection types.

Server Address Settings:

- **Address:** This attribute can be set to the IP address or network name of the host system for the Agentry Server. When set to default, the host will be the one entered by the user on the Client during the initial transmit.
- **Port:** This attribute can be set to the port number of the Agentry Server with which the Client is to connect using the transmit configuration. This is normally set to allow for multiple Agentry Servers running on the same host system, or to allow access through a firewall between the Client and Server.

Transmit Configuration - Session Attributes

General Settings:

- **Stay Logged In:** This attribute controls whether the client user will remain logged in and the Agentry Client will remain connected to the Agentry Server. The purpose of setting this attribute is to support real-time communications within the mobile application, which includes Background Sending and Push behaviors. This requires a constant network connection be available to the Client's. This attribute must be set for any of the other Session attributes to be enabled for the transmit configuration.

- **Prompt on Log In:** This attribute applies when a the connection between the Agentry Client and Agentry Server is lost, and when the transmit configuration is defined to attempt to reconnect. A prompt can be displayed to the user in this situation when the connection is re-established, or hidden from the user based on this attribute setting.
- **Prompt on Log Out:** This attribute controls whether or not the user is prompted when the Agentry Client is logged out of the Agentry Server. When set to false, no prompt is displayed. This only applies when the connection for the transmit configuration is lost and the transmit configuration is set to stay logged in.
- **Inactive Timeout:** This attribute specifies the time limit, in hours, minutes, and seconds, the Agentry Client should remain connected to the Agentry Server with no activity. Activity is defined as the transmission of data between the Client and Server.
- **When Off-line:** This attribute controls whether or not the Agentry Client should attempt to reconnect to the Agentry Server when the connection has been lost. If the Client should reconnect, the duration of time to wait before attempting to reconnect is set in minutes and seconds.
- **Attempts:** This numeric attribute is set only when the **When Off-line** attribute is set to reconnect. The attempts attribute defines how many attempts to make before failing. If the number of attempts is tried without success, the behavior of the Agentry Client is dictated by the transmit configuration's **Failover** to attribute, as well as the **Prompt on Log In/Log Out** attributes.

Background Sending:

- **Allow:** This attribute enables background sending of pending transactions on the Agentry Client. When this attribute is enabled the Client will attempt to send transactions to the Agentry Server in the background as soon as they are applied.
- **Retry Period:** This attribute specifies the amount of time in hours, minutes, and seconds to wait between failed attempts to send a transaction in the background.
- **Attempts:** This specifies the number of attempts to make at sending a transaction in the background before failing.

Push Session:

- **Allow:** This attribute enables Server Push functionality. This functionality also requires the definition of a push within a module of the application. When enabled, users connecting the server using the transmit configuration will be logged in to the Agentry Server as a Push User. This also opens the Agentry Client to receiving push data.
- **Retry Period:** This attribute specifies how long the Agentry Server should wait before attempting to re-send data for a push when a failure occurs.
- **Attempts:** This attribute specifies how many attempts the Agentry Server should make to re-push data before failing.

- **Client Port:** This attribute specifies the port upon which the Agentry Client listens for push communications from the Server. The default port is 7001.

Transmit Configuration - Modem Connection Attributes

- **Check for Modem Connection:** This attribute specifies whether or not the Agentry Client should check for a modem connection when the transmit configuration is used. If true, this check is made prior to beginning the transmit. This attribute must be true for any of the remaining modem attributes to be enabled.
- **Connection Name:** This attribute can be set to the name of any Windows network connection configured on the client device. It can also be set to Any Dial-Up Connection. In the case of the former, it will use the settings of the named connection to establish the modem connection to the network. If set to “Any Dial-Up Connection,” the user will be required to establish the network connection manually outside the mobile application before beginning the transmit. In this case, the remaining modem connection attributes are not enabled.
- **If Not Connected:** This attribute specifies whether the Agentry Client should attempt to create a connection using the Windows network connection named in Connection Name when there is no current connection. If this is set to false, the remaining modem connection attributes are disabled.
- **Connect Prompt:** This attribute is set to the message to display to the user prior to the Agentry Client attempting to create a modem connection. If this attribute is left blank, no message will be displayed to the user prior to creating the connection. Normally the contents of this message prompt the user to connect a phone line or perform similar actions in order for the connection to be made.
- **Username:** This attribute prompts the user to enter a user name for the network connection. This will be used as the login name for the network connection once the modem’s hand shaking processes are successful. If this is set to false, the users Agentry Client login will be used.
- **Password** - This attribute prompts the user to enter a password for the network connection. If this attribute is set to false, the users Agentry Client password will be used.
- **Modem Init Wait:** This attribute specifies the amount of time in milliseconds the Agentry Client should wait for the client device’s modem to initialize before beginning the dial-up process.
- **Post-connect Wait:** This attribute specifies the amount of time in milliseconds the Agentry Client should wait after the network connection has been made before beginning the transmit process between the Client and Agentry Server.
- **Close Connection:** This attribute specifies whether the connection made by the transmit configuration should be closed if no data has been transmitted between the Agentry Client and Agentry Server for the specified amount of time. The attribute can be set to Never, meaning the connection will not be closed, or to the minutes and seconds to wait before closing the connection.

System Connection

A system connection sets the connection type the Agentry Server will use to synchronize data with a back end system. A system connection specifies what type of system the Agentry Server is communicating with: SQL Database, Java Virtual Machine, HTTP-XML Server, or File System.

An Agentry application project must have at least one system connection. More system connections can be added if the application requires the Agentry Server to communicate with multiple back end systems. Each system connection may be of different types, or multiple connections for the same type can be defined, depending on the environment in which the mobile application will run.

There are four supported System Connection types:

- **SQL Database** - This system type is used when the Agentry Server needs to communicate with a database system using the Structured Query Language, or SQL. This includes database types such as Oracle or SQL Server.
- **Java Virtual Machine** - This system type is used when the Agentry Server needs to communicate with an interface using the Java Virtual Machine. This logic is implemented using the Java development language and includes usage of the Agentry Java API.
- **HTTP/XML Server** - This system type is used when the Agentry Server needs to communicate with an HTTP server by making HTTP requests that will return structured XML data.
- **File System** - This system type is used when the Agentry Server needs to communicate with the host system upon which the Server has been installed, specifically for file access or command-line processing.

The SQL Database and File System connection types have only the two attributes of Name and the ID number. The name is set by the developer when the system connection is defined. The ID number is generated automatically by the Agentry Editor. This ID number ties the definition to the set of configuration options, configured in the Agentry Server.

The Java Virtual Machine connection type contains the additional attribute API Version. This attribute specifies the version of the Agentry Java API to be used by the mobile application. For all new development, version 5 of this API should be used. Version 4 is available for existing applications developed on versions of the Agentry Mobile Platform prior to the version 5.0 release.

A system connection defined for the HTTP-XML connection type contains the child definition type Validate User Request. This is an HTTP Request definition intended to validate the client user, as well as to capture user information to be stored in the `<<user.info>>` SDML data tag.

Validate User Request

When a system connection is defined for an HTTP-XML connection type, it can contain one or more HTTP Request child definitions called Validate User Requests. These requests can be made to validate the client user during transmit. This request can also be used to create one or more `<<user.info>>` SDML data tags.

The validate user request is sent to the back end system at the beginning of the transmit process as a part of the user validation behavior. Each validate user request definition includes child definitions to encapsulate the request arguments, as well as those to map any data returned by the request to structures within the mobile application.

Validate User Request Child Definitions

- **Request Arguments:** This definition encapsulates an argument to be passed with the request to the back end system. Includes the ability to use data within the mobile application with the argument.
- **Response Mappings:** This definition encapsulates the XML data returned by the HTTP Request. The specific values are extracted from the XML return data using XPaths defined within each response mapping. The mapping “maps” the extracted values to values within the mobile application.

Validate User Request Attributes

- **Name:** The name of the request, set by default to ValidateUser. May be modified if desired.
- **URL:** The URL to the specific CGI or other process being called by the HTTP request.
- **Method:** The HTTP request method for the request. May be one of GET, POST, HEAD, or PUT.

Validate User Request Argument

The request argument definition encapsulates a data value to be passed from the mobile application to the process being called by the parent validate user request definition. The request argument specifies the argument type, which may be CGI Argument, Cookie, HTTP Header, or XML Body. The request argument also specifies the data or data source within the mobile application to pass as the argument to the process or service being called by the parent validate user request definition.

For an HTTP-XML system connection, the data value may be the user ID, the user’s password, a fixed string whose value is defined as a constant within the request argument, or markup text. A given parent validate user request may contain multiple request arguments. The order in which they are passed to the process or service when called is defined in the parent validate user request’s list of request arguments.

HTTP Request Argument Attributes

The attributes of a request argument depend in part on the data type of the argument (Data Type attribute). The following list makes note of those attributes specific to a certain argument type.

- **Argument Type:** This attribute specifies the type of argument the definition contains. This may be one of CGI, Cookie, HTTP Header, or XML Body.
- **Name:** Alternately displayed as Argument Name, Cookie Name, Header Name, or Name depending on the **Argument Type** selection. This value must be unique among all request arguments defined within the same parent validate user request definition.
- **Data Type:** Specifies the data value or source for the data value for the request argument. For a complex table this may be the User ID, user's password, Small or Large Markup, or Fixed String.
- **String:** This attribute is available only when the **Data Type** attribute is set to Fixed String. String contains the constant string value that is the request argument's data.
- **Markup Text:** This attribute is available only when the **Data Type** attribute is set to Small Markup or Large Markup. **Markup Text** contains the single line (Small Markup) of markup text or the contents of the Markup File (Large Markup) that is the data for the request argument.
- **Markup File:** This attribute is available only when the **Data Type** attribute is set to Large Markup. Markup File contains a reference to the text file containing the multi-line markup text. This file is displayed in the **Markup Text** field directly below the file name in the Editor.

Validate User Request Response Mapping

The response mapping definition is a child to a validate user request definition. This definition maps a data value returned from the process called by the HTTP request to a value within the mobile application. This value may be extracted from structured XML using XPaths or XSL. It may also be a Cookie value or the HTTP Header.

For an HTTP system connection the values may be mapped to the user ID, validation, partial validation, the `<<user.info>>` set of SDML data tags, an error message, a local data tag, or a local XML data tag.

HTTP Request Response Mapping Attributes

The response mapping attributes are in part dependent on the selection made in the Mapping Type attribute. Those specific to a certain type are denoted in the following list.

- **Mapping Type:** This attribute specifies the mapping type. This may be one of Cookie, HTTP Header, XPath Expression, or XML Transformation.
- **Base XPath:** This attribute is only available when the **Mapping Type** is set to XPath Expression or XSL Transformation. This attribute is optional and should be used when returning multiple instances of the same data element in the XML content. When a **Base XPath** is defined for a response mapping, the same value will be set by default in the add wizard for subsequent response mappings within the same parent HTTP request definition.

- **XPath:** This attribute is only available when the **Mapping Type** is set to XPath Expression. This attribute contains the XPath value to extract the desired value from structured XML data returned from the HTTP Request.
- **XSL:** This attribute is only available when the **Mapping Type** is set to XML Transformation. It contains the XSL expression to transform the XML data to be mapped to a value for the request.
- **Cookie Name:** This attribute is only available when the Mapping Type is set to Cookie. It contains the name of the cookie for the response mapping.
- **Header Name:** This attribute is only available when the Mapping Type is set to HTTP Header. It contains the name of the HTTP header for the response mapping.
- **Maps To:** This attribute specifies where the value extracted by the response mapping is stored in the mobile application. This may be one of the following values for a complex table:
  - **User ID:** This selection will map the value extracted by the response mapping to the user’s ID. This value is the equivalent to the SDML data tag `<<user.id>>`. If a previous response mapping in any HTTP Request processed by the Agentry Server set the user ID, setting it here will override that value. This will then be the value available for all HTTP-XML system connection processing where the data tag `<<user.id>>` is referenced.
  - **Validation:** This selection will map the value extracted by the response mapping to the validation structures for the Agentry Server. This value is used to indicate whether or not the user passed validation.
  - **Partial Validation:** This selection will map the value extracted by the response mapping to the validation structures for the Agentry Server. This differs from the Validation selection in that mapping the validation result to partial validation can fail user validation with a false response, just as the validation response will, but true for Partial Validation will not fully validate the user. This is intended to provide support for validation using multiple system connections.
  - **Error Message:** This selection will map the data to error text display by the mobile application.
  - **Local String (`<<local>>`):** This selection will create a local data tag available to subsequent data mappings in the same parent HTTP Request. The value of this data tag will the value extracted by the response mapping. When selected, the attribute String Name will be available to name the local data tag created. This is the equivalent to calling the SDML function tag `<<local ...>>`.
  - **Local XML (`<<localXML>>`):** This selection will create a local XML data tag available to subsequent data mappings in the same parent HTTP Request. The value of this data tag will be the value extracted by the response mapping.
  - **Save to User info:** This selection will map the value extracted by the response mapping to the set of data tags in the `<<user.info>>` group. When this selection is made, you will also be required to enter a name for the data tag. Referencing these values is then accomplished via the syntax `<<user.info.name>>`.

- **String Value:** This attribute is available when the map type is set to Local String or Local XML. It contains the name of the data tag being created by the mapping. This is the name by which the data tag is referenced in subsequent references.
- **With Name:** This attribute is available when the map type is set to Save to User Info. it contains the name of the data tag being created by the mapping. This is the name by which the data tag is referenced in subsequent references.

Global

A global definition defines a constant value, including data type, for the application. This value can be referenced throughout the application, both by the attributes of other definition types and for use in synchronization components. A global value cannot be changed on the Client at run-time but can be overridden during synchronization.

A global's value is constant and cannot be modified on the Agentry Client. It can be overridden at run time during synchronization.

The value of a global definition is dependent on the global's data type. Following is a list of the global data types:

- **Boolean:** A value that may be either true or false.
- **Date:** A value representing a calendar date.
- **Date and Time:** A value containing a calendar date and time of day.
- **Decimal Number:** A numeric value that contains a fractional portion and that may be positive or negative.
- **Duration:** A value containing a duration of time in hours, minutes, and seconds.
- **Identifier:** A numeric value that is primarily used to represent an identifying value. Can contain whole, positive numeric values.
- **Integral Number:** A numeric value containing whole numbers that may be positive or negative.
- **Selection:** A special data type for a global definition that represents an attribute setting that is selected from a list. This data type cannot be selected when defining a global, but rather is the automatic data type of the global when it is created specifically for an attribute whose setting is selected from a drop-down list in the Editor. Valid values for this type of global are those found in the specific list for the attribute.
- **String:** A value containing alphanumeric or other printable characters.
- **Time:** A value containing a time of day.

The data type of a global is important as it will determine where in the application the Global can be used. The data type of the attribute and the global definition used to set it must be the same. For example, string properties contain attributes for their size, i.e., the number of characters they can contain. This size attribute is an integral number. This then requires the user of a global with a data type of integral number.

A global definition may be added to the application project from either the list of globals for the application, or at the point where it will be referenced by another attribute that may be set via a global. In the latter case, the data type of the global is set automatically based on the data type for that attribute.

Global Attributes

- **Global Type:** The data type of the global, selected when the global is added to the application, or set automatically by the Editor based on the attribute to use the global for its setting.
- **Group:** The group into which the global definition will be organized. Unlike the group setting for other definitions, a global's group is a required attribute. References to the global definition throughout the application must include its group as well as its name.
- **Name:** The unique identifier for the global definition. This value must be unique among all global definitions within the same group.
- **Value:** The value of the global definition returned when the global is referenced. Valid values for a global depend on its defined **Global Type**.

Style

A style definition defines a set of style elements that can be applied to the Agentry Client's user interface to affect its appearance. These elements include text and background colors, font face and size, borders, and other similar UI items. A style may be defined for all supported application platforms or for a single platform.

The Agentry Editor allows the developer to create display styles for screens, buttons, text, fields and list controls. A style is defined as a collection of display elements combined to provide an overall look and feel to the application.

Styles exist at the application level in a project. They are then available to be used, or "applied" at the application, module, platform, screen and control levels. Each attribute, or "style element" of a style definition may be set to a specific value or default. Default results in the system default being used for that aspect of the user interface.

If styles are applied at multiple levels within the application they are merged at run time before being applied to the user interface. The style definition applied at a lower level in the application hierarchy will override the settings of a style applied at a higher level. If the lower level style has an element set to default, the setting for that same element in the higher level style definition will be used. This merge then results in the overall appearance of the user interface component to which the style is applied.

A style may defined for a specific platform. Multiple styles may be defined with the same name but with different platform selections. When a style is applied to the user interface, only the name is referenced. At run time, a given client device type will receive only the styles with a matching platform. This is optional behavior and a style may defined for all platforms.

iPhone and iPad/iPod Touch Platform Note

Due to the nature of the iOS devices, the current style support for these device platforms is limited to the specification of the Font Face and the Foreground Color. Styles can only be applied to specific controls and the affects of the two supported style attributes are the font in which text is displayed and the color of that text.

Style Attributes

Following is a list of the attributes for a style definition. In the context of a style definition these attributes are commonly referred to as “style elements” and the terms are interchangeable:

- **Foreground Color:** This attribute specifies the color of any text displayed on the user interface component to which the style is applied. If a particular user interface component has no text, the Foreground Color setting will have no affect on its appearance.
- **Background Color:** This attribute specifies the color of the background of the user interface component. The background of a screen or control is the area that contains no controls, text, or list items.
- **Font Face:** This attribute specifies the font used to display any text on the user interface to which the style is applied. Within the Agentry Editor, the **Font Face** attribute field contains a drop-down list. Its contents will be any fonts installed on the host system of the Editor. The name of a font may also be manually entered if it is one that is known to be available on the client devices, even if it is not available on the Editor’s host system. Any font face entered manually in the list will be available in this same list for all style definitions within the application. If a font name is entered that is not available on a client device, the behavior will be the same as if Default had been selected for the **Font Face** attribute.
- **Point Size:** This attribute specifies the size of the text displayed to the user. If the point size is larger than the viewable area given to that text value, that viewable area will not be increased in size.
- **Font Style:** This attribute specifies whether the text is displayed normally (referred to as the regular font style), or in bold, italics, or bold italics. The Font Style attribute may not have an effect on the appearance of the text based on the selected Font Face. Certain fonts are inherently bold or italicized, or may not support either behavior.
- **Underline:** This attribute specifies whether the text is underlined. This attribute may have no effect on certain Font Face selections, as the selected font may not support underline or may be inherently underlined.
- **Border Style:** This attribute controls how the border around certain UI components will appear. This includes detail screen fields and buttons. The border style can be None, Flat, or 3-D.
- **Text Alignment - Horizontal Alignment:** This attribute specifies the alignment of the text displayed by the UI component to which the style is applied. The options are:
  - “Align Left” - This selection specifies that the text is to be aligned to the left of the viewable area allotted for the text being displayed.
  - “Align Right” - This selection specifies that the text is to be aligned to the right of the viewable area allotted for the text being displayed.
  - “Center” - This selection horizontally centers the text within the viewable area allotted for the text being displayed.

- “Default” - This selection will horizontally align the text according to the default behavior of the item containing the text.

Image

An image definition incorporates an image file into the application data. This image can be displayed on various components of the Agentry Client’s user interface. An image can be used to add icons and interactive graphics to the UI for branding purposes and to enhance the user experience.

Once an image has been defined it can be referenced in several components of the user interface definitions. This can include button icons, list icons, and detail screen fields, as well as the login dialog, the module selection dialog, and the help dialog. The first group of definition types that may reference an image definition support the use of image lists, which can allow for the display of a different version of the image based on some condition.

When an image definition is created, it must use a file of one of the types:

- Bitmap
- JPEG/JPG
- GIF
- PNG

The selected file must exist prior to creating the image definition. The file is copied within the application project definitions. Modifications to the selected source file after this point will not effect the appearance of the image within the application. This image can be edited from within the Editor if necessary.

An image may be defined for a specific platform. Multiple image definitions may exist with the same name and different selected platforms. References to images from other definitions within the application are made by name only. At run time a given device type will receive the images defined only for the matching platform. This is optional behavior and an image definition can be created for all platforms.

Image Attributes

- **Name:** This is the internal name of the image definition. This value must be unique among all Images with the same setting in the Platform attribute.
- **Platform:** The platform attribute specifies the platform of the client devices to which the image will be downloaded. This can be either All, or one of the available platforms listed. Selecting a specific platform will prevent the image from being downloaded to any device of any other platform. This can be used to download images with different file sizes to different client devices while using the same name.
- **Image File:** This value is the file name that will be used to store the image file within the application project, as well as by the Agentry Server and Agentry Client. The default

Image File value is a combination of the **Name** and **Platform** attribute values. It is rarely necessary to change the **Image File** setting.

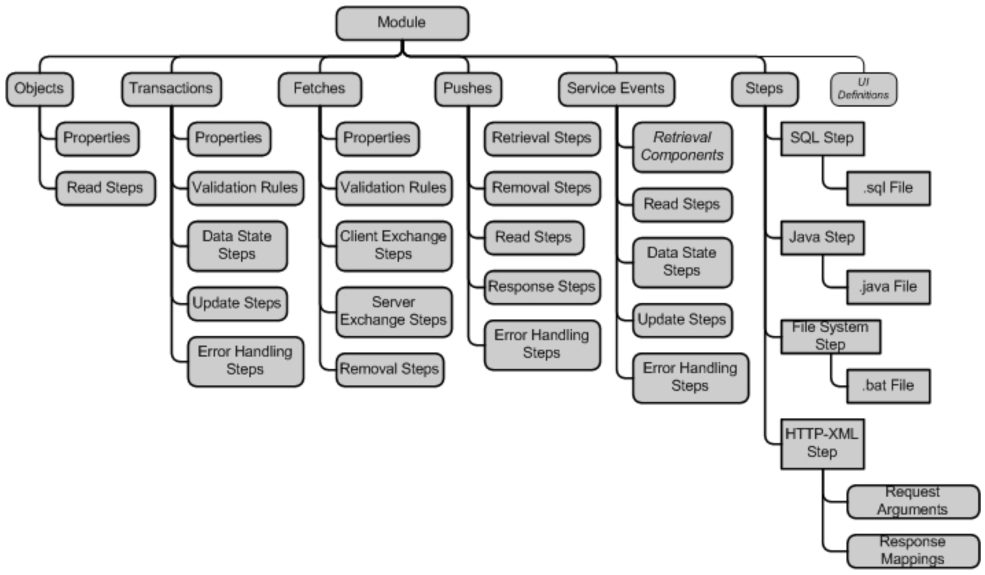
- **Mask Color:** This optional attribute can be used to create a mask color for the image, which will be incorporated into the image's display on the Agentry Client. This setting does not affect the image file itself, but rather is applied at run time. It is set using the RGB values, or by selecting the desired color from the Windows color palette. A mask color is used to remove a color from the displayed image, such as the white background of an icon.
- **Image:** This is the actual image file that contains the image. This is selected by clicking the ellipsis button to the far right of the field, which will display the standard Windows File Dialog. From here you can select the file to be used for the image definition. You will only be allowed to select files of the types `.bmp, .gif, .jpeg, .jpg, or .png`. Once the file has been selected it will be displayed in the image definition within the Agentry Editor.

Module-Level Data Definitions Overview

Within the module level of the application project in Agentry there are definitions for both data and user interface encapsulation. The data-focused definition types include those for business entity encapsulation, data capture, and data synchronization between the Agentry Client and Agentry Server.

Most of the data definitions at the module level have child definitions of their own. Each child definition encapsulates some aspect of the parent's behavior related to the data for which it was defined. This can include the values for the parent definition, or the methodology for data synchronization.

Following is an illustration representing the structure of the module-level data definitions within the application project. This includes the definitions within the module provided to encapsulate data storage or synchronization, as well as the child definitions to each. Excluded from this graphic are the user interface definitions within the module. Note that this separation is for discussion purposes only. Within the application project structure, all child definitions to the module exist at the module level with no distinction made between them in the Agentry Editor in relation to whether they are data or user interface definitions.



A common child definition to objects, transactions, and fetches are the properties. A property is a variable data value stored within the parent definition. The purpose for these values differs depending on the parent definition, but the property definition type itself is the same among all three.

Many of the child definitions to the module-level data definitions are referred to as “step usage definitions.” This term describes a definition that references a step definition within the same module. This reference provides the context to the step, specifying why and when it should be executed by the Agentry Server during synchronization. Any child definition to a module-level definition that includes the term “Step” in its name is a step usage definition. The creation of a step usage definition requires that the step to be used exists first.

As illustrated in this graphic, the step definition itself is defined for different types of processing. Steps are defined for a specific system connection within the application. The step definition has a type that matches the system connection type. The step will then contain a component matching that type, such as a SQL statement or Java logic. HTTP-XML steps include two child definitions that define the arguments passed to the HTTP server with the step request, and mappings between the data returned from that request to the data components of the mobile application.

The data definitions illustrated and described here are displayed, modified, and exposed to the mobile application uses via the module-level user interface definitions. The data definitions must exist before the user interface definitions can be created, as the UI definitions will need to reference the data definitions they display.

Object

An object definition encapsulates a business entity and its related data. An object’s child property definitions give that object its characteristics. An object can also define how its data is retrieved from the back end system.

The object definition is in essence a container for the properties defined within it. Objects are defined to encapsulate the different business entities in a module in support of the functionality to be provided in the mobile application. The properties then define the data stored within that object.

A special type of object will exist in every module defined within an application project. This is the module main object, named by default MainObject. The intent of this main object is to be the starting point, or top level of the module’s overall object data structure. Via the use of the collection property data type, object instances may be stored within other object instances at run time. This then results in a parent-child relationship within the module’s data structure. At the top of this structure is the module main object.

When a new module is defined, the module main object will be added automatically. Additionally, a prompt is displayed in the New Module Wizard for the definition of another object. The object defined in the New Module Wizard is normally the primary object for the module. The primary object is a term of convenience used to denote the object around which most of the module’s functionality will revolve. This includes the functionality provided to the end user in the form of information and data capture, as well as synchronization processes for the module. Examples of a primary object include a work order object for a work management module, or a message object for a mail module. The module main object will include a single collection property defined to store instances of the primary object.

Object Child Definitions

- **Property:** An object property defines a single piece of data for the parent object.
- **Object Read Step:** An object read step references a step definition run to retrieve data from a back end system to populate an object's properties.

Object Attributes

- **Name:** This is the unique name of the object. This value must be unique among all objects defined within the same module.
- **Display Name:** This is the default name displayed for the object on the Agentry Client.
- **Key Property:** The key property for an object is used whenever that object is to be a part of a collection. The value of this property must be a value that uniquely identifies the object and in most cases will be the same value as the key value from the back end system. Note that almost all object definitions are stored in collections and therefore must have this attribute set. The property to be used must be defined before setting this attribute.
- **Transmit Display Property:** This attribute specifies the object property value to display to the user on the Agentry Client transmit screen. When an object is being retrieved from

the back end system, the property specified here is displayed to the user during its retrieval. By default the value displayed is the object’s key property.

- **Main Object:** This attribute specifies whether or not the object is the main object for the module. Each module contains a main object. This attribute is set to true for that object, and to false for all other objects. This attribute is displayed for reference purposes within the Editor and cannot be modified.

Object Read Step

An object read step references a step definition within the same module. Its purpose is to retrieve data for instances of the object from the back end system. The steps are processed by the Agentry Server during a transmit. The step being referenced can be executed once per transmit or iteratively.

The data returned by the object read step is expected to be identified to match the property values of the object. How this data is identified is dependent on the type of step being executed. A given read step need not return all data values, but must always include the key property of the object type for which it is retrieving data and the key property of any parent objects up to the top-level object in the module’s data structure.

Object read steps are executed by the Agentry Server in any of the following situations related to the parent object definition:

- When a fetch is processed that is defined to target a collection of the read step’s parent object type.
- When a push defined to target a collection of the read step’s parent object type polls the back end for data changes and finds this to be true, and when that push is defined to use the object’s read steps to retrieve the data rather than the push read steps.
- When the processing of a transaction targeting the read step’s parent object type sends a client response of replace client object.

In any of these situations, the read steps for an object will be run and the data returned will be used to either create new object instances or replace existing object instances that will ultimately be sent to the Client.

It is important to note that the step being executed by the read step must account for which situation it is being run. The read step definition itself is not aware of the synchronization context in which it is being executed.

Object Read Step Attributes

- **Step:** This attribute references the step definition within the same module to run as an object read step for the parent object.
- **Run:** This attribute specifies how to run the read step during a single transmit. This may be set to one of the following values:
  - *Run one Time:* This setting will run the read step a single time for a given synchronization context. This setting assumes the step need be executed only once to

return the data for all object instances to be added or replaced during synchronization, or the step being executed is not returning data but rather is being run in support of synchronization.

- *Run once per Object:* This setting will execute the read step once for each object instance in the collection that is being synchronized. This includes both those object instances sent by the Client to the Server, as well as any that may have been added by previous synchronization steps. For push processing the step will be executed once for each object instance created prior to the read step's execution. For transaction processing this setting will have the same behavior as "Run one Time."
- *Run once per Collection Object:* This setting will execute the object once for each object instance in the child collection referenced by the Read Into attribute. This child collection is assumed to have been populated with object instances prior to the read step's execution. Note that this setting is primarily intended for file transfer functionality, though it is not limited to this purpose.

- **Read Into:** This attribute specifies the child or descendent object collection property of the read step's parent object into which the data returned by the step should be read. This attribute has a default setting of "None." This default means the data will not be read into a child collection but will instead be used to create object instances of the read step's parent object. Other valid options for this attribute are any child collection properties of the read step's parent object, or any descendent collections (e.g. collections within collections) of the parent object.

Object Property

An object property definition defines a single piece of data and its type. A property can also define minimum and maximum values, a default, or "special value" and other data-related behaviors.

The properties of an object define the aspects of the business entity the object is intended to encapsulate. Each object must include a key property that will uniquely identify each instance of the object at run time. The object key property is important to all aspects of object data synchronization. Both the Agentry Client and the Agentry Server use this value to determine if an object is added to a collection, or if it should be replaced. On the Agentry Client, a new object cannot be added to a collection using an add transaction if the instance it creates has the same key property value as an existing object instance.

The key property is specified in the object definition itself in the Key Property attribute. The property to be used as the key property must be defined first.

The attribute Name is an important one to consider when defining the properties of an object. In addition to uniquely identifying the property definition within the parent object, it also plays a part in the downstream synchronization of objects at run time.

When a step definition returns data for an object, the values returned will be identified in some manner depending on the type of step. For a SQL step this is the column name designated in the SELECT portion of the step's query. In a Java step, it is the name of the members of the `returnData` structure. HTTP-XML and File steps use different mechanisms involving mapping behaviors. Regardless of the step type, the name used by the step to identify a value must match the name of the property definition. The Agentry Server will populate a property with the value in the data returned by the step with the same name or identifier as that property definition's Name.

Properties are defined to be of a certain data type, of which there are many in Agentry. They are a child definition to the object, transaction, and fetch definitions. Each property data type has its own set of attributes specific to that type. Review the information on property data types for more detailed information on properties.

Transaction

The transaction definition defines data to be captured on the Agentry Client. As a part of its definition, the transaction includes a target object type, data values to be captured, client-side data validation, and processing its data to the back end system by the Agentry Server during synchronization. Transactions can add new object instances, edit an existing object, delete an object, or modify an complex table or data table record. Each of these behaviors is exhibited by a different transaction type, selected during the creation of the transaction.

A transaction definition is created within the application to target a specific object type within the same module. Transactions are instantiated on the Agentry Client one at a time as the result of the execution of a transaction step within a client action. A transaction instance can target only one instance of an object.

The transaction can be displayed to the user in a screen set, which will behave as a wizard allowing the user to enter data in a series of one or more screens.

There are five different types of transactions that can be defined for an application. Each captures data for a specific type of change on the Agentry Client. The transaction types are:

- **Add:** An add transaction type is defined to allow the user to create a new object instance on the Agentry Client.
- **Edit:** An edit transaction is defined to allow the user to edit the property values of an existing object instance on the Agentry Client.
- **Delete:** A delete transaction is defined to remove an object instance from the Agentry Client.
- **Data Table Change:** A data table change transaction is defined to allow the user to add or edit a data table record on the Agentry Client.
- **Complex Table Change:** A complex table change transaction is defined to allow the user to add or edit a complex table record stored on the Agentry Client.

Transaction Child Definitions

All transactions, regardless of type, have the same child definitions. The purpose of these child definitions is the same for all transaction types.

- **Properties:** A transaction property defines a single piece of data a transaction will capture, including its data type and initial value.
- **Validation Rules:** A transaction validation rule defines what rule definition will be used to validate the transaction's data and how failed validation is handled on the Agentry Client.
- **Server Data State Steps:** A transaction server data state step references a step definition within the same module to be run by the Agentry Server to check the back end system for data collisions during transaction processing.
- **Server Update Steps:** A transaction server update step references a step definition that is run during transmit to update the back end system with the data captured by the transaction.
- **Error Handling Steps:** A transaction error handling step references a step definition that is run during transmit if an error occurs while processing the transaction's data state or update steps.

Transaction Attributes

Transaction attributes specify the type of transaction, the object type it targets, the key property of the transaction, and the transaction's name and display name. There are also type-specific attributes for the different transaction types. Review the information on the specific transaction types for details on these attributes.

Transaction Authentication

Transaction authentication is definable behavior for all transaction types and is available to support user authentication during data capture, often referred to as "electronic signatures". To define this behavior, attributes specific to authentication must be set within the transaction definition after it has been defined. These attributes are not displayed in the add transaction wizard.

These attributes are used to define transaction authentication on the Agentry Client. This functionality provides the means to authenticate users when they make data changes. Using transaction authentication you can require users to enter their user ID, password, and other information as may be necessary.

Transaction authentication is defined for each transaction definition. This allows for authentication behavior to be exhibited only for data capture operations that require it.

This information can be captured in properties of the transaction itself, or in an instance of an object defined specifically for this purpose, called the authentication object. A separate screen set defined to display the authentication object to the user must exist prior to defining the authentication within the transaction.

Transaction Authentication Attributes

The following attributes are common to all transaction types. They are set to define the transaction authentication behavior. They can only be modified for existing transactions and are not displayed during the add transaction wizards.

- **Screen Set:** This attribute is set to the screen set to display to the user for the purpose of entering the authentication information you wish to capture. This can include the user ID, password, and other information as may be necessary. If this is set to “No Authentication” the authentication functionality is disabled for the transaction.
- **Authenticate When:** This attribute determines when the transaction requires authentication. This can be set to: “Do Not Authenticate”, disabling the behavior; “Always authenticate”, or can be based on a rule definition. When a rule is referenced by this attribute, it is evaluated in the context of the transaction and is expected to return a boolean value. A true return will require the user to authenticate. A false return will not and the authentication screen set will not be displayed.
- **Information In:** This attribute is set to either “Properties of this transaction” or to an object type defined within the same module. If set to the former, the properties displayed in the Authentication Screen Set are defined within the same transaction. If set to an object, the properties of that object are displayed and store the authentication data.

Transaction Type: Add

An add transaction type is defined to allow the user to create a new object instance on the Agentry Client. An add transaction definition includes a target object collection property to which the new object instance will be added. This transaction type should contain all non-collection properties found in the object type it creates.

Add Transaction Attributes

Following are the attributes for an add transaction:

- **Type:** This attribute specifies the type of transaction. For add transactions this is set to “Add”. This attribute cannot be changed once the transaction has been defined.
- **Object:** The Object attribute specifies the type of object the add transaction instantiates on the Agentry Client. This may be any existing object defined within the same module.
- **Collection:** The collection attribute specifies the collection property in which the new object instance will be stored on the Agentry Client when the transaction is applied.
- **Name:** This is the identifier for the transaction definition. This value must be unique among all transaction definitions within the same module.
- **Display Name:** The display name is the value shown to users for the transaction on the Client. This is normally seen by the user in the Agentry Client’s Transmit Screen when an instance of the transaction is sent to the Agentry Server to be processed.
- **Key Property:** This attribute specifies the property within the transaction to be treated as the key property. This is almost always the transaction property that targets the object’s key property and is set as such by default.

Transaction Type: Edit

An edit transaction is defined to allow the user to edit the property values of an existing object instance on the Agentry Client. This type of transaction should, at a minimum, include the key property of the object type and all property values that should be changed in the object.

It is highly recommended that users never be allowed to edit the key property of an object, as this can make it difficult, if not impossible, to update the enterprise system with any other changes for the object. Remember that the key property of an object is the value that uniquely identifies that object within both the mobile application and the enterprise system.

When an edit transaction is applied, the value of the properties are copied to the object properties they target. These new values will replace the previous values of the object properties. Object properties not modified by the transaction will not be changed. Once an object property is updated from an edit transaction, the previous value of that object property is lost and cannot be recovered.

When designing and developing an Edit transaction, the developer should consider whether or not the transaction definition should include merge functionality. Transaction merging is the behavior when an instance of an edit transaction is merged with an existing pending transaction targeting the same object instance on the Agentry Client. This functionality is controlled by the Edit transaction's merge attributes and is optional behavior.

Edit Transaction Attributes

Following are the attributes for an edit transaction:

- **Type:** This attribute specifies the type of transaction. For edit transactions this is set to Edit. This attribute cannot be changed once the transaction has been defined.
- **Object:** The Object attribute specifies the type of object the edit transaction targets on the Agentry Client. This may be set to any object type defined within the same module.
- **Name:** This is the identifier for the transaction definition. This value must be unique among all transaction definitions within the same module.
- **Display Name:** The display name is the value shown to users for the transaction on the Client. This is normally seen by the user in the Client's Transmit Screen when an instance of the transaction is sent to the Agentry Server to be processed.
- **Key Property:** This attribute specifies the property within the transaction to be treated as the key property. This is almost always the transaction property that targets the object's key property and is set as such by default.
- **Merge When:** This attribute specifies when the transaction should be merged. This can be set to either "Merge with adjacent transactions only" or "Merge with any transaction" to enable transaction merging on the Client. Adjacent transactions means the last transaction applied on the Client. Any transaction means the transaction will be merged with the first transaction found to meet the proper criteria for merging. This begins with the most recently applied transaction for the same object instance. The search continues back to the first applied transaction, or until a transaction is found that the edit transaction can be merged with.
- **Merge With:** This attribute specifies the type of transaction the edit transaction should be merged with. This can be set to "Same transaction type only" or "Similar transactions." The same transaction type is only another instance of the same edit transaction that targets the same object instance on the Client. A "Similar transaction type" also must target the

same object instance on the client, but may be an instance of any add or edit transaction that meets the merge criteria.

- **Timestamp:** The timestamp can be set to either "New Timestamp" or "Original Timestamp." This attribute specifies whether the timestamp from the original transaction is kept after the merge, or whether the timestamp from the new transaction instance is used.

Transaction Type: Delete

A delete transaction is defined to remove an object instance from the Agentry Client. When applied, this transaction will remove the object instance from the Client and may also remove any pending transactions for that object instance. A delete transaction should, at a minimum, contain the key property of the object type it targets.

When a delete transaction is applied, the object instance targeted by the transaction is removed from the Client. All data properties of the object instance, including any object collection properties, are removed.

When defining a delete transaction, the developer should ensure that the object should be allowed to be deleted. This is normally controlled by defining an enable rule for the action that will instantiate the delete transaction. The object and its data removed by the delete transaction cannot be recovered once the transaction has been applied.

Delete Transaction Attributes

Following are the attributes for a delete transaction:

- **Type:** This attribute specifies the type of transaction. For delete transactions this is set to "Delete". This attribute cannot be changed once the transaction has been defined.
- **Object:** The Object attribute specifies the type of object the delete transaction targets and will remove from the Agentry Client.
- **Name:** This is the identifier for the transaction definition. This value must be unique among all transaction definitions within the same module.
- **Display Name:** The display name is the value shown to users for the transaction on the Client. This is normally seen by the user in the Client's Transmit Screen when an instance of the transaction is sent to the Agentry Client to be processed.
- **Key Property:** This attribute specifies the property within the transaction to be treated as the key property. This is almost always the transaction property that targets the object's key property and is set as such by default.
- **Discard Pending Transactions:** This attribute specifies whether or not pending transactions for an object instance removed by the delete transaction should also be removed. If this attribute is set, pending transactions targeting the deleted object instance will be removed. If false, these pending transactions will remain on the Agentry Client until the next transmit.

Transaction Type: Complex Table Change

A complex table change transaction is defined to allow the user to add or edit a complex table record stored on the Agentry Client. This transaction type is still defined to target an object type. It should, at a minimum, contain a property for the key field of the table and the properties to target each field to be modified by the transaction. To allow for the addition of a new record, it should contain one property for each field in a table record.

When a complex table change transaction is applied, the transaction first looks for a record in the complex table whose key field value is equal to the value of the corresponding property in the transaction. If a match is found, the record is updated with the property values of the transaction. If no match is found, a new record is added to the complex table. The indexes of the complex table are then updated to match the new or modified record.

Complex Table Change Transaction Attributes

Following are the attributes for a complex table change transaction.

- **Type:** This attribute specifies the type of transaction. For complex table transactions this is set to “Complex Table Change.” This attribute cannot be changed once the transaction has been defined.
- **Object:** The Object attribute specifies the type of object the transaction targets. Though primarily intended to change a complex table record, this transaction type must still target an object.
- **Table:** The table attribute specifies the complex table the transaction targets and that will be changed when the transaction is applied.
- **Name:** This is the identifier for the transaction definition. This value must be unique among all transaction definitions within the same module.
- **Display Name:** The display name is the value shown to users for the transaction on the Client. This is normally seen by the user in the Client’s Transmit Screen when an instance of the transaction is sent to the Agentry Server to be processed.
- **Key Property:** This attribute specifies the property within the transaction to be treated as the key property. If no change is made to the targeted object, this attribute need not be set.

Transaction Type: Data Table Change

A data table change transaction is defined to allow the user to add or edit a data table record on the Agentry Client. This transaction is still defined to target an object type. It should, at a minimum, contain a property for the key and value fields of a data table record.

When a data table change transaction is applied, the transaction first looks for a record in the data table with the same key value as the corresponding key property in the transaction. If one is found, that record will be updated from the property for the value field. If there is no match on the key field, then a new record will be added to the data table using the values of the two properties for the key and value fields. Data table change transactions cannot delete a record from a data table.

Data Table Change Transaction Attributes

Following are the attributes for a data table change transaction.

- **Type:** This attribute specifies the type of transaction. For data table transactions this is set to “Data Table Change.” This attribute cannot be changed once the transaction has been defined.
- **Object:** The Object attribute specifies the type of object the transaction targets. Though primarily intended to change a data table record, this transaction type must still target an object.
- **Table:** The table attribute specifies the data table the transaction targets and that will be changed when the transaction is applied.
- **Name:** This is the identifier for the transaction definition. This value must be unique among all transaction definitions within the same module.
- **Display Name:** The display name is the value shown to users for the transaction on the Client. This is normally seen by the user in the Client’s Transmit Screen when an instance of the transaction is sent to the Agentry Server to be processed.
- **Key Property:** This attribute specifies the property within the transaction to be treated as the key property. If no change is made to the targeted object, this attribute might not be set.

Transaction Validation Rule

A transaction validation rule defines what rule definition will be used to validate the transaction’s data and how failed validation is handled on the Agentry Client. The rule referenced is called in a Boolean context and is expected to return true or false. False indicates failed validation, which may be treated as a warning or error. Messaging may be displayed to the user in relation to failed validation. An error requires the user to change the offending value(s) before proceeding. A warning displays an informational message giving the user the option to change the value(s), but does not require a change.

Not every transaction will have validation rules. Certain types of values do not need to be validated using a validation rule. Simple requirements such as the size of a string value or the minimum and maximum values of a numeric property can be enforced by the property itself. In other cases the information may not need to be validated. An example of this is some sort of note or description entry where the user is entering free form text.

Validation rules are used when more complex validation is required, such as when the valid value for a property is dependent on the value of a second property. Also, validation rules offer the flexibility to differentiate between a warning and an error. With a warning, the user is given the option of changing the value that violates the rule or leaving it as is. If treated as an error, the user must change the value before being allowed to proceed.

Validation rules are evaluated when the user clicks any navigation buttons in the wizard screen set displaying the transaction. If a validation rule references a property not yet displayed in the wizard, it will not fail validation. A false return by the rule is treated as a validation failure and the validation rule definition will then dictate the behavior of the Agentry Client.

Validation Rule Attributes

- **Rule:** This attribute specifies the rule definition within the same module to be used as a validation rule for the transaction. The rule is expected to return a Boolean value is evaluated in the context of the current transaction instance.
- **Type:** This attribute can be set to either "Warning" or "Error" and determines how a false return from the rule is treated by the validation rule. Warning means a failed validation does not require the user to change the value. The user will be displayed a message and given the option to change the value or keep it as set. An error type requires the user to change the value before proceeding.
- **Caption:** This is the text displayed in the title bar of the message for the validation rule.
- **Text:** This is the message displayed to the user when validation fails.
- **OK Label:** This is the text to label the OK button for the message screen on the Agentry Client.
- **Cancel Label:** This is the text to label the Cancel button for the message screen on the Agentry Client. This attribute is available only when the **Type** attribute is set to "Warning."

Transaction Validation Rule Properties

Rule properties associate one or more object properties with a transaction validation rule. Rule properties are used to set the cursor focus on the Client when a validation warning or error rule is triggered. The focus is set to the first property on the current screen set screen that is included in the Rule Properties list. If no match is found or if all of the listed properties are contained in a screen other than the current screen shown on the Client, no focus is set.

Validation rules and their associated rule properties are evaluated when the user clicks any navigation buttons in the wizard screen set displaying the transaction. If the rule returns false, the rule runs through all properties on the rule list until it finds one that matches a transaction property that is displayed on the Client screen. At that point, the 'next' or 'finish' process is stopped, the user remains on the same screen, and the cursor focus is set to the matching property. Any property controls that are hidden, disabled, or set to read-only will be ignored.

Validation rule properties are available on the following transaction types: Add, Edit, Complex Table Change, and Data Table Change.

Setting rule properties is optional. If no rule properties are set and a rule returns false, only the error or warning message associated with the rule is displayed on the Client.

Transaction Server Data State Steps

A transaction server data state step references a step definition within the same module to be run by the Agentry Server to check the back end system for data collisions during transaction processing. Server data state steps are the first steps run by the Server when a transaction is being processed. When a data state step's return is true, its defined data state is set for the transaction. This data state will then affect which server update steps for the transaction are run by the Server. A data state step may also define a response to be sent to the Client to perform some additional action in relation to the object instance targeted by the transaction.

The step executed by a data state step should be defined to either return data or not, based on some condition. The data returned by a data state step is unimportant in most cases. The exception to this is when the Client Response attribute is set to “Update Client Key Property.” In this scenario the Agentry Server will expect the step to return a value identified as the key property for the target object.

Other than in this situation, the Server only looks to see if data is returned by the step. By default, when data is returned by a data state step, the Server treats this as a true response and will set the defined data state for the transaction. No data will be treated as false. This logic may be inverted, with data return treated as false and no data treated as true, if the logic of the step being executed is more efficient or more practical to be written in this manner.

If multiple server data state steps return true for a single transaction instance, the defined data state for the last step with a true return will be the one set for the transaction. The server data state steps may be defined to halt further data state step processing for the transaction if one of them returns true. The order in which server data state steps are processed is defined in the list of these definitions displayed in the transaction definition’s properties view of the Editor.

The Client Response attribute allows for the definition of a response to be sent to the Client in relation to the object targeted by the transaction. This response will be sent after the transaction has been successfully processed by the Server. The defined Client Response for a data state may be overridden by a subsequent data state step, or by the transaction’s server update steps.

Server Data State Step Attributes

The following is a list of the attributes for a server data state step definition:

- **Step:** This attribute references the step definition within the same module to run as a server data state step for the transaction.
- **Data State:** This text value is the name of the data state the data state step will set if its return is true. This value can then be referenced by the server update steps for the same transaction.
- **Step is True if:** This attribute is set to define what is treated as true for the data state step. When a data state step is true, its defined data state is set for the transaction. Its available options depend on the type of step selected in the **Step** attribute:
  - *SQL Step*: For a SQL step, this attribute can be set to “1 or more rows are returned” or “0 rows are returned”. The former will treat data being returned as true and no data returned as false. The latter will treat data returned as false and no data returned as true.
  - *Java Step*: For a Java step, the options are “doSteplet returns True” and “doSteplet returns False”. The first will treat a true return from the `doSteplet()` method of the Java step as true. The second will treat a false response from the `doSteplet()` method as true.

  - *HTTP-XML Step*: For an HTTP-XML step, the available options for this attribute are "All response mappings succeed" and "A response mapping fails". The former will set true for the data state when the HTTP-XML step is able to map all of the responses, per its definition. The latter will treat one or more failed mappings as true.
- **If True:** This attribute specifies whether the remaining server data state steps for the transaction should be processed if the data state step returns true.
- **If False:** This attribute specifies whether the remaining server data state steps for the transaction should be processed if the data state step returns false.
- **Response to Client:** This attribute specifies what response is sent to the Agentry Client after the Update Step has been processed. The response defined here will only be sent if the data state step is run and returns true. The responses that may be sent are "Delete Client Object", "Replace Client Object", "Update Client Key Property", and "No Action Required". If "Update Client Key Property" is set, the step being run by the server data state step is expected to return a value identified as the transaction's target object's key property. This value will replace the current value of this property on the Client for that object instance.

Transaction Server Update Step

A transaction server update step references a step definition within the same module that is run during transmit to update the back end system with the data captured by the transaction. This step has access to all of the properties of the transaction using the SDML or mechanisms available using the Agentry Java API. The value of these properties can be used by the steps to update the back end system. An update step can be defined to run or not run based on a data state being set for the transaction. An update step can also define a response to be sent to the Client to perform some additional action in relation to the object instance targeted by the transaction.

Using the data state functionality, update steps may be defined for a single transaction that process the data captured in the transaction normally, and other steps that run only when data states are set to provide data collision handling. Each server update step can contain its own list of selected data states, that is, the data states it is aware of. It can then be defined to run or not when one of its selected data states is set.

Server update steps can send a client response after they have been processed by the Agentry Server. This response will only be sent if the step that defines it is run. Only one response is sent for a transaction. There are different responses possible, and which one is ultimately sent to the Client is based on the type of response.

Server Update Step Attributes

Following is a list of the attributes for a server update step definition:

- **Step**: This attribute references the step definition within the same module to be run by the Agentry Server as a server update step for the transaction.
- **Run for which States:** This attribute defines when the step is run in relation to the transaction's data states. This can be "All Data States", "Data States except selected",

"Only selected data states", and "Do Not run Step". This last option is normally only set for testing purposes, as the step will never be run if this option is selected. When set to one of the two data state options, a second tab is available in the Properties view of the Agentry Editor. This second tab lists all selected data states for the update step and allows for additional data states to be added.

- **Response to Client:** This attribute specifies what response is sent to the Agentry Client after the Update Step has been processed. The response defined here will only be sent if the update step is run. The responses that may be sent are "Delete Client Object", "Replace Client Object", "Update Client Key Property", and "No Action Required". If "Update Client Key Property" is set, the step being run by the server update step is expected to return a value identified as the transaction's target object's key property. This value will replace the current value of this property on the Client for that object instance.

Transaction Error Handling Steps

A transaction error handling step references a step definition that is run during transmit if an error occurs while the Server is processing the transaction. This includes errors returned by the data state or update steps. Error handling steps are run only when transaction failure handling is enabled, via a configuration option of the Agentry Server. An error handling step can respond to the Client to indicate the proper action to take in relation to the error that has occurred.

Error handling steps can perform multiple tasks to resolve such an issue. These include:

- Any post-error processing that may be necessary
- Setting the error fatality level
- Returning messaging to the Agentry Client for display to the user

One of the key components to transaction error handling steps is the error fatality. This term refers to the severity of the error and the proper way in which the transaction should be handled as a result of the error. This can include retrying the transaction, possibly after a change is made to it by the user, or removing the transaction from the Agentry Client and storing its data to the failed transactions queue on the Agentry Server.

Error handling steps may not need to be defined as a apart of the transaction failure handling. The Agentry Server contains configuration options to set default behaviors, including the fatality level of an error. Error handling steps are normally defined to override these defaults where necessary.

Error Handling Step Attributes

- **Step**: This is the step definition within the module to be run as an error handling step for the transaction. The step referenced here should be defined to return data in the event of an error, or a specific type of error.
- **Error Type:** This attribute determines the behavior of the application when the error handling step returns true, indicating the error that occurred should be handled by the step. The options for this attribute are:

    - *Fatal with Message* - The transmit will be aborted automatically and a message will be displayed to the user. The transaction will be removed from the Client and the data for it stored in the failed transactions que on the Server.
    - *Fatal without Message* - The transmit will be aborted automatically and no message will be displayed to the user specific to the transaction. The transaction will be removed from the Client and the data for it stored in the failed transactions que on the Server.
    - *No Change* - This selection will not change the error fatality for the transaction. Either another error handling step for the transaction will handle this, or the default fatality based on the error information returned by the back end system will remain. This is normally set for steps that either create messaging displayed to the user, or that perform other actions against the back end system to handle the error.
    - *Retry with Change* - The user will be able to choose to abort the transmit and to change the data for the transaction. This requires transaction merging be enabled, as a new transaction will be instantiated by the user and it will then merge with the pending transaction as a result of an error. This will be an option for the user and, should the user choose not to retry, the transmit will continue. The transaction will be removed from the Client and saved to the failed transactions queue on the Server.
    - *Retry without Change* - The user will be able to retry the transaction without editing the data it contains.
- **Step is true if:** This attribute controls whether data returned by the step is treated as a true or false return. When this attribute is true and the step returns data, this is treated as a true response.
- **If True:** This attribute defines whether or not the remaining error handling steps for the transaction should be run if the current error step returns true.
- **If False:** This attribute defines whether or not the remaining error handling steps for the transaction should be run if the current error step returns false.
- **Notification:** This Boolean attribute controls the external notification on the client device. If this attribute is true, a true result for the error handling step will result in the LED on the client device being activated and the transmit dialog flashing.
- **Sound:** This attribute defines whether or not the system default sound on the client device should be played when the error step returns true. It also controls the number of times to repeat the sound.
- **Interval:** If the **Sound** attribute is set to play the system sound two or more times, the interval attribute can be set to the number of seconds in between each time the sound is played.

Fetch

A fetch defines how the Agentry Server synchronizes data for a target object collection. This object collection must be a top-level collection within the module. A fetch is made up of steps that retrieve the data for the collection from the back end system. These steps are grouped into three categories within the Fetch definition: Client Exchange Steps, Server Exchange Steps, and Removal Steps. A fetch may also include properties to store data captured from the user and validation rules for those property values.

A fetch may be a main or non-main fetch. A main fetch is processed during every transmit between the Agentry Client and Server. A given module may contain multiple main fetches. The order in which multiple main fetches, either within the same module or within multiple main fetches, are processed is undefined and should therefore not be a factor in the synchronization logic.

A non-main fetch will only be executed when an action step of type transmit explicitly defines such a fetch to be processed. Non-main fetches are normally defined to provide the search functionality to end users.

The basic structure of a fetch definition is intended to support the exchange data model of synchronization. This model is intended to allow for the synchronization of data in a more efficient manner, where only data changes on the back end system as compared to the current data on a given client are retrieved. Any data that has not been changed as compared to the client’s data is not retrieved.

A fetch definition can be defined to retrieve new object instances to be added to a client application, replace existing objects on that client, a remove any objects the client should no longer store locally. The read steps of the object type targeted by the fetch are run after the fetch has been processed and may also retrieve objects for the client to either add them or replace existing instances.

Fetch Child Definitions

- **Property:** A fetch property defines data to be captured on the Agentry Client for use during fetch processing by the Agentry Server.
- **Validation Rule:** A fetch validation rule defines what rule definition will be used to validate the fetch’s data and how failed validation is handled on the Agentry Client.
- **Client Exchange Step:** A fetch client exchange step defines how information about the target collection is processed by the Agentry Server.
- **Server Exchange Step:** A fetch server exchange step defines how information about the back end system’s data is processed.
- **Removal Step:** A fetch removal step is defined to determine which objects should be removed from the collection targeted by the parent fetch.

Fetch Attributes

- **Collection:** This attribute references the object collection property within the same module and that is a direct child of the module main object for which the fetch will synchronize data. Steps executed by the fetch’s child step usage definition will be processed by the Agentry Server in the context of this collection.
- **Name:** This attribute contains the name that identifies the fetch. This value must be unique among all fetch definitions within the same module.

- **Display Name:** This attribute contains the value that identifies the fetch on the client. This is displayed during synchronization in the Client's Transmit Screen when the fetch is processed by the Server.
- **Clear Collection:** This attribute specifies whether or not the object instances stored in the targeted collection should be removed from the Client prior to processing the fetch during synchronization. This attribute is normally only left set on when the fetch is either not using the exchange data model for synchronization, or when it is a non-main fetch performing search functionality and the previous search results should be removed from the client before performing a new search.
- **Main Fetch:** This attribute specifies whether the fetch is a main fetch. When checked, the fetch will be processed during every transmit between the Client and Server. When unchecked, the fetch will only be run when an action step of type Transmit explicitly lists the fetch to be processed and that action step is the one that initiates the transmit.

Fetch Validation Rule

A fetch validation rule defines what rule definition will be used to validate the fetch's data and how failed validation is handled on the Agentry Client. The rule referenced is called in a Boolean context and is expected to return true or false. False indicates failed validation, which may be treated as a warning or error. Messaging may be displayed to the user in relation to failed validation. An error requires the user to change the offending value(s) before proceeding. A warning displays an informational message giving the user the option to change the value(s), but does not require a change.

Not every fetch will have validation rules. Certain types of values do not need to be validated using a validation rule. Simple requirements such as the size of a string value or the minimum and maximum values of a numeric property can be enforced by the property itself. In other cases the data may simply not need to be validated.

Validation rules are used when more complex validation is required, such as when the valid value for a property is dependent on the value of a second property. Also, validation rules offer the flexibility to differentiate between a warning and an error. With a warning, the user is given the option of changing the value that violates the rule or leaving it as is. If treated as an error, the user must change the value before being allowed to proceed.

Validation rules are evaluated when the user clicks any navigation buttons in the wizard screen set displaying the fetch. If a validation rule references a property not yet displayed in the wizard, it will not fail validation. A false return by the rule is treated as a validation failure and the validation rule definition will then dictate the behavior of the Client.

Validation Rule Attributes

- **Rule:** This attribute specifies the rule definition within the same module to be used as a validation rule for the fetch. The rule is expected to return a Boolean value in the context of the current fetch instance.
- **Type:** This attribute can be set to either "Warning" or "Error" and determines how a false return from the rule is treated by the validation rule. Warning means a failed validation

does not require the user to change the value. The user will be displayed a message and given the option to change the value or keep it as set. An error type requires the user to change the value before proceeding.

- **Caption:** This is the text displayed in the title bar of the message for the validation rule.
- **Text:** This is the message displayed to the user when validation fails.
- **OK Label:** This is the text to label the OK button for the message screen on the client.
- **Cancel Label:** This is the text to label the Cancel button for the message screen on the client. This attribute is available only when the **Type** attribute is set to “Warning”.

Fetch Validation Rule Properties

Rule properties associate one or more object properties with a fetch validation rule. Rule properties are used to set the cursor focus on the Client when a fetch warning or error rule is triggered. The focus is set to the first property on the current screen set screen that is included in the Rule Properties list. If no match is found or if all of the listed properties are contained in a screen other than the current screen shown on the Client, no focus is set.

Validation rules and their associated rule properties are evaluated when the user clicks any navigation buttons in the wizard screen set displaying the transaction. If the rule returns false, the rule runs through all properties on the rule list until it finds one that matches a fetch property that is displayed on the Client screen. At that point, the ‘next’ or ‘finish’ process is stopped, the user remains on the same screen, and the cursor focus is set to the matching property. Any property controls that are hidden, disabled, or set to read-only will be ignored.

Validation rule properties are available on the following fetch types: Add, Edit, Complex Table Change, and Data Table Change.

Setting rule properties is optional. If no rule properties are set and a rule returns false, only the error or warning message associated with the rule is displayed on the Client.

Fetch Client Exchange Step

A fetch client exchange step defines how information about the target collection is processed by the Agentry Server. This definition references a step definition within the same module. This step has access to information about the target collection, as well as to any data captured in fetch properties. A client exchange step can be defined to execute once or iteratively, and can return data for an object collection. A fetch can contain multiple client exchange step definitions, which are processed by the Server in a defined order.

Though a client exchange step can return data to create and populate object instances, its intended purpose is to provide information about the current objects stored in the collection property targeted by the parent fetch (target collection) definition. A Client exchange step has access to the key property and last update value for each object instance in the target collection. This information is provided in support of the exchange data model. The intent is that the client exchange steps update this information to an exchange data object in the back end for later comparison to determine which data may need to be retrieved to update the Client.

Client Exchange Step Attributes

- **Step:** This attribute references the step definition within the same module to run as a client exchange step for the parent fetch.
- **Run:** This attribute specifies how to run the client exchange step during a single transmit. This may be set to one of the following values:
  - *Run one Time:* This setting will run the client exchange step a single time for the fetch processing. This setting assumes the step needs to be executed only once to return the data for all object instances to be added or replaced during synchronization, or the step being executed is not returning data but rather is being run in support of synchronization.
  - *Run Once per Object:* This setting will execute the client exchange step once for each object instance in the collection that is being synchronized. This includes both those object instances sent by the Client to the Server, as well as any that may have been added by previous fetch steps.
- **Read Into:** This attribute specifies the child or descendent object collection property of the target collection into which the data returned by the step should be read. This attribute has a default setting of “None”. This default means the data will not be read into a child collection but will instead be used to create object instances of the target collection. Other valid options for this attribute are any child collection properties of the target collection, or any descendent collections (e.g. collections within collections).

Fetch Server Exchange Step

A fetch server exchange step defines how information about the back end system’s data is processed. This definition references a step definition within the same module. This step has access to information about the target collection, as well as to any data captured in fetch properties. A server exchange step can be defined to execute once or iteratively, and can return data for an object collection.

The server exchange step definition is intended to perform one of two tasks within the exchange data model. First, it should compare information provided by the client exchange steps concerning which object instances the Client currently has and when they were retrieved to information in the back end system about when that same data was last modified or added. Second, it can then retrieve the data needed by the Client based on the differences found during this comparison. These tasks are normally accomplished by separate server exchange steps. Alternately or in addition to these definitions, the object read steps defined in the object type targeted by the fetch may retrieve data for the object instances.

Server Exchange Step Attributes

- **Step:** This attribute references the step definition within the same module to run as a server exchange step for the parent fetch.
- **Run:** This attribute specifies how to run the server exchange step during a single transmit. This may be set to one of the following values:

    - *Run one Time:* This setting will run the server exchange step a single time for the fetch processing. This setting assumes the step needs to be executed only once to return the data for all object instances to be added or replaced during synchronization, or the step being executed is not returning data but rather is being run in support of synchronization.
    - *Run Once per Object:* This setting will execute the server exchange step once for each object instance in the collection that is being synchronized. This includes both those object instances sent by the Client to the Server, as well as any that may have been added by previous fetch steps.
- **Read Into:** This attribute specifies the child or descendent object collection property of the target collection into which the data returned by the step should be read. This attribute has a default setting of “None”. This default means the data will not be read into a child collection but will instead be used to create object instances of the target collection. Other valid options for this attribute are any child collection properties of the target collection, or any descendent collections (e.g. collections within collections).

Fetch Removal Step

A fetch removal step is defined to determine which objects should be removed from the collection targeted by the parent fetch. A removal step references a step definition within the same module. This step has access to information about the target collection, as well as to any data captured in fetch properties. The step referenced by a removal step definition is expected to return the key property of any object(s) that should be deleted from the target collection on the Agentry Client.

Removal Step Attributes

- **Step:** This attribute references the step definition within the same module to run as a removal step for the parent fetch.
- **Run:** This attribute specifies how to run the removal step during a single transmit. This may be set to one of the following values:
    - *Run one Time:* This setting will run the removal step a single time for the fetch processing. This setting assumes the step need be executed only once to return the data for all object instances to be removed during synchronization, or the step being executed is not returning data but rather is being run in support of synchronization.
    - *Run Once per Object:* This setting will execute the removal step once for each object instance in the collection that is being synchronized. This includes both those object instances sent by the Client to the Server, as well as any that may have been added by previous fetch steps.
- **Read Into:** This attribute has no effect on a fetch removal step and will be deprecated in a future release.

Transaction and Fetch Properties

A transaction property defines a value to be captured by a transaction. Definable behaviors include the initial value for the property, the object property or table record field it targets, as well as data-related behaviors. These include minimum and maximum values, a special value, and similar settings. These last behaviors will vary depending on the data type of the property.

A fetch property defines data to be captured on the Agentry Client for use during fetch processing by the Agentry Server. A fetch that contains properties is normally displayed in a screen set to allow the user to enter the desired values. The steps of the fetch then have access to these property values for use during synchronization. The fetch properties themselves define the data types of the values, and the initialization values when the fetch is instantiated.

Both transaction and fetch properties contain attributes related to initialization. These attributes are a part of all transaction and fetch property definitions regardless of the property data type. These attributes are in addition to the data type specific attributes.

For both a fetch and a transaction property, the purpose is to capture data on the Client. How this data is used depends on the property's parent. A transaction property's value will be copied to the object property it targets when the transaction is applied. This value will then also be available to the steps used by the transaction during synchronization and, depending on the defined processing, will likely be updated to the back end system.

A fetch property will be stored with the fetch and sent to the Agentry Server during synchronization. This will make the value available to all steps run by the fetch. However, the fetch property value will not affect the object property, as fetch properties do not modify object instances on the Client.

Transaction and Fetch Property Attributes

The following list of attributes are specific to properties defined for a transaction or fetch. These attributes are common to all properties regardless of data type:

- **Object Property:** This attribute specifies the object property targeted by the transaction or fetch property. This value may be used for initialization. For a transaction, this is also the object property the transaction property will set when the transaction is applied.
- **Initial Value:** This attribute specifies the data source to initialize the property. This may be the object property targeted by the transaction or fetch property, the property of a different object not targeted by the fetch or transaction, a constant value, or via a rule. When a rule is used, the rule may be evaluated before or after data entry.
- **Constant:** This attribute is enabled only when **Initial Value** is set to “Constant”. The **Constant** attribute then contains the constant value to which the property will be initialized whenever the parent transaction or fetch is instantiated on the Client. This may be left blank for many property data types to initialize the property to null.
- **Rule:** This attribute is enabled only when the **Initial Value** attribute is set to either “Rule - before data entry” or “Rule - after data entry”. It contains a reference to the rule definition to be evaluated to initialize the property.
- **Other Property:** This attribute is enabled only when the **Initial Value** attribute is set to “From a different object property”. **Other Property** then contains the target path to the object property whose value will be used to initialize the property.

Property Data Types

The property data type definition can be a child to an object, transaction, or fetch definition. A property is defined to be a certain data type when it is created. This data type then specifies the type of data and its behavior within the property. The data types range from primitive types common to most or all development platforms, to more robust types that in other languages would be created by developers as classes, structures, or objects depending on the tool or language in use.

Following a brief description of each property data type available in Agentry:

- **Boolean:** The Boolean property data type stores a true or false value.
- **Collection:** The collection property data type is defined to store multiple object instances of the same type as a property of a parent object, transaction, or fetch.
- **Complex Table Selection:** The complex table selection property type is used to store a selection made by the user from a complex table.
- **Data Table Selection:** The data table selection property type is used to store a selection made from a data table.
- **Date:** The date property type is used to store a calendar date value.
- **Date and Time:** The date and time property type stores a value consisting of a calendar date and time of day.
- **Decimal Number:** The decimal number property data type stores numeric value with a fractional component.
- **Duration:** The duration property data type is used to store a duration of time.
- **External Data:** An external data property stores a reference to a file stored on the client device’s file system and that is external to the production data of the application.
- **Identifier:** The identifier property data type stores a non-negative integer value that is a unique identifier for an object.
- **Image:** The image property stores a still picture or other image captured on the client device from either the device’s camera or selected from the file system.
- **Integral Number:** An integral number property stores whole numbers.
- **Location:** A location property stores a location value returned by a GPS unit that includes the latitude, longitude, dilution, and number of satellites.
- **Object:** The object property data type stores an object instance as a property of a parent definition.
- **Signature:** The signature property type stores a signature entered by a user on the Agentry Client.
- **String:** The string property data type stores any character values as a single string.
- **Time:** The time property data type stores a time of day value.

Boolean Property Type

The Boolean property data type stores a true or false value. When a Boolean property value is set, a null value is treated as false and any other value is treated as true.

The attributes for a Boolean property include the true and false value. These values define what will be displayed when the property contains a true or false value.

Boolean Property Attributes

Note: This property type does not have Special Value attributes.

- **Name:** This is the unique identifier for the property definition. This value must be unique among all properties within the same parent definition.
- **Display Name:** This attribute sets the default display name to label or otherwise identify the property value on the Client. This will also be the default label for any screen control that displays this property.
- **True Value:** This attribute contains the value to display when the Boolean property is set to true.
- **False Value:** This attribute contains the value to display when the Boolean property is set to false.

Collection Property Type

The collection property data type is defined to store multiple object instances of the same type as a property of a parent object, transaction, or fetch. The object type used in a collection property must have a defined key property to uniquely identify each object instance within the collection property. The default initialization for a collection property is an empty collection.

Each object instance within a collection is considered a child instance to the parent definition of the collection property. In objects, collection properties are commonly used to store object instances within a module to provide a data structure within the module representing the relationship between the different business entities for the module. Collection properties defined in the module main object are commonly referred to as “top-level collections”. Collection properties defined within an object other than the main object are referred to as nested collections.

The collection property type may also store other data types. However, in practice there is limited use for this type of definition. A collection defined to store another collection is not valid.

Collection Property Attributes

Note: This property type does not have Special Value attributes.

- **Name:** This is the unique identifier for the property definition. This value must be unique among all properties within the same parent definition.
- **Display Name:** This attribute sets the default display name to label or otherwise identify the property value on the Client. This will also be the default label for any screen control that displays this property.

- **Property Type:** This attribute specifies the type of property the collection will contain. The default and most common setting for this attribute is Object. There are limited use cases for collections storing instances of any other type of data.
- **Object:** This attribute is available when **Property Type** is set to “Object”. It lists all object definitions within the module and the selection made specifies the type object instances the collection property will contain. The object type to be stored in the collection property must have been defined previously and must have its **Key Property** attribute set prior to selecting it in the **Object** attribute field of the collection property definition.

Complex Table Selection Property Type

The complex table selection property type is used to store a selection made by the user from a complex table. The value stored in a complex table selection property is the key field of the selected record within the complex table. The data type of this value will be a string, integral number, or decimal number, based on the data type of the key field.

The complex table selection contains a single attribute specific to the data type named complex table. The setting of this attribute specifies the complex table definition that is the source for the property.

The value contained within a complex table selection property requires a brief explanation of complex tables. Complex tables are made up of records. The records are made up of multiple fields. Within the complex table definition, indexes are defined on the fields to allow users to search the table. Each complex table is required to contain a unique index, which is defined for the field that contains the unique value for each record. The complex table selection property will contain the value of this field for the record selected by the user.

Complex Table Selection Property Attributes

- **Name:** This is the unique identifier for the property definition. This value must be unique among all properties within the same parent definition.
- **Display Name:** This attribute sets the default display name to label or otherwise identify the property value on the Agentry Client. This will also be the default label for any screen control that displays this property.
- **Complex Table:** This attribute specifies the complex table definition within the application that the property will use. Only selections from the complex table specified here can be stored in the property. The complex table definition must exist and contain at least one field and the primary index before it may be selected for the **Complex Table** attribute of the property definition.

Data Table Selection Property Type

The data table selection property type is used to store a selection made from a data table. The value stored in a data table selection property is the code field of the selected data table record. This value will always be a string data type.

The data table selection property type includes display options for its value within the definition. Whenever this property type is displayed, the entire data table record may be displayed for its code. Also, only the code field or the value field may be displayed. Which is shown on the client is defined within the data table selection property definition.

It is important to note that it is not a requirement that a value selected from a data table be stored in a data table selection property. It is only one of the options available, and other property data types may be used for this purpose.

Data Table Selection Property Attributes

- **Name:** This is the unique identifier for the property definition. This value must be unique among all properties within the same parent definition.
- **Display Name:** This attribute sets the default display name to label or otherwise identify the property value on the Client. This will also be the default label for any screen control that displays this property.
- **Data Table:** This attribute specifies the data table from which the selection will be made for the value stored in the property. Only selections for the data table definition chosen here can be stored in the property. The data table definition selected here must exist within the application prior to defining this property type.
- **Display Type:** This attribute specifies how the selected data table record stored in the property will be displayed. The options are to display the code field, value field, or code and value field of the selected data table record; or to specify format text.
- **Format Text:** This attribute is available only when Display Type is set to “Format Text.” It specifies the format string to display the selected data table record stored in the property. This attribute may contain any printable characters plus the format strings `%code` and `%value`.

Date Property Type

The date property type is used to store a calendar date value. This value is stored internally as the number of days before or after the Agentry epoch date of January 1st, 2001. Negative values reflect dates prior to epoch. A date property is displayed on the Agentry Client in the format MM/DD/YYYY by default.

The date property may also be unset or invalid. In this case the year portion of the date property is set to zero (0000). This condition may be checked to determine if the date property has been set.

Date Property Type Attributes

- **Name:** This is the unique identifier for the property definition. This value must be unique among all properties within the same parent definition.
- **Display Name:** This attribute sets the default display name to label or otherwise identify the property value on the Agentry Client. This will also be the default label for any screen control that displays this property.

- **Blank:** This attribute specifies whether or not a blank value is displayed for a date property when it has not been set (year is zero).

Date And Time Property Type

The date and time property type stores a value consisting of a calendar date and time of day. This value is stored internally as the number of seconds before or after the Agentry epoch date of January 1st, 2001 12:00:00 am. Negative values reflect dates prior to epoch. A date and time property is displayed on the Agentry Client in the format MM/DD/YYYY HH:MM:SS am/pm by default.

A date and time property may contain an unset or invalid value. This is indicated by the year portion of the value, which is set to the year zero (0000). This condition may be checked to determine if the date and time property is invalid.

Date And Time Property Attributes

- **Name:** This is the unique identifier for the property definition. This value must be unique among all properties within the same parent definition.
- **Display Name:** This attribute sets the default display name to label or otherwise identify the property value on the Agentry Client. This will also be the default label for any screen control that displays this property.
- **Blank:** This attribute specifies whether or not to display a blank field when the date and time property does not contain a valid date and time (year is 0000).
- **Display Order:** This attribute specifies the order in which to display the date and time components of the property value. This may be set to either “Date - Time” or “Time - Date”.
- **Time Zone Adjust:** This attribute specifies whether or not to adjust the date and time value of the property during synchronization based on differences in time zones. For object properties the value retrieved from the back end system may be adjusted from the back end systems local or standard time, or from universal time, to the client device’s time zone. For transaction properties the date and time value can be adjusted from the client device’s time zone to the local or standard time of the system connection or to universal time. The default for this attribute is “Do not adjust”, which will not modify the date and time value during synchronization.

Decimal Number Property Type

The decimal number property data type is used to store a numeric value with a fractional component. The definable behaviors of a decimal number property include standard or NIST rounding, precision, and significant digit math options.

Values stored in a decimal property can contain values with a precision of up to 20 places past the decimal point. The values may be positive or negative.

Decimal Number Property Attributes

- **Name:** This is the unique identifier for the property definition. This value must be unique among all properties within the same parent definition.
- **Display Name:** This attribute sets the default display name to label or otherwise identify the property value on the Agentry Client. This will also be the default label for any screen control that displays this property.
- **Minimum Value:** This is the smallest value that may be contained within the property. This attribute and the **Precision** attribute are interdependent. You cannot specify a minimum value with more decimal places than is specified in the **Precision** attribute. To force the value to be positive, set the this attribute to 0. The minimum value specified here can be no greater than the defined **Maximum Value**.
- **Maximum Value:** This is the maximum value that the decimal property can contain. This is dependant on the **Precision** attribute. You cannot specify a maximum value with more decimal places than is specified in the **Precision** attribute. The value defined here can be no less than the defined **Minimum Value**.
- **Precision:** This attribute specifies the maximum number of places past the decimal point. A negative precision indicates places before the decimal, with any values past this point padded with zeroes. A precision of 0 specified whole numbers only, though consider using an integral number property for this purpose.
- **Blank** - This is a Boolean attribute that specifies whether to display a blank for the property when it has a value of 0.
- **Math:** This is a Boolean attribute that specifies whether or not to use significant figure math in any calculations that use the property value.
- **Rounding:** This attribute specifies the rounding method to use when this value is rounded. This may occur within rule definitions (ROUND function term) or when calculations involving this property are performed. The resulting value for the property will be rounded to the defined precision, as well as based on the significant digits operations. The methods for rounding are **Nearest** or **NIST**. Nearest is the typical rounding method in which the digit immediately after the digit to be rounded determines value of that rounded digit. Values below 5 leave the digit unchanged. Values 5 or above increment the rounded digit by 1. The NIST rounding method rounds values according to the rules set forth by the National Institute of Standards and Technology, specifically as they relate to calibrations measurements.

Duration Property Type

The duration property data type stores a duration of time. The value of a duration property is stored in seconds and may be positive or negative. It is possible to convert the value to other time units, including hours, minutes, or milliseconds when referenced in a step definition. This behavior is controlled by the definition of the duration property.

This data type does not store fractional seconds. During downstream synchronization, if the back end units for this property include precision smaller than whole seconds, the fractional second portion of the value will be truncated when assigned to the property. The logic of the synchronization step should round the value prior to returning it to the Server if this is not the desired behavior. If it is necessary to keep the fractional portion of the duration value during synchronization, a decimal number property should be used.

Duration Property Attributes

- **Name:** This is the unique identifier for the property definition. This value must be unique among all properties within the same parent definition.
- **Display Name:** This attribute sets the default display name to label or otherwise identify the property value on the Agentry Client. This will also be the default label for any screen control that displays this property.
- **Minimum Value:** This attribute specifies the minimum duration value the property will accept. This is set in hours, minutes and seconds and is converted to total number of seconds within the property. This value can be no greater than the defined **Maximum Value**.
- **Maximum Value:** This attribute specifies the maximum duration value the property will accept. This is set in hours, minutes and seconds and is converted to total number of seconds within the property. This value can be no less than the defined **Minimum Value**.
- **Display Format:** This attribute specifies the format in which to display the duration value. The options are Fractional Hours, Hours:Minutes:Seconds (“H:M:S”), Minutes:Seconds (“M:S”), or Hours:Minutes (“H:M”).
- **Back End Units:** This attribute specifies the units in which the duration value is stored in the back end system. For object properties the value returned from the back end system will be converted from the unit selected here to seconds. For transaction and fetch properties the value will be converted from seconds to the units specified here.

External Data Property Type

An external data property is used to reference a file stored on the client device’s file system. This file is external to the application’s production data. The file data itself is not stored with the production data. This property type is normally used in conjunction with the file transfer functionality. The default display value of an external data property is the full path and file name of the referenced file.

For object properties the attributes of this property type related to the location for the file specify where the file will be stored on the client device when retrieved from the back end system. For transaction properties these same attributes specify the default location from which the user should make a selection. The file dialog opened in this case does allow the user to navigate the file system to select the desired file. Two separate paths can be defined for the external data property, one for client devices running the Windows PC group of operating systems, and a second for client devices running supported versions of the Mobile Windows OS’s.

The attributes of this data type also allow for designating whether the file should be read-only on the client device, the file extension for the file, and whether or not to delete the file when the parent object to the property is deleted.

The recommended use for this property type is define an object that represents the document and includes this property, as well as other information about the file. The external data property itself will reference the location of the file and can return the file's full path and name, just the file name, just the file path, just the file extension, as well as metadata about the file such as its last modified date and time and whether or not it has been modified since it was downloaded to the client device. Many of these values are exposed via rules and/or format strings. A separate property of a data type other than external data must exist and be referenced by the external data property that contains the name the file will be given when saved on the client device. This value must be set during synchronization prior to transferring the file itself.

External Data Property Attributes

Note: This property type does not have Special Value attributes.

General Attributes

- **Name:** This is the unique identifier for the property definition. This value must be unique among all properties within the same parent definition.
- **Display Name:** This attribute sets the default display name to label or otherwise identify the property value on the Agentry Client. This will also be the default label for any screen control that displays this property.

Client File Attributes

- **File Name:** This attribute references a property within the same parent definition as the external data property. The referenced property's value will be used as the file name for the file referenced by the external data property when it is downloaded from the back end during object synchronization. In a transaction, the File Name property will store the name of the file as saved on the client device.
- **File Extension:** This attribute is optional and can contain the file extension for the file referenced by the external data property. For objects this extension will be appended to the file when it is downloaded from the back end and saved on the client device. For transactions this extension will be used to filter the options displayed to the user in the file dialog when selecting the file. Only files with the same extension will be displayed. If this attribute is not set, files saved to the Agentry Client during object synchronization will keep the same file extension as provided by the back end. For transactions, all files will be listed in the File Dialog to the user, regardless of file extension.
- **When Object is Deleted:** This attribute specifies what to do with the file referenced by the external data property when the parent object of the property is deleted from the Agentry Client. The options are to always delete the file, never delete the file, or only delete the file when retrieved from the back end by the mobile client application. This last option will exclude files attached locally on the Agentry Client via a transaction. This option is

unaffected by the Read Only attribute, meaning if this setting results in the file being deleted, it will be removed regardless of whether or not it is read-only.

- **Read Only:** This attribute specifies whether or not the file's read-only switch will be true. When set, this will prevent the user from modifying the file but will not prevent the Agentry Client from deleting or otherwise accessing the file.
- **Use Most Recent Location:** This attribute specifies whether, when selecting a file on the Agentry Client to be referenced bu the external data property, the file dialog displayed should be opened to the most recently selected folder, or to the default folder regardless of the previous selection made.

Filter

- **File Filter:** This attribute specifies the file type that may be selected or referenced by the external data property.
- **File Filter Description:** This attribute allows for the specification of a file description to be associated with the file extension listed in the File Filter attribute.
- **Restricted Files:** This attribute allows for the specification of file names or file extensions that may not be selected. Multiple files or file types can be listed here separated by semi-colons.

Windows 9.x/NT/2000/XP

- **Base Path:** This attribute specifies the base path to which the file will be saved (objects) or the default location the user will be displayed in the file dialog to select a file (transactions). This attribute is for Windows PC operating system builds for PC's, laptops, and tablets. This may or may not be the entire path for the application, dependent on the Relative Path attribute. Options for this attribute include:
  - **Absolute Path:** This selection will result in the value of the Relative Path attribute being used and is assumed to contain the full path, including drive letter, for the files location.
  - **Application Data:** This selection will set the file's location to be the path configured in Windows to be the location for application data.
  - **My Documents:** This selection will set the file's location to be the path configured in Windows to be the user's My Documents folder.
  - **My Pictures:** This selection will set the file's location to be the path configured in Windows to be the user's My Pictures folder.
  - **Program Files:** This selection will set the file's location to be the path configured in Windows to be the Program Files folder.
  - **Windows Temporary Directory:** This selection will set the file's location to be the path configured in Windows to be the Windows TEMP folder.
- **Relative Path:** The value of this attribute will be appended to the path resulting from the Base Path attribute setting. If Base Path is set to Absolute Path, the value of Relative Path will be used as the full path for the file's location.

Windows CE (Mobile Windows versions)

- **Use Path:** This attribute, when checked, will use the same path as defined in the Windows 9.x/NT/2000/XP set of attributes. This will disable the Base Path and Relative Path attributes for mobile devices.
- **Base Path:** This attribute specifies the base path to which the file will be saved (objects) or the default location the user will be displayed in the file dialog to select a file (transactions). This attribute is for Mobile Windows operating system builds. This may or may not be the entire path for the application, dependent on the Relative Path attribute. Options for this attribute include:
  - **Absolute Path:** This selection will result in the value of the Relative Path attribute being used and is assumed to contain the full path, including drive letter, for the files location.
  - **My Documents:** This selection will set the file’s location to be the path configured in Windows to be the user’s My Documents folder.
  - **Program Files:** This selection will set the file’s location to be the path configured in Windows to be the Program Files folder.
  - **Windows Temporary Directory:** This selection will set the file’s location to be the path configured in Windows to be the Windows TEMP folder.
- **Relative Path:** The value of this attribute will be appended to the path resulting from the Base Path attribute setting. If Base Path is set to Absolute Path, the value of Relative Path will be used as the full path for the file’s location.

Identifier Property Type

The identifier property data type is used to store a non-negative integer value that is a unique identifier for an object. The intent of this data type is to be used as a key property for an object. This is not a requirement and a property of a different data type may be used as an object key property.

Identifier Property Attributes

- **Name:** This is the unique identifier for the property definition. This value must be unique among all properties within the same parent definition.
- **Display Name:** This attribute sets the default display name to label or otherwise identify the property value on the Client. This will also be the default label for any screen control that displays this property.
- **Blank:** This attribute specifies whether to display a blank value or 0 when the identifier property has not been set.

Image

The image property stores a still picture or other image captured on the client device from either the device’s camera or selected from the file system. This property type is provided as a part of the overall image capture functionality that may be implemented in the mobile application. This property should only be displayed in detail screen fields with an edit type of image capture.

This property type will simply store an image captured from the client device’s camera or selected from the device’s file system. Its contents can be displayed to the user in detail screen fields of type image capture. It has no attributes beyond the standard property attributes. For an object these are the name and display name. If the parent is a transaction, which it should be in most cases, the standard transaction property attributes are set, including name, display name, the initial value attributes, and optionally special value attributes.

To synchronize data for an Image property, the file document management step type can be used to store the image as a `.jpg` file on the file system of the Agentry Server. Also, SDML data tags can be used to access the image data within other step types.

Integral Number Property Type

The integral number data type stores a whole number. An integral number property can define the minimum and maximum values it can contain. The hard minimum and maximum limits for this data type are equivalent to a 32-bit value, allowing for a positive/negative indicator bit.

Integral Number Property Attributes

- **Name:** This is the unique identifier for the property definition. This value must be unique among all properties within the same parent definition.
- **Display Name:** This attribute sets the default display name to label or otherwise identify the property value on the Agentry Client. This will also be the default label for any screen control that displays this property.
- **Minimum Value:** This attribute specifies the minimum value accepted by the property. This attribute can be no greater than the defined Maximum Value.
- **Maximum Value:** This attribute specifies the maximum value accepted by the property. This attribute can be no less than the defined Minimum Value.
- **Blank:** This attribute specifies whether to display a blank value or 0 when the value of the property is zero.

Location Property Type

A location property stores a location value returned by a GPS unit that includes the latitude, longitude, dilution, and number of satellites. The location property value can be invalid if the parameters of the property definition are not met. The transaction property location type includes attributes to define these parameters for the location value. The location property may also be set via rule functions that take the latitude and longitude values, converting them to a location value.

When defining this property type there are certain attributes specific to it for transactions and fetches verses objects. The object location property will contain attributes to initialize the value with a latitude, longitude, position dilution, and satellite count.

For a transaction or fetch location property, these same attributes can be set. In addition to these, there are also attributes to specify what is considered the minimum requirements for a valid location value for that property. These attributes set the minimum number of satellites, and the maximo age and position dilution for a location value returned from the GPS unit. If these minimums are not met, the behavior is definable within the property. The value can still be accepted, or it can be rejected.

For transaction and fetch location properties, there also exist the common initialization attributes. These attributes will override the defined latitude, longitude, position dilution, and number of satellites values for the property.

Location Property Attributes

- **Name:** Contains the internal unique name for the property definition. This value must be unique among all properties within the same parent definition.
- **Maximum Reading Age:** This attribute specifies the maximum reading age in seconds for the value returned by the GPS unit. This reading age represents the last time the unit took a reading. The Maximum Reading Age will dictate the oldest allowable reading for the location property. A location with a reading age older than the one specified in this attribute will be considered an invalid location.
- **Minimum Number of Satellites:** This attribute specifies the minimum number of satellites used to calculate the location. There is a minimum of 3 satellites required for any GPS location. A higher minimum may be specified. Note that this differs from the number of satellites the unit can see. This value specifies the number actually used to calculate the location. If this number is less than the minimum number specified the location will be considered invalid.
- **Maximum Position Dilution:** This attribute specifies the maximum acceptable position dilution for a location returned to the location property. This is an integral number with a range of values from 1 through 50, inclusive. If the position dilution returned with the location value exceeds this maximum the location will be considered invalid.
- **Accept Invalid Data:** This attribute specifies whether or not a location value that does not meet the criteria set for a valid location value to be accepted. If this attribute is set, invalid locations will be accepted. The property will return an invalid location value, which may be checked using the rule function term @IS_VALID_LOCATION.

Transaction and Fetch Attributes - The standard fetch and transaction attributes for initializing the property and targeting object properties are available for the Location property type. The attribute Initial Value includes the normal available settings plus the options listed below, which are specific to the Location type.

- **Current Location After Data Entry:** This option specifies the property should be updated to the device’s location after the transaction has been finished and just before it is applied. This value is obtained from the device’s GPS unit.
- **Current Location Before Data Entry:** This option specifies the property should be updated to the device’s location before the transaction is displayed to the user. This value is obtained from the device’s GPS unit.

Object Property Type

The object property data type is used to define an object as a property. The object property type stores a single object instance of a defined type as a property of a parent object, transaction, or fetch.

Object Property Attributes

Note: This property type does not have Special Value attributes.

- **Name:** This is the unique identifier for the property definition. This value must be unique among all properties within the same parent definition.
- **Display Name:** This attribute sets the default display name to label or otherwise identify the property value on the Client. This will also be the default label for any screen control that displays this property.
- **Object:** This attribute specifies the type of object the property is to contain. The object selected here must already exist within the same module before the property is defined.

Signature Property Type

The signature property type stores a signature entered by a user on the Agentry Client. This is an actual, written signature that can be entered on the device using a stylus or some other electronic pen. This signature is stored internally as a bitmap image. Normally only transaction definitions contain signature properties.

Signature properties may not be initialized to the value of another property. Also, it is outside the normal usage to target an object property with a transaction property of type signature. The primary intent of the signature property is to capture a signature in bitmap format on the client device and to then transfer that bitmap image to the back end system as a part of the transaction’s synchronization processing.

This property type includes definable behaviors covering the control that will display the property, and the minimum height and width of the bitmap image captured to treat as a valid signature.

This property type has several associated SDML data tags for accessing its bitmap data. The information on these should be reviewed when working with this property type.

Signature Property Attributes

Note: This property type does not have Special Value attributes.

General Attributes

- **Name:** This is the unique identifier for the property definition. This value must be unique among all properties within the same parent definition.

- **Display Name:** This attribute sets the default display name to label or otherwise identify the property value on the Client. This will also be the default label for any screen control that displays this property.
- **Required:** This attribute specifies whether or not the signature is required. For a transaction when this attribute is true, the user will not be able to advance the wizard until the signature has been captured. This includes meeting the *Minimum Signature Size Requirements* attribute settings.
- **Time and Date:** This attribute specifies whether to embed the client device's current date and time in the image.
- **Update Rule:** This attribute references a rule definition, the return value from which is expected to be a string. This value will be embedded in the bitmap image with the signature.
- **Signed:** This attribute contains the text value displayed in the detail screen field targeting the property when the signature has been captured.
- **Get Signature:** This attribute contains the text value displayed in the detail screen field targeting the property before the signature has been captured.

Maximum Window Size

- **Height:** This attribute specifies the maximum height of the window, in pixels, where the signature is entered.
- **Width:** This attribute specifies the maximum width of the window, in pixels, where the signature is entered.

Minimum Required Signature Size

- **Height:** This attribute specifies the minimum height, in pixels, for the signature value. If the signature does not meet this minimum, the signature will not be accepted. If the signature is required, the user will not be able to advance the wizard until the signature has been entered with this minimum height.
- **Width:** This attribute specifies the minimum width, in pixels, for the signature value. If the signature does not meet this minimum, the signature will not be accepted. If the signature is required, the user will not be able to advance the wizard until the signature has been entered with this minimum width.

String Property Type

The string property data type stores any character values as a single string. Definable behaviors of a string property include the ability to word wrap its contents upon display, to trim leading or trailing spaces within the string, and to treat the value as a password.

String Property Attributes

- **Name:** This is the unique identifier for the property definition. This value must be unique among all properties within the same parent definition.

- **Display Name:** This attribute sets the default display name to label or otherwise identify the property value on the Agentry Client. This will also be the default label for any screen control that displays this property.
- **Case:** This attribute specifies the case of the characters within the string property. This also can affect how multiple parent definitions, such as objects, are sorted based on the value of the property. The options for this attribute are: Lowercase Only, Uppercase Only, Mixed (case-insensitive), and Mixed (case-sensitive). The lower and uppercase settings will force any characters within the property to either lower or uppercase, respectively. The two Mixed case options will preserve the case of the characters as entered. The case-sensitive and case-insensitive settings specify how the value of the property is compared, either with respect to case or ignore it.
- **Format:** This attribute specifies how the value contained within the string property should be treated by the device or, more specifically, the operating system of the device. The options available for this attribute include email address, telephone number, and URL. Specifying this one of these options will result in the value being passed to the operating system with instructions to “open” the value in the corresponding application for the selected format; e.g. specifying the option URL will open the device’s web browser and navigate to the value in the string property.
- **Minimum Length:** This attribute specifies the minimum number of characters the property will accept. For transaction string properties, the user will not be able to advance the wizard unless this minimum number of characters is entered or set for the property.
- **Maximum Length:** This attribute specifies the maximum number of characters the property can contain. Editable fields displaying this property will not allow the entry of more than this number of characters. Object properties will truncate any value to this maximum number of characters for the string property.
- **Carriage Return:** This attribute will affect properties that are displayed in fields with multiple lines. If set to true, when the user hits the Enter key, or if a carriage return value exists in the string, a new line will be started within the multi-line field.
- **Word Wrap:** This attribute is another that affects properties displayed in multi-line fields. When set to true, if the text contained within the property is longer than the width of the field in which it is displayed, it will automatically wrap to the next line of the field, rather than scrolling past the far right edge.
- **Password:** This attribute controls whether the value entered for the property should be displayed or hidden. When set to true, the value for the property will not be displayed, but rather each character will be replaced by an asterisk (*). This is also true when users enter a value for this property.
- **Trim:** This attribute specifies whether white space characters at the beginning or end of the string should be preserved. When set to true, any leading or trailing white space will be trimmed from the value. Any white space within the string will not be trimmed.

Time Property Type

The time property data type stores a time of day value. This value is stored internally as the number of seconds after midnight, with midnight itself represented as 0. The default display format of a time property is HH:MM:SS am/pm.

Time Property Attributes

- **Name:** This is the unique identifier for the property definition. This value must be unique among all properties within the same parent definition.
- **Display Name:** This attribute sets the default display name to label or otherwise identify the property value on the Client. This will also be the default label for any screen control that displays this property.

Push

A push defines when it is necessary to push an object from the back end system to the Agentry Client and how that object's data is retrieved. A push provides real-time data synchronization for server-to-client data transfer, targeting a top-level object collection property within the same module. The push determines if changes have been made to the back end system and also retrieves the needed data to send those changes to the client. Part of the push definition is the optional behavior to notify users when data has been pushed to their clients.

There are five child definitions to the push, each a step usage definition. These steps provide the behaviors of polling the back end system for data changes that include new or modified business objects, polling and retrieving data for objects to be removed from clients, steps to retrieve data when changes have been found during a poll, updating the back end system after the new objects have been processed, and error handling steps.

The push itself defines how often to poll the back end system for changes, whether or not to display notifications on the Client of data after is has been pushed down and the nature of that notification, and an optional action that may be executed for each object pushed down to a Client.

The Push definition is the primary definition for implementing push behavior, but it is not the only definition type involved. The application-level definition transmit configuration also plays a part in this behavior. Specifically, a transmit configuration must be defined to maintain a constant connection between the Agentry Client and Agentry Server.

Additionally, if the system connection to be used for push processing is a SQL Database connection type, the configuration file `SqlBE.ini` for the Agentry Server is likely to need modification. Two sections within this file, `EnablePushUser` and `DisablePushUser` are processed by the Server as a part of the overall push processing for a database system. For other system connection types this file is not involved. Information on this configuration file can be found in the `Agentry Implementation Guide` for both Windows and Linux.

Push Child Definitions

- **Retrieval Step:** A push retrieval step references a step definition run to determine if object data has changed in the back end system and how that data is retrieved.
- **Removal Step:** A push removal step references a step definition run to determine what objects should be removed from the collection on the Agentry Client.
- **Read Step:** A push read step references a step definition to be run to continue the data retrieval for the object collection targeted by the push.
- **Response Step:** A push response step references a step to be run after the Agentry Server receives notification from the Agentry Client that an object has been successfully pushed down.
- **Error Step:** A push error step references a step definition to be executed when one of the other step usage definitions within the push return an error.

Push Attributes

General Attributes

- **Collection:** This attribute specifies the target object collection for which the push will synchronize data. This must be a top-level object collection property within the same module as the push definition.
- **Name:** This is the internal definition name for the push. This must be unique among all push definitions within the same module.
- **Display Name:** This is the default display value for the push definition when reference is made to it on the Agentry Client.
- **Poll Interval:** This attribute contains a duration value in hours, minutes and seconds, specifying how often to poll the back end system for modifications. At each poll interval the push retrieval steps and push removal steps will be processed by the Agentry Server. The value of the **Poll Interval** must be less than the transmit configuration attribute **Inactive Time** attribute of the transmit configuration defined to support the push behavior.
- **Read Steps:** This attribute specifies whether to use the read steps defined in the object type for the push's target collection property. When this attribute is checked, the read steps in the object type will be processed to synchronize data rather than the push read steps.
- **Queue Messages:** This attribute allows for push messages from the Agentry Server to the Agentry Client to be queued if they are not successfully sent after the first attempt or if the Agentry Client indicates it is still processing the previously received message.

Notification Attributes

- **Dialog Pops Up:** This attribute specifies whether or not a notification dialog is displayed on the Agentry Client after an object has been pushed down to the Client and when that dialog should be displayed. The options for this attribute are:
  - *After all data received:* This setting will display the notification after all objects have been successfully pushed to the Client.

    - *Immediately:* This setting will display the notification after each object has been successfully pushed to the Client. If multiple objects are pushed based on a single poll, the notification dialog will displayed once for each object.
    - *No Dialog (sound only):* This setting will not display any notification dialog to the user, with the client's default system sound being the only one played.
    - *When user clicks icon:* This setting will display an icon on the Agentry Client after objects have been pushed down. The notification dialog will then not be displayed until the user clicks this icon.
- **Data Received Text:** This attribute specifies whether or not to display notification text for new or replaced object instances and, if enabled, the contents of the notification message. This includes both the message to display to the user and the text in the notification dialogs title bar.
- **Data Removed Text:** This attribute specifies whether or not to display notification text when object instances are removed and, if enabled, the contents of the notification message. This includes both the message to display to the user and the text in the notification dialogs title bar.
- **Rec'd & Rem'd Text:** This attribute specifies whether or not to display notification text when both new object instances received from the push and other object instances are removed. If enabled, the contents of the notification message are also a part of this attribute setting. This includes both the message to display to the user and the text in the notification dialogs title bar.
- **Notification:** This attribute enables or disables the external notification behavior. When enabled, the client device's hardware LED light will be activated by the Agentry Client. This attribute has no affect on client devices without such hardware.
- **Play Sound:** This attribute enables or disables the system's default sound for notification when objects are synchronized by the push. When this attribute is enabled, the option for how many times to play the sound is set. This will also enable the Interval Between Sounds attribute.
- **Interval Between Sounds:** This attribute is enabled when the Play Sound attribute is enabled and it specifies the sound be played multiple times. The Interval Between Sounds attribute then specifies the duration of time between each instance of the sound.

Action Attributes

- **Action After Object Received:** This attribute references an action to execute when an object is pushed down to the Agentry Client. When an action is selected here, that action will be executed for each object instance pushed to the Client, targeting that object. This action is not executed when objects are removed based on the push synchronization. If the user is currently executing an action when an object is pushed to the client (e.g., the user is viewing a transaction wizard), the action defined for the push is queued and will be executed when the current action is completed.
- **Action When Push Completes:** This attribute references an action to execute when the push has finished pushing down all object instances to the Agentry Client. This action targets the object collection targeted by the push.

- **Cancel Action:** This attribute specifies whether or not any action currently being executed on the Agentry Client is cancelled when objects are pushed down. If this option is disabled and an action is being executed while an object is being pushed down to the Agentry Client, the object will not be received by the Agentry Client.

Push Retrieval Step

A push retrieval step references a step definition run to determine if data has changed in the back end system and how that data is retrieved. A retrieval step can be executed either once, or iteratively based on the number of users logged in to receive push data. A push retrieval step is run as a part of the back end polling performed by the push. The step referenced by a push retrieval step is expected to return the key property of any object instances to be pushed to the Agentry Client. It may also return additional property values for the object type. If defined to one run once per poll period it is also expected to return the Client user ID to which the object will be sent.

A push retrieval step can also return data for the object type targeted by the fetch, as well as for its child objects. However, it is recommended that child, or nested collections be synchronized by the push read steps or the object read steps (depending on how the push is defined). Since retrieval steps are run every poll period, the step definitions they use should be defined to perform the least amount of processing necessary to determine if new object data needs to be retrieved. Non-collection property values for the target object type can also be retrieved by the retrieval step in this model, but no additional data should be retrieved here for the sake of efficiency of the push's polling activity.

The retrieval step may be executed, or run, in one of two ways for a given poll. First it may be defined to run once per user currently logged in to receive push data. For this type of execution, the data returned by the retrieval step will be organized internally by the Server for each user. The step being run should then include logic that includes retrieving data specific to each user, matching the criteria for the implementation related to how objects are synchronized. The step itself will then be executed multiple times, once for each user, during a single poll of the back end. Note that this can be a significant number of executions in a production environment, where it is common for hundreds of users to be connected to the Server for push processing. When run in this manner, the user ID value for each user is acessible via the `<<userID>>` SDML data tag.

The second option for running push retrieval steps is to run them once per poll period. This behavior will run the step a single time for a given poll regardless of the number of users currently connected to the Server for push data. In this scenario, the data returned by the step must include the user ID specifying which client user will receive a given object. This value should be identified to the Server as UserID. Once the object instance has been created by the Server and the synchronization of the push overall is completed, the object instance will be pushed to that user. When run in this manner individual user ID's are not available to the step.

When a push retrieval step returns the key property of the push's target object type, the push read steps or the object read steps (depending on the definition of the push) will be run to continue the synchronization of the target object collection. If no key property is returned by any push retrieval step, it is assumed no new data needs to be pushed to the Client. No read steps will be run in this situation.

Retrieval Step Attributes

- **Step:** This attribute references the step definition within the same module to be run as a push retrieval step. These steps may return values for any property within the targeted object type of the push, but must return the key property of that object to indicate that one or more objects should be synchronized by the push and sent to the Client. For steps run once, the data returned by the step must also include the Client user ID to which the object will be sent.
- **Run:** This attribute specifies how often to run the referenced step during a single poll interval. This may be set to either Run One Time, or Run Once Per User. The former will execute the step once for a given poll period and the user ID value is not available. The latter will execute the step once per user currently connected to the Server for push data during a given poll period and the user ID value is available.
- **Read Into:** This attribute specifies for which object within the data structure of the targeted object collection the step will return data. While retrieval steps can return data for nested collections, it is recommended that this be handled by the push read steps, as these will only be run when the retrieval steps indicate new data is needed.

Push Removal Step

A push removal step references a step definition run to determine what objects should be removed from the target collection on the Agentry Client. A removal step can be executed either once, or iteratively based on the number of users logged in to receive push data. A removal step is run as a part of the back end polling performed by the push. The step referenced by a push removal step definition is expected to return the key property of any object instance to be deleted from the Client. If defined to run once per poll period it is also expected to return the Client user ID from which the object will be removed.

A removal step may be run once per poll of the back end system, or once per user per poll period, depending on how it is defined. When a removal step is run once per user, data returned by the step will be organized according to each user. Note that this scenario can result in a large number of executions of the step per poll, as the number of users logged in into the Server is commonly in the hundreds or more in a production environment. When run in this manner, the user ID value for each user is accessible via the `<<userID>>` SDML data tag.

When a removal step is defined to run one time, it will be executed once per poll period, regardless of the number of users connected to the Server. In this situation, the data returned by the removal step should also include the user ID as entered on the Client, indicating which client will receive the key property for the object to be removed. It is recommended that this is how most, if not all removal steps are defined within a push as it is a more efficient model of data synchronization. When run in this manner individual user ID’s are not available to the step.

Removal Step Attributes

- **Step:** This attribute specifies the step definition within the same module as the push to be run as a push removal step.
- **Run:** This attribute specifies how often to run the step in a given poll period. The options for this attribute are Run One Time or Run Once Per User. The former will execute the step once during the poll period and individual user ID's are not available. The latter will execute the step once per user currently connected for push data and individual user ID's are available to the step.
- *Read Into:* Not currently supported - leave set to default.

Push Read Step

A push read step references a step definition to be run to continue to read data for the target object collection. This is a continuation of the synchronization process begun by the retrieval steps for the push. A push read step can be executed once or iteratively based on the number of objects and number of users logged in to receive push data. A push read step is only run if the retrieval steps indicate their are objects to be retrieved.

The execution of the read step can be based on the number of objects created by the retrieval steps, the number of objects in a nested collection created by previous retrieval or read steps, or based on the number of users currently connected to receive push data.

When run once per user, the step will have access to individual user ID's via the SDML data tag `<<user.agentryID>>`. Also, any values stored in the `<<user.info>>` data tag are still available. When run in this manner it is important to note that the step will be executed once for each user during each poll period when the retrieval steps indicate there is data to be synchronized. For production systems it is not uncommon for the number of users connected for push data to be in the hundreds or more.

When run once per object, the read step will be executed for each object instance for the collection targeted by the push created by any other push steps prior to the current step's execution. When run in this manner the step will have access to the key property of the object for which it is currently executing.

When run once per collection object, the read step should target (Read Into) a nested collection of the collection targeted by the push. The step will then be executed once for each object instance within in this nested collection created by the push steps prior to the current step's execution. The step is expected to return property values for the object type stored in the nested collection. It should also return the key property of any object between the nested object and the top-level object type in the data hierarchy of the module. This configuration is primarily intended for file transfer functionality and it is recommended it not be used for other purposes unless no alternative is available.

When the read step is defined to run once per poll period, it is expected to return data for the object type in the collection it targets, which will either be the same as the push's target collection, or a nested collection of that target. It will not have access to individual user ID's or to any object key properties. It must also return the Client user ID indicating to which user the object should be sent.

Read Step Attributes

- **Step:** This attribute specifies the step definition within the same module as the push to be run as a push read step.
- **Run:** This attribute specifies how often to run the step in a given poll period. The options for this attribute are Run One Time or Run Once Per User. The former will execute the step once during the poll period and individual user information is not available. The latter will execute the step once per user currently connected for push data and individual user information, including user ID's are available to the step.
- **Read Into:** This attribute specifies which objects to create with the data returned by the step. This may be the same collection as is targeted by the fetch, or one of its nested collections. To read data into the push's target collection, this attribute is left set to its default value of "None." For nested collections, the desired collection is selected in the Add Wizard or in the properties view.

Push Response Step

A push response step references a step to be run when the Agentry Server receives notification from the Agentry Client that an object has been successfully pushed down. This step is run to update the necessary back end objects that the object has been processed by the push. A push response step is always executed once per object pushed to the Client.

Response Step Attributes

- **Step:** This attribute specifies the step definition within the same module as the push to be run as a push response step.
- **Run:** This attribute cannot be changed for this step usage definition. This step type is always run once per object.
- *Read Into:* Not currently supported - leave set to default.

Push Error Handling Step

A push error step references a step definition to be executed when one of the other step usage definitions within the push return an error. An error step is always executed once per object.

The intended purpose of a push error step is to perform any cleanup or similar actions in the event an error occurs in processing on of the push's steps. This may include items such as marking the object data in the back end as not pushed, or other similar processing.

Error Handling Step Attributes

- **Step:** This attribute specifies the step definition within the same module as the push to be run as a push error step.

- **Run:** This attribute cannot be changed for this step usage definition. This step type is always run once per object.
- *Read Into:* Not currently supported - leave set to default.

Service Event

A service event defines how the Agentry Server synchronizes data between two back end systems. A service event will normally perform such a synchronization when a change or “event” occurs in a source back end system that must be reflected in a destination back end system. Depending on its type, a Service Event can either actively poll a back end system, or listen to the source back end for messages notifying it of a change. A service event targets an object collection to facilitate this data transfer, with the object instances in that collection storing the data retrieved from the source back end. The synchronization processing of a service event does not involve or affect any Agentry Clients.

The service event creates object instances based on data retrieved from the source back end system. It then updates this object data to the destination back end system. The components of the service event that retrieve data from the source back end system differ for each service event type. The child definitions to update the destination back end system are the same set of step usage definitions for all service event types.

There are four types of service events that may be defined:

- **Poll With Step:** A Poll With Step service event type references a step definition that is run by the Agentry Server periodically to actively poll the source back end system for data changes.
- **Java Callback:** A Java Callback service event type includes a Java code component that is an extension of the ServiceEvent Agentry Java API class that allows a the source back end system to call into the Agentry Server as a notification of a modification to that back end system’s data.
- **HTTP-XML Message Received:** An HTTP-XML message received service event type includes XML message mappings that will map messages sent from the source back end system to the Agentry Server to indicate data has changed in that back end system.
- **File System Monitor:** A File System Monitor service event type is defined to monitor a specified directory on the Agentry Server’s host file system for changes and includes document mappings to map data in that directory’s files to the properties of the service event’s target object type.

Service Event Child Definitions

While each service event type has different components to capture data changes in the source back end system, all service event types contain the same child definitions to update the destination back end system:

- **Read Step:** A service event read step references a step definition run to retrieve any additional data for the target object collection from the source back end system.

- **Data State Step:** A service event data state step references a step definition run to check for data collisions in the destination back end system before the service event makes any changes to it.
- **Update Step:** A service event update step references a step definition run to update the destination back end system with data stored in the service event's target object collection.
- **Error Handling Step:** A service event error handling step references a step definition run only when one of the other service event child step usage definitions returns an error.

Service Event Attributes

The attributes for a service event vary depending on the service event type. See the service event type-specific information for details on these attributes.

Service Event Type: Poll With Step

A Poll With Step service event type references a step definition that is run by the Agentry Server periodically to actively poll the source back end system for data changes. This step definition is commonly a SQL step, though this is not a requirement. The step polling the source back end must return the key property of the object for which the service event has been defined to indicate there is data to be updated to the destination back end system. When the step returns this value, the service event's read, data state, and update steps are processed.

This service event type is typically defined when the source back end is a SQL Database system connection, although any back end may be actively polled provided the correct step type definition is used. The only requirement of the step referenced by the service event to poll the source back end is that it return the key property for each object instance of the target object type to be synchronized.

This service event type includes the poll interval, which is the duration of time between polls of the source back end by the service event. The number of objects retrieved from the source back end can be limited to a maximum number of instances per poll interval.

Poll With Step Service Event Attributes

- **Name:** The internal unique name for the service event definition, must be unique among all service events defined within the same module.
- **Step:** This attribute references the step definition to be run by the service event to poll the source back end system. The step referenced here should be written to return the key property of the target object type of the service event whenever data has changed in the source back end.
- **Object Type:** This attribute specifies the target object type of the service event. The properties of this object will store the data retrieved from the source back end to be updated to the destination back end. This may include child object collections where necessary.
- **Poll Interval:** This attribute specifies the duration of time between polls of the source back end by the service event. This attribute is set in hours, minutes and seconds. The step

definition referenced in the Step attribute will be run periodically based on the value in the Poll Interval attribute.

- **Object Limit:** This attribute specifies the maximum number of object instances to create using the data returned by the service event's defined step. If this step returns the key property values for objects than specified here, the order in which the step returns them will determine which object instances will be created and which will not.
- **Read Steps:** This attribute specifies whether to retrieve data from the source back end system using the read steps defined in the service event, or those defined in the target object type of the service event. When this attribute is true, the object's read steps will be used and the service event's read steps will not be processed.

Service Event Type: Java Callback

A Java Callback service event type includes a Java code component that is an extension of the ServiceEvent Agentry Java API class. Included in this class must be a method into which the source back end system can the Agentry Server as a notification of a modification to that back end system's data. this class is instantiated when the service event is loaded by the Server during startup. The information passed to this method must include the key property of the service event's defined object type. When a message is received by this class that includes the service event's target object's key property the read steps, data state steps, and update steps will be processed by the Agentry Server to update the destination back end system.

The Java code component of this service event type is initially created as a skeleton class. The developer must then implement the methods for this class to process the message received from the source back end system. It may include additional methods for processing the message once received, but only one method within the class can be called by the source back end. Data captured from the source back end must then be passed from this class to the Agentry Server using the standards within the Agentry Java API.

Java Callback Service Event Attributes:

- **Name:** The internal unique name for the service event definition, must be unique among all service events defined within the same module.
- **Connection:** This attribute references a Java Virtual Machine system connection within the application. This system connection is the one to with which the service event will communicate. This connection will be the one over which the source back end system sends the message to the service event when a data change occurs. This system connection must exist prior to defining the service event and it must be of type Java Virtual Machine.
- **Object Type:** This attribute specifies the target object type of the service event. The properties of this object will store the data retrieved from the source back end to be updated to the destination back end. This may include child object collections where necessary.
- **Read Steps:** This attribute specifies whether to retrieve data from the source back end system using the read steps defined in the service event, or those defined in the target object type of the service event. When this attribute is true, the object's read steps will be used and the service event's read steps will not be processed.

Service Event Type: HTTP-XML Message Received

An HTTP-XML message received service event type includes XML message mappings that will map messages sent from the source back end system to the Agentry Server to indicate data has changed in that back end system. This call is made by the back end system via a CGI message containing XML data or an XML document. The service event will filter messages using an XPath value. The service event will then handle this message by processing the read, data state, and update step definitions to update the destination back end system.

This type of service event includes a child definition called message mapping. A given HTTP-XML Message Received service event may have one or more message mapping. Each mapping definition is intended to map data from the XML message or document to the properties of the target object of the service event, or to one of a selection of other data items within the application.

As a part of the HTTP-XML service event type, an HTTP response is defined. This response is sent by the Agentry Server back to the HTTP server that initially sent the XML message; that is, the response is sent back to the source back end system. Included in this response is one of the standard HTTP response status codes, as well as other possible information. This information can include fixed string data, or HTTP markup. This response after all processing for the service event has completed, including all read, data state, and update step execution.

HTTP-XML Message Received Service Event - General Attributes

- **Name:** The internal unique name for the service event definition, must be unique among all service events defined within the same module.
- **Connection:** This attribute references an HTTP-XML system connection type within the same application. This connection is the one over which the source back end system will send the message to the Agentry Server containing the XML data or document to be processed by the service event. This system connection must exist prior to defining the HTTP-XML Message Received service event and must be of type HTTP-XML.
- **Message Filter:** This attribute contains the XPath statement to filter messages received by the service event. If the service event can select one or more nodes within the message document using this XPath, the service event will process the message. If it cannot make such a selection, it will ignore the message.
- **Object Type:** This attribute specifies the target object type of the service event. The properties of this object will store the data retrieved from the source back end to be updated to the destination back end. This may include child object collections where necessary.
- **Read Steps:** This attribute specifies whether to retrieve data from the source back end system using the read steps defined in the service event, or those defined in the target object type of the service event. When this attribute is true, the object’s read steps will be used and the service event’s read steps will not be processed.
- **Run Push:** This attribute specifies a push definition to run when this service event type is called from the back end system. This is provided as an alternative to the default polling

behavior of a push definition, allowing them to be run on demand by the back end system instead.

HTTP-XML Message Received Service Event - HTTP Response Attributes

HTTP Response

- **HTTP Response Code:** This attribute specifies the HTTP response status code sent by the Agentry Server to the source back end system that initially called the service event.
- **Response Data Type:** This attribute specifies the data type of any data within the HTTP response status sent by the Agentry Server. This can be set to Fixed String, Small Markup or Large Markup. A fixed string is a single string value defined in the **Response** attribute. Small Markup is a short piece of HTML text, also set in the **Response** attribute. Large Markup is a larger chunk of HTML text, likely spanning multiple lines. This text is stored in the file referenced by the **Markup File** attribute.
- **Markup File:** This attribute is available only when the Response Data Type is set to Large Markup. Markup File references the text file containing the HTML text to be sent as a part of the HTTP Response by the service event.
- **Response:** This attribute is available when Response Data Type is set to either Fixed String or Small Markup. For fixed string, the Response attribute can contain any text value. For Small Markup the Response attribute should contain HTML text. The contents of the Response attribute are sent as a part of the HTTP Response by the service event.

Error Response

- **Error Response Code:** This attribute specifies the HTTP response status code sent by the Agentry Server to the source back end system that initially called the service event.
- **Error Data Type:** This attribute specifies the data type of any data within the HTTP response status sent by the Agentry Server. This can be set to Fixed String, Small Markup or Large Markup. A fixed string is a single string value defined in the **Response** attribute. Small Markup is a short piece of HTML text, also set in the **Response** attribute. Large Markup is a larger chunk of HTML text, likely spanning multiple lines. This text is stored in the file referenced by the **Markup File** attribute.
- **Markup File:** This attribute is available only when the **Response Data Type** is set to Large Markup. **Markup File** references the text file containing the HTML text to be sent as a part of the HTTP Response by the service event.
- **Response:** This attribute is available when **Response Data Type** is set to either Fixed String or Small Markup. For fixed string, the **Response** attribute can contain any text value. For Small Markup the **Response** attribute should contain HTML text. The contents of the **Response** attribute are included as a part of the HTTP Response sent by the service event.

HTTP-XML Service Event Message Mapping

The HTTP-XML service event type includes the child definition type Message Mapping. Service events of this type can contain one or more of these child definitions. The purpose of a message mapping is to map data in the XML document that is a part of the message received by the service event to the properties or other data values within the application.

A part of the message mapping definition is the XPath to the location of the data in the XML structure. When a data value is found it is then mapped, according to the message mapping, to either the property values of the object type targeted by the service event, or to one of a list of other options for data sources within the application.

HTTP XML Service Event Message Mapping Attributes

- **Mapping Type:** This attribute specifies the mapping type. This may be either XPath Expression or XML Transformation.
- **Base XPath:** This attribute is optional and should be used when returning multiple instances of the same data element in the XML content. When a **Base XPath** is defined for a response mapping, the same value will be set by default in the add wizard for subsequent message mappings within the same parent service event.
- **XPath:** This attribute is only available when the **Mapping Type** is set to XPath Expression. This attribute contains the XPath value to extract the desired value from structured XML data contained in the message received by the service event.
- **XSL:** This attribute is only available when the **Mapping Type** is set to XML Transformation. It contains the XSL expression to transform the XML data contained in the message received by the service event.
- **Maps To:** This attribute specifies where the value extracted by the message mapping is stored in the application. This may be one of the following values for a service event:
  - *Last Update:* This selection specifies the extracted value is a date and time indicating when the object’s source in the back end system was last modified. This value is mapped to the last update value within the object instance created by the service event.
  - *Local String (`<<local>>`):* This selection will create a local data tag available to subsequent message mappings in the same parent service event. The value of this data tag will be the value extracted by the response mapping. When selected, the attribute String Name will be available to name the local data tag created. This is the equivalent to calling the SDML function tag `<<local ...>>`.
  - *Local XML (`<<localXML>>`):* This selection will create a local XML data tag available to subsequent message mappings in the same service event. The value of this data tag will be the value extracted by the response mapping.
  - *Parent Object Key Property:* This selection will set the value extracted by the message mapping to the key property of the parent object to the object created by the message received by the service event. This will not change the parent object’s key property, but rather is used by the Agentry Server to identify which object is the parent object.
  - *Property Path:* This select will set the value extracted by the message mapping to the property selected in the Property Path field. This will change the value of the property to the value extracted from the message received by the service event.

Service Event Type: File System Monitor

A File System Monitor service event type is defined to monitor a specified directory on the Agentry Server's host file system for changes. When a file system monitor service event is defined and published to the Server, the Server will begin monitoring the directory the service event has been defined to watch, which is then the source back end system for the service event. When a change occurs to the contents of this directory, the Server will attempt to open each file in the directory for reading. Once a file is opened successfully, the service event's defined command will be executed, followed by its read, data state, and update step definitions to update the destination back end system.

If the Agentry Server is unable to open a file for reading, it will wait a short period and attempt the operation again. If it is unable to open the file after this second attempt, it will skip the file and process the next on found in the directory. When a file is successfully opened, the defined command for the service event is executed for that file. The service events child step usage definitions are then processed. Once this is complete, the next file in the directory is processed according this same procedure. Note that the Agentry Server will delete all files from the directory that it successfully processes. To prevent this behavior, the command should include copying the file to a different location, or rename the file to one that does not match the **File Filter** attribute of the service event definition.

The command defined for this service event type is primarily intended to prepare files for transfer or reading by the service event. The command can be any executable file type and is a Windows batch file (`.bat`) by default. The file type may be changed by editing the file extension of the file name in the **File** attribute of the Command tab.

A file system monitor service event includes the document mapping child definition type. The service event may contain one or more of these child definitions, each of which will map a file or file-related data to the properties of the object type targeted by the service event; or alternately one of the other data components of the application.

File System Monitor Service Event Attributes

- **Name:** The internal unique name for the service event definition, must be unique among all service events defined within the same module.
- **Connection:** This attribute references a File system connection type to the Agentry Server's host file system. The directory monitored by the service event must exist on this file system. The Agentry Server must have read-write access to this directory. The system connection must be of type File System and must exist prior to defining the service event.
- **Directory:** This attribute specifies the directory to be monitored by the service event. This path can be either a full path beginning with the file system root, or it may be a relative path to the installation location of the Agentry Server. The Server must have read-write privileges to this directory. Changes to this directory monitored by the service event include the addition of new files or modifications of existing files determined by changes to the modification date in the file's metadata.

- **File Filter:** This attribute can contain any file name matching characters to specify which files or file types the service event should monitor. This value can include wild cards in the form of asterisks. Any files at the location specified by the Directory attribute that match the file name pattern specified in File Filter will be monitored. Any others will be ignored by the service event. If File Filter is left blank, all files within the directory will be monitored.
- **Object Type:** This attribute specifies the target object type of the service event. The properties of this object will store the data retrieved from the source back end to be updated to the destination back end. This may include child object collections where necessary.
- **Read Steps:** This attribute specifies whether to retrieve data from the source back end system using the read steps defined in the service event, or those defined in the target object type of the service event. When this attribute is true, the object's read steps will be used and the service event's read steps will not be processed.

File System Monitor Service Event Document Mapping

The File System Monitor service event type includes the child definition type Document Mapping. A document mapping is defined to map a file or other data generated by the service event's command to a property within the application. Data that may be mapped includes a file created by the command, output written by the command to standard error or standard output, or the exit code passed to the operating system by the command.

Multiple document mappings may be defined for the same parent service event to capture each of these values. This can allow the application to determine if an error occurs when the service event's command is executed, as well as specific information about the error.

File System Monitor Service Event Document Mapping Attributes

- **Property:** This attribute specifies the property to which the data extracted by the document mapping is assigned. The data type of the property selected should reflect the setting for the Output Type attribute.
- **Output Type:** This attribute specifies which output from the service event's command contains the data to be mapped to the item referenced in the **Property** attribute. This may be set to one of the following options:
  - *Command Exit Code:* This selection specifies the value returned by the command to the operating system. This exit code will be the value stored in the item selected in the Property attribute. The property selected should of type integral number to store the command exit code in most cases.
  - *File Created by Command:* This selection specifies that the file created by the service event's command should be assigned to the item referenced in the Property attribute. This data of the selected property should be External Data when this Output Type is defined.
  - *STDERR:* This selection specifies that any output from the service event's command written to STDERR, or standard error, is assigned to the item referenced in the Property

attribute. The data type of the selected property should be String when this Output Type is defined.

- *STDOUT:* This selection specifies that any output from the service event's command written to STDOUT, or standard output, is assigned to the item referenced in the Property attribute. The data type of the selected property should be String when this Output Type is defined.

- **File Name:** This attribute is available only when the Output Type is set to File Created by Command. The File Name attribute specifies the name of the file created by the service event's command that is to be referenced by the item selected in the Property attribute. This attribute can include the SDML <<script>>, which expands to the name of the file in which the service event's command is stored. It is common to use this value as a part of the name for the file generated by the command.
- **Delete File:** This attribute specifies whether or not to delete the file created by the service event's command. When set, the file will be removed after the service event has finished processing it. Otherwise the file will remain after the service event has completed processing.

Step

A step defines a single piece of processing to be performed by the Agentry Server with a specific back end system. There are different types of steps, defined based on the system connection for which the step is defined. A step defines what action to take and against which back end system. It will not be executed by the Agentry Server unless it is referenced by another definition that defines Server processing.

The step definition must be used by a step usage definition to give it context and purpose. The step itself defines the back end system and the task to perform. The context of the step will dictate what values the step will have access to within the application data.

Regardless of the type of step, all may update data to a back end system, including adding, editing, or deleting that data; and all may retrieve data from the back end system, returning it to the Agentry Server for use in the application.

There are five types of steps that may be defined, with one each for the Java Virtual Machine, SQL Database, and HTTP-XML system connection types. The File system connection supports two step types. When a step definition is created in the Agentry Editor the first information entered is the system connection for which the step is defined. Based on this selection the type of step can then be entered. Following are the types of step definitions that can be defined. The appropriate system connection must exist prior to defining the step.

- **SQL Query:** A SQL Query step is defined for a SQL Database system connection and contains the SQL logic to be processed by the Agentry Server for a database back end system.
- **Java Steplet:** A Java Step, or Steplet, is defined for a Java Virtual Machine system connection and contains the Java logic to be processed by the Agentry Server for a Java interface.

- **XML via HTTP:** An XML via HTTP step is defined for an HTTP-XML system connection and defines a URL called by the Agentry Server and also defines how the XML data returned from this call is mapped to the data members of the mobile application.
- **File Command Line:** A File Command Line step is defined for a File System Connection and defines a command to be executed by the Agentry Server on the host system of the Server.
- **File Document Management:** A File Document Management step is defined for a File System Connection and defines a command to be executed by the Agentry Server on the host system of the Server in support of transferring files between the Server and the Agentry Client.

Step Type: SQL Query

A SQL Query step is defined for a SQL Database system connection and contains the SQL logic to be processed by the Agentry Server for a database back end system. The logic for a SQL Query step is contained in a text file with a `.sql` extension. The contents of this file are processed by the Agentry Server prior to submission to the database for execution. This preprocessing includes expanding any SDML tags. The results from this expansion must be a valid SQL statement for the target database type.

The contents of the query file for this step type may be accessed directly on the Agentry Development Server for the application project. The file path listed for this file is relative to the installation location of this Server. If this file is modified it is not necessary to publish the application project for the change to be exhibited. In a production environment this file is not directly accessible in this manner and must be modified through the Agentry Editor. Changes made in a production environment must be published.

SQL Query Step Attributes

- **Name:** Contains the unique internal name of the step definition. This value must be unique among all step definitions within the same module.
- **Connection:** This attribute references the system connection for which the step is defined. This attribute is set when the step is initially created and cannot be modified. For a SQL Query step this must be a SQL Database system connection type.
- **File:** This attribute contains the path and file name of the `.sql` file containing the step's SQL statement. This path is relative to the path

  ```
  ServerDirectory\Application\Development\Scripts
  ```

  where `ServerDirectory` is the installation location of the Agentry Development Server.

Step Type: Java Steplet

A Java Step, or Steplet, is defined for a Java Virtual Machine system connection and contains the Java logic to be processed by the Agentry Server for a Java interface. The logic for a Java Steplet is contained in a Java source file with a `.java` extension. This file is added to an existing Java project in the Eclipse Java perspective. This class created is an extension of the Agentry Java API class `Steplet`. The contents of this file are processed by the Java Virtual Machine running on the host system for the Agentry Server.

With the release of the Agentry Mobile Platform version 5.1 the process for creating of a Java step definition has changed. The new procedure reflects support for the Java perspective provided within the Eclipse and allows the developer to add Java logic for a step definition to existing projects within the Java perspective. The creation of the Java logic portion of a step definition is now performed through the Java perspective's wizard for creating classes, and allows for the selection of the package to which the step is to be added. Java steps created in previous versions of the mobile platform are still supported and will still reside on the Server's file system. New Java steps defined for the application should be created using the Java wizards provided by the Java Perspective. The file for these steps will then be saved to the file system according to the configuration of the Java project to which the Steplet is added.

Java Steplet Attributes

- **Name:** Contains the unique internal name of the step definition. This value must be unique among all step definitions within the same module.
- **Connection:** This attribute references the system connection for which the step is defined. This attribute is set when the step is initially created and cannot be modified. For a Java Steplet this must be a Java Virtual Machine system connection type.
- **Source Type:** This attribute specifies the source of the Java logic for the step and may be set to one of the following options:
  - *Existing Class:* This selection will allow for the selection of an existing class within a project in the Java perspective. The class selected will be the one called when the Java step is processed by the Server at run time. This class must be an extension of the Agentry Java API class Steplet.
  - *New Class:* This selection will create a new Java class that is an extension of the Agentry Java API class Steplet. The Java class wizard within the Java perspective in Eclipse will be displayed to allow for the creation of this class, including specifying the project and Java package to which it should be added.
  - *Source (deprecated):* This selection, as indicated, is deprecated and is provided to support the now deprecated method of managing Java classes for an Agentry mobile application. This selection will use and store a `.java` file on the Agentry Server's file system. Note that this selection will prevent the ability to organize the source file for the step in a Java project within the Eclipse Java perspective.
- **File:** Note that this attribute is deprecated as of version 5.1 of the Agentry Mobile Platform. While still supported for existing Java steps, it should not be used in new step definitions. It is only valid when the Source Type attribute is set to the option “Source.” This attribute contains the path and file name of the `.java` file containing the step's SQL statement. This path is relative to the path

```
ServerDirectory\Application\Development\Scripts
```

where `ServerDirectory` is the installation location of the Agentry Development Server.

Step Type: XML via HTTP

An XML via HTTP step is defined for an HTTP-XML system connection and defines a URL called by the Agentry Server and also defines how the XML data returned from this call is mapped to the data members of the mobile application. This step includes two child definitions that encapsulate the arguments passed by the Server to the defined URL, and the mapping of return values to the data members of the application. An XML via HTTP step is defined for a specific definition type within the same module.

The first information entered for an XML via HTTP step is the definition for which it is defined, which may be an object, transaction, or fetch. This information is needed by the step definition for use in its child definitions, which must have access to the property values of the selected data definition as a part of their behaviors.

Within this step type is the HTTP request. This portion of the step defines the URL called by the Agentry Server and the HTTP request method. This may be one of GET, HEAD, POST, or PUT.

The child definitions to an XML via HTTP step include its request arguments and response mappings. Request arguments provide access to the property values and other data values in scope for the step to be passed to the URL defined by the step. Included in this definition is the type of argument the data represents.

Response mappings extract data from the structured XML data or document returned from the request. They may use of XPaths to locate and retrieve these values from the XML and define to which property or other data member of the application the XML contents will be mapped. Response mappings may be used to extract a specific XML element’s contents, or a parent element may be specified with a second, child element within that parent of which there may be multiple instances.

XML via HTTP Child Definitions

- **Request Argument:**
- **Response Mapping:**

XML via HTTP Step Attributes

General Attributes

- **Used For:** This attribute specifies the data definition within the module for which the step will synchronize data. This may be any fetch, transaction, or object definition within the application. This attribute set by first selecting the type of definition and then selecting the specific definition within the project.
- **Name:** Contains the unique internal name for the step definition. This must be unique among all steps within the same module.

- **Connection:** This attribute references the system connection for which the step is defined. This attribute is set when the step is initially created and cannot be modified. For an XML via HTTP step this must be an HTTP-XML system connection type.

HTTP Request Attributes - These attributes are accessible after the step has been defined. They are organized as a child definition to the step itself and can be navigated to in the Editor within the Application Explorer view. The HTTP Request for an XML via HTTP step is one of the rare instances within the Agentry project structure where there may be only one instance of a child definition within a given parent.

- **Name:** Contains the unique internal name for the HTTP request within the step definition. This is set by default to the name of the parent step definition. It may be modified. A given XML via HTTP step will have only one HTTP request.
- **URL:** This attribute contains the URL to which the Agentry Server will make a request. This value will be appended to the value configured as the base URL for the HTTP-XML system connection. This base URL is configured within the HTTP-XML system connection configuration options for the Agentry Server. Proper use of both this base URL option and the URL entered in the requests of the step definitions can support portability for the application, with the base URL being the implementation-specific portion and the step’s URL being the portion not likely to change for the same back end system from one implementation to the next.
- **Method:** This attribute specifies the HTTP request method for the request. This may be set to one of GET, HEAD, POST, or PUT.

XML via HTTP Step Request Argument

An HTTP Request Argument is a child definition to the XML via HTTP step definition, defining the data values passed as arguments to the parent step’s defined URL. Included in the request argument definition is the type of request and the property value or other data value in scope for the step to be passed as the argument. Request arguments also have a data type, which specifies the source for the argument’s data.

A request argument is defined for the parent step only when it is necessary to pass arguments to the step’s defined URL request. The request allows for access to the property values of the definition for which the step was defined, as well as values at the user or application level via the SDML. A fixed string value may also be defined to passed as the argument.

The values accessed via the SDML can be contained in either a small or large markup value. Both allow for the use of HTML markup text. The difference between these two items is the manner in which the markup is stored. For a small markup argument, a single field that can contain one line of markup text is available within the request argument.

Each of these data sources is a different data type within the request argument. A data type is selected first within the definition, followed by the specific value of that type.

The argument itself also has a type. This may be one of CGI Argument, Cookie, HTTP Header, or XML Body. The selection of the argument type specifies how the data for the request argument is passed to the URL defined in the parent step definition.

A given XML via HTTP step can contain multiple arguments. All arguments are listed within the Properties View of the parent step definition. Within this list, the position of each request argument definition specifies the order in which the arguments will be passed to the URL request. This order can be changed by moving the arguments up or down within the list.

Request Argument Attributes

- **Argument Type:** This attribute specifies how the data within the argument will be passed to the URL defined in the parent step definition. This may be one of CGI Argument, Cookie, HTTP Header, or XML Body.
- **Name:** Contains the unique internal name of the request argument. This value must be unique among all request arguments within the same step definition. The field label in the Editor for this attribute will change based on the selection of the Argument Type attribute.
- **Data Type:** This attribute specifies the type of the data for the argument. This selection determines the source for the argument data within the mobile application.
  - *Fixed String:* This selection specifies the argument will be a plain text value. When this is selected the String attribute will be enabled, allowing for the entry of text value to be passed for the argument.
  - *Large Markup:* This selection specifies the argument will be HTML markup. The markup text will be stored in a text file. This file is accessible on the Agentry Development Server for the application project and may be edited directly from this location, or from within the Agentry Editor. The relative path and name for this file is listed in the Markup File attribute, which is enabled when the “Large Markup” type is selected in the **Data Type** attribute.
  - *Property Path:* This selection specifies that the argument value is contained in a property of the definition for which the parent step has been defined. When this selection is made the Property Path attribute is enabled, where the property can be selected.
  - *Small Markup:* This selection specifies the argument will be HTML markup. The markup text will be entered in the Markup Text attribute field, which will be enabled for this selection. This argument Data Type allows for a single line of HTML markup to be entered for the argument.
  - *User ID:* This selection specifies that user’s login ID for the Agentry Client is passed as the argument value. No other attributes are enabled in relation to this selection.
- **String:** This attribute is enabled when the Data Type attribute is set to Fixed String. The String attribute contains the plain text value passed as the argument value to the parent step’s URL request.
- **Mask in Log:** This attribute can be set to hide the value of the argument in logs generated by the Agentry Server. in place of the value, a series of asterisks is recorded. Typically this is used for passwords and other sensitive values.
- **Markup File:** This attribute is enabled when the Data Type attribute is set to Large Markup. The Markup File attribute lists the relative path and file name for the text file containing the HTML markup text. This path is relative to the Agentry Development Server’s installation location. The default value is the location:

```
ServerDirectory\Application\Development\Scripts
```

where `ServerDirectory` is the installation location for the development server. This path may be changed, relative to this location, though this is rarely necessary for this definition type.

- **Property Path:** This attribute is enabled when the Data Type attribute is set to Property Path. The Property Path attribute references the property definition within the definition for which the parent step was defined. The value of this property will the value passed as the argument to the parent step's URL request.
- **Markup Text:** This attribute is enabled when the Data Type attribute is set to Small Markup. The Markup Text attribute can contain a single line of HTML markup text that will be passed as the argument to the parent step's URL request.

XML via HTTP Step Response Mapping

An XML Response mapping is a child definition to the XML via HTTP step definition, defined to extract data returned by the parent step and map it to the property values or other data members of the application. Data returned may be extracted by the mapping when that takes the form of a Cookie, HTTP Header, or XML. The value extracted may be assigned to the properties of the definition for which the step was defined, or to one of several other data values within the application.

The response mapping defines both the source type, or "Mapping Type," of the data returned by the step definition, and the data component of the mobile application where the value is stored. If the mapping type is an XPath Expression or XSL Transformation, the return data must be structured XML. Included in the response mapping then is the XPath or XSL to extract the data from the XML document received by the step after its request was submitted.

Once the data is extracted by the response mapping definition it is assigned to the data component of the mobile application as specified within the mapping definition. This can include a property within the definition for which the parent step was defined, as well as messaging, user ID, the creation of local and local XML data tags, the parent object's key property, or the value may be used for validation.

A given XML via HTTP step can contain multiple response mappings. Each will extract data from the same data set returned by the step to the Agentry Server. The parent step definition's property view contains a list of all response mappings. The order in which the mappings are processed is the position in which the mappings are listed in this view. This order can be modified by moving the mappings up or down within the list.

Response Mapping Attributes

- **Mapping Type:** This attribute specifies the type of data from which the value will be extracted for the response mapping. This may be one of Cookie, HTTP Header, XPath Expression, or XSL transformation. If XPath or XSL is selected, the return set from the step is assumed to be an XML document. This requires the definition of the Base XPath and XPath, or XSL attributes to specify which components of the XML document are to be extracted.

- **Name:** Contains the unique internal name for the response mapping. This value must be unique among all response mappings within the same parent step. The label for this attribute field in the Editor will change based on the selected Mapping Type.
- **Base XPath:** This attribute is enabled when the Mapping Type attribute is set to either XPath Expression or XSL Transformation. The Base XPath is set to locate an element within the XML Document that contains one or more child elements of the same type. The XPath attribute then specifies the specific child element type within the element specified by the Base XPath. The response mapping will iterate over all instances of the child element, extracting the value of each and assigning to the value specified in the mapping attributes. This is most commonly used when synchronizing data object instances within a collection property.
- **XPath:** This attribute contains the XPath expression for the specific element within the XML Document whose contents are to be extracted by the response mapping. This expression is used in combination with the Base XPath (if specified) to provide iterative processing of multiple instances of the same element within the same parent element within the document.
- **XSL:**
- **Maps To:**
  - *Error Message:* This selection will map the data to error text display by the mobile application.
  - *Last Update:* This selection specifies the extracted value is a date and time indicating when the data table's source in the back end system was last modified. This value is compared against the internal last update value for the data table as provided by the Client.
  - *Local String (`<<local>>`):* This selection will create a local data tag available to subsequent data mappings in the same parent HTTP Request. The value of this data tag will the value extracted by the response mapping. When selected, the attribute String Name will be available to name the local data tag created. This is the equivalent to calling the SDML function tag `<<local ...>>`.
  - *Local XML (`<<localXML>>`):* This selection will create a local XML data tag available to subsequent data mappings in the same parent HTTP Request. The value of this data tag will be the value extracted by the response mapping.
  - *Notification [Cancel] Button Label:* This selection specifies that the value extracted by the response mapping should be used to label the cancel button displayed in the Notification Dialog displayed by push definitions. If the step is used in any other manner than push processing, this selection will have no effect.
  - *Notification [OK] Button Label:* This selection specifies that the value extracted by the response mapping should be used to label the OK button displayed in the Notification Dialog displayed by push definitions. If the step is used in any other manner than for push processing, this selection will have no effect.

  - *Notification Text:* This selection specifies that the value extracted by the response mapping should be used as the message text displayed in the Notification Dialog displayed by push definitions. If the step is used in any other manner than for push processing, this selection will have no effect.
  - *Notification Title:* This selection specifies that the value extracted by the response mapping should be used as the title bar text of the Notification Dialog displayed by push definitions. If the step is used in any other manner than for push processing, this selecting will have no effect.
  - *Parent Object Key Property:* This selection specifies that value extracted by the response mapping should be matched with the key property of the parent object to the object being synchronized. This is primarily used in fetches and object read steps, where the data for a nested collection is being retrieved. When this option is selected, the **Key Property** attribute will be enabled to allow for the selection of the parent key property to match with the value.
  - *Property Path:* This selection specifies that value extracted by the mapping is assigned to a property within the object definition for which the step has been defined. This selection has no meaning for steps defined for fetch definitions. For steps defined for transactions this option is selected when the parent step is expected to return the key property of the object targeted by the transaction. This is only the case when the transaction step usage definition defines a client response of Update Client Key Property.
  - *User ID:* This selection will map the value extracted by the response mapping to the user's ID. This value is the equivalent to the SDML data tag `<<user.id>>`. If a previous response mapping in any HTTP Request processed by the Server set the user ID, setting it here will override that value. This will then be the value available for all HTTP-XML system connection processing where the data tag `<<user.id>>` is referenced.
  - *Validation:* This selection allows for validation within of the user during the request made by the step. When the validation item is selected, the XPath defined for the step must successfully locate and XML element. The failure to locate the element is treated as failed validation.
- **String Name:** This attribute is enabled when the **Maps To** attribute is set to "Local String (`<<local>>`)." This attribute contains the name of the local data tag to be created. This name can be set to any character consisting of alphanumeric characters.
- **XML Name:** This attribute is enabled when the **Maps To** attribute is set to "Local XML (`<<localXML>>`)." This attribute contains the name of the local XML data tag to be created. This name can set to any string consisting of alphanumeric characters.
- **Key Property:** This attribute is enabled when the **Maps To** attribute is set to "Parent Object Key Property." The **Key Property** attribute is set to the key property of an object that is an ancestor to the object being synchronized.
- **Property:** This attribute is enabled when the **Maps To** attribute is set to "Property Path." The **Property** attribute is set to the property whose value will be set to the one extracted by

the parent step definition. The property selected here should be defined within the definition for which the parent step was defined

Step Type: File Command Line Step

A File Command Line step is defined for a File System Connection and defines a command to be executed by the Agentry Server on the host system of the Server. This command is contained in a script file, with a default file extension of `.bat`, which is the Windows batch file extension. This extension may be changed to match the script language used in the file. The command executed by the Agentry Server can be monitored for its return value. The script file is processed by the Server to expand any SDML tags it may contain prior to execution against the host system.

The contents of the script file for this step type may be accessed directly on the Agentry Development Server for the application project. The file path listed for this file is relative to the installation location of this Server. If this file is modified it is not necessary to publish the application project for the change to be exhibited. In a production environment this file is not directly accessible in this manner and must be modified through the Agentry Editor. Changes made in a production environment must be published.

As an alternative to storing the command in an external file, it may be contained in the Command attribute of the step. Such a command must consist of a single line. By default the Command attribute is set to the SDML tag `<<script>>`, which expands at run time to the file referenced in the steps File attribute.

A file command step can be defined to wait for the command it calls to complete execution. When defined in this manner, the back end synchronization for a user will not continue until the command returns, or until the defined wait period expires. If the wait period is exceeded, the Agentry Server will log an error and the synchronization will be halted.

If the step is not defined to way for the command to complete, an error will only be logged if the defined command cannot be executed by the Server for any reason.

The script file or the text in the Command attribute for this step is processed by the Agentry Server, which runs it through the Server's SDML pre-processor before executing the step. The results of this SDML expansion are written to a temporary directory, based on the Server's configuration.

File Command Line Step Attributes

- **Name:** Contains the unique internal name of the step definition. This value must be unique among all step definitions within the same module.
- **Connection:** This attribute references the system connection for which the step is defined. This attribute is set when the step is initially created and cannot be modified. For a File Command Line step this must be a File system connection type.
- **File:** This attribute contains the path and file name of the script file containing the step's commands. This path is relative to the path

```
ServerDirectory\Application\Development\Scripts
```

where `ServerDirectory` is the installation location of the Agentry Development Server.

- **Command:** This attribute is set the SDML data tag <<script>> by default. This tag expands to the script referenced in the File attribute. If the command does not contain this data tag, its contents are assumed to be the command to be executed by the Server. In this case the command must be a single line, which may contain SDML tags.

Wait Attributes

- **Wait:** This attribute specifies whether or not the Agentry Server should wait for the command executed by the step to complete before processing the next step in the synchronization process. When set to true, the Server will wait for the duration of time specified in the **Wait Period Limit** attribute. If the command does not complete within this limit, the Server will attempt to kill the command process. It will then log an error message and halt synchronization.
- **Wait Period Limit:** This attribute specifies the duration of time the Agentry Server is to wait for the command executed by the step to complete. This attribute is available only when the **Wait** attribute is set to true.
- **Delete Script File:** This attribute specifies whether or not the script file created by the Agentry Server as result of processing the script file for SDML expansion should be deleted or kept. This attribute is available on when the **Wait** attribute is set to true.

Step Type: File Document Management Step

A File Document Management step is defined for a File System Connection and defines a command to be executed by the Agentry Server on the host system of the Server in support of transferring files between the Server and the Agentry Client. The command for this step is stored in a text file executed as a script by the Server. This step type also includes a child definition to encapsulate mappings between the file data and the data members of the mobile application. A File Document Management step is defined for a specific definition type within the same module.

A file document management step can define a command to be executed to retrieve a file from a file system or version control system so that it may be transferred to the Agentry Client. The child definition document mapping can then associate this file with an object property, normally of type External Data. It may also define a command that moves a file referenced by an External Data property within a transaction to a permanent location on the file system or version control system. The file is also associated with the property via the document mapping child definition.

The definition for which the step is defined may be an object, transaction, or fetch. The property referencing the file to be transferred should be a child property of the selected definition.

A component of this definition is the Document Management Script. This script contains the command or commands the Agentry Server will execute in support of the file transfer behavior. This script is by default a Windows batch script (.bat). The file extension for the script may be changed to reflect the type of script language it contains.

The document management step can be defined to wait for the command it executes to return, or it can execute the command without waiting. If defined to wait for the command, the next step to be processed in the synchronization will not be run until the command has completed execution, or until a defined wait period has been exceeded. If the wait period is exceeded, the Agentry Server will log an error and synchronization will stop.

For downstream synchronization, i.e. fetch, push, or object read step processing, the command is expected to product a file to be transferred to the Agentry Client. For upstream synchronization, i.e. transaction processing, the command is expected to process the file after it has been transferred from the Agentry Client to the Server's host system. This may include moving it to another location on the file system, or checking it in or updating it to a version control system, or any other post-transfer processing that should occur for the file.

In addition to the file itself, it is also possible to capture values from the document management command run by the step. This is behavior is also defined in the child definition document mapping. Return values, error codes, and similar data can be assigned to properties of the appropriate data type.

The contents of the script file for this step type may be accessed directly on the Agentry Development Server for the application project. The file path listed for this file is relative to the installation location of this Server. If this file is modified it is not necessary to publish the application project for the change to be exhibited. In a production environment this file is not directly accessible in this manner and must be modified through the Agentry Editor. Changes made in a production environment must be published.

File Document Management Step Child Definitions

Document Mapping: A document mapping definition is a child to a file document management step and defines the correlation between the file produced by that step to a property definition, normally of type External Data.

File Document Management Step Attributes

- **Name:** Contains the unique internal name of the step definition. This value must be unique among all step definitions within the same module.
- **Connection:** This attribute references the system connection for which the step is defined. This attribute is set when the step is initially created and cannot be modified. For a File Document Management step this must be a File system connection type.

Wait Attributes

- **Wait:** This attribute specifies whether or not the Agentry Server should wait for the command executed by the step to complete before processing the next step in the synchronization process. When set to true, the Server will wait for the duration of time specified in the **Wait Period Limit** attribute. If the command does not complete within this limit, the Server will attempt to kill the command process. It will then log an error message and halt synchronization.

- **Wait Period Limit:** This attribute specifies the duration of time the Agentry Server is to wait for the command executed by the step to complete. This attribute is available only when the **Wait** attribute is set to true.
- **Delete Script File:** This attribute specifies whether or not the script file created by the Agentry Server as a result of processing the script file for SDML expansion should be deleted or kept. This attribute is available only when the **Wait** attribute is set to true.

Document Management Script Attributes

- **File:** This attribute contains the path and file name of the script file containing the step's commands. This is relative to the path

```
ServerDirectory\Application\Development\Scripts
```

  where `ServerDirectory` is the installation location of the Agentry Development Server.
- **Command:** This attribute is set to the SDML data tag `<<script>>` by default. This tag expands to the script referenced in the File attribute. If the command does not contain this data tag, its contents are assumed to be the command to be executed by the Server. In this case the command must be a single line, which may contain SDML tags.

Document Mapping

A document mapping definition is a child to a file document management step and defines the correlation between the file produced by that step to a property definition, normally of type External Data. The specific behavior of a document mapping differs depending on the type of definition for which the parent step was defined. For objects and fetches, the document mapping defines where and how to access the file produced by the parent step's command. For a transaction, the document mapping defines how and where the file should be provided to the step's command.

Because of the differences between a document mapping for an object and fetch, and one for a transaction, there are different attributes for this definition type depending the how the parent step has been defined.

When a document mapping is defined within a file document management step for an object or fetch, the purpose of the document mapping is to capture output from the parent step definitions document script and map it to a property within the object or fetch. This output is primarily intended to be a file that is mapped to an external data property. This file will be transferred down to the Agentry Client. Other outputs may be captured from the document script, including output written to standard out and standard error, as well as the command's exit code as returned to the operating system.

When a document mapping is defined within a file document management step for a transaction, the purpose of the document mapping is to provide the contents of a property to the parent step definition's document script. This input to the command may be provided from an external data property and passed to the command by either writing the file to the file system, or by piping it to the commands standard input. When piped to standard input, the option exists to pass the EOF character to that command after all file data has been passed in. When writing the file to the file system, the command is then expected to look for the file at that location and process it accordingly. When the command has completed processing the file, the option exists to delete the file from the file system. Note that this option will not be available if the parent document management step has been defined to not wait for the document script to complete execution.

Document Mapping Attributes - Object and Fetch

- **Property:** This attributes specifies the property to which the output from the parent step's command will be mapped. For a file produced by the command this should be a property of type External Data. For other output types, the proper data type of the property will vary.
- **Output Type:** This attribute specifies which output from the command to map to the selected property. The options to this attribute are:
  - *Command Exit Code:* This selection specifies that the exit code returned by the command to the operating system should be captured and mapped to the selected property.
  - *File Created By Command:* This selection specifies that a file created by the command should be mapped to the selected property. For this output type the **Property** attribute should be set to a property of type External Data.
  - *STDERR:* This selection specifies that any output written by the command to standard error should be mapped to the selected property. This may be done to determine if an error has occurred, and the nature of that error.
  - *STDOUT:* This selection specifies that any output written by the command to standard out should be mapped to the selected property.
- **File Name:** This attribute is enabled when the selected **Output Type** is "File Created By Command." The **File Name** attribute specifies the name of the file to be mapped to the selected property. This value may include SDML tags, with the default being `<<script>>-1.tmp`.
- **Delete File:** This attribute is enabled when the selected **Output Type** is "File Created By Command." The **Delete File** attribute specifies whether to keep the file created by the command after it has been transferred, or if it should be deleted.

Document Mapping Attributes - Transaction

- **Property:** This attribute specifies the transaction property containing the value to be passed to the document command. If the **Input Type** is "File Input to Command Line," this should be an External Data property.
- **Input Type:** This attribute specifies how the value or file referenced by the selected property will be passed to the document command of the parent step. This can be set to one of the following options:

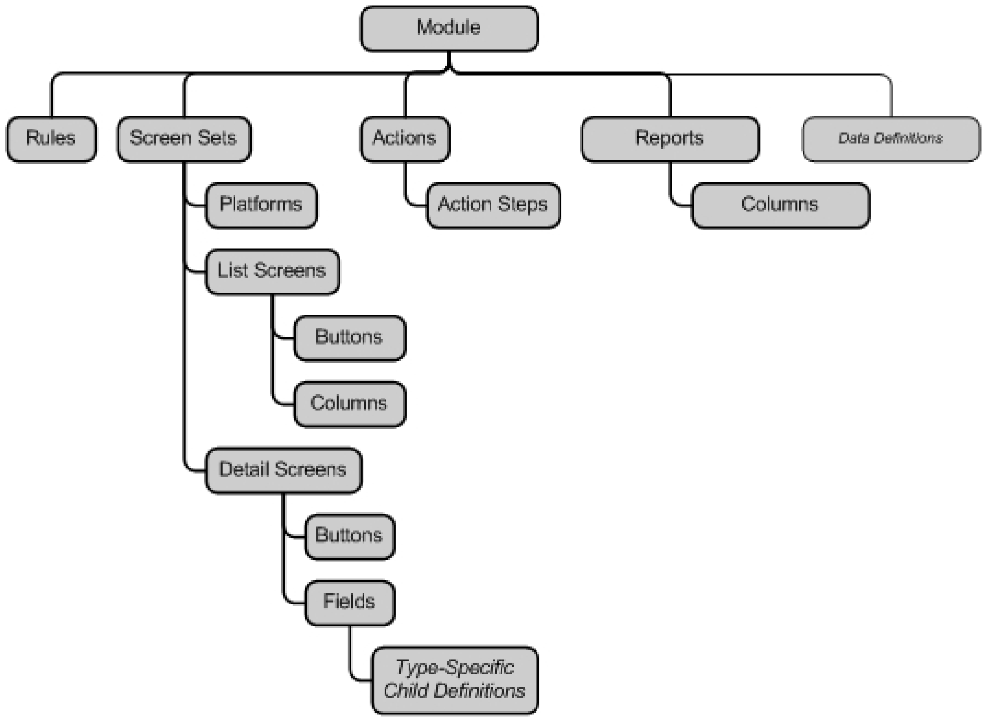
    - *File Input to Command Line:* This selection specifies that file referenced by the selected property should be written to the file system and that the command will the read it in from that location. The **File Name** attribute is enabled when this option is selected, and specifies the file name to which the file will be saved.
    - *STDIN:* This selection specifies that the value of the selected property should be piped to the document command through standard input. For external data properties the file data will be streamed directly to the command without being written to the file system. When this selection is made the **Send EOF** attribute is enabled, indicating whether the EOF character should be sent to the command after the property data as been piped to the command.
- **File Name:** This attribute is enabled when the **Input Type** attribute is set to “File Input to Command Line.” This attribute contains the name to be given to the file when it is saved to the file system. This value may include SDML tags. It is set to `<<script>>-1.tmp` by default.
- **Delete:** This attribute is enabled when the **Input Type** attribute is set to “File Input to Command Line.” The **Delete** attributes specifies whether the file saved to the file system by the Server should be deleted after the document command has finished processing it.
- **Send EOF:** This attribute is enabled when the **Input Type** attribute is set to “STDIN.” This attribute specifies whether or not to send an End of File character to the document command at the end of the file data. This is provided in support of those processes that require this character to indicate no further input is being sent.

Module-Level User Interface Definitions Overview

Within the module level of the application project in Agentry there are definitions for both data and user interface encapsulation. The user interface definitions encapsulate the screens and behaviors that expose the functionality within the application to the mobile users on the Agentry Client. These definitions do not have any direct impact on the behavior or functionality of the application as defined for the Agentry Server.

Of the user interface definitions, those that encompass the screens displayed on the Client are the most robust. The structure of these definition types is deeper than any of the other module-level definitions within the application.

Following is a graphic illustrating the module-level user interface definitions and their child and descendent definition types. This includes definitions that encapsulate the screens and screen controls displayed on the Agentry Client, the behaviors and functionality, and other similar user interface-related application components. Excluded from this graphic are the data definitions within the module. Note that this separation is for discussion purposes only. Within the application project structure, all child definitions to the module exist at the module level with no distinction made between them in the Agentry Editor in relation to whether they are data or user interface definitions.



The rule definition type within the module is actually one that crosses the line between a user interface and data definition. It is organized here with the user interface definition types, as a large portion of the rules written for a module affect this aspect of the behavior. However, rules can also be written and used within data definition types. The rule definition is described in this reference manual in its own section.

As indicated in this graphic, the screen set definition type is a deep structure, with several levels of child definitions below it. Note that, while separated in the above graphic, the list screen and detail screen items are both the same definition type, screen. A screen is a child definition to the screen set and, when defined, is either a list or detail screen. Each screen type has distinct child definitions, and thus are separated in the graphic shown here.

The field child definition to the detail screen can have child definitions of its own. This is dependent on the type of field defined, or the field’s “edit type.” The edit type of a field impacts the fields appearance and behavior on the Agentry Client. Certain field edit types include child definitions that support their intended behaviors. Field edit types are discussed individually within this section of the manual and those that include child definitions are noted.

Overall the user interface definitions within the module display, expose, and provide the means to capture data to and from the mobile users. User interface definitions can display not only data from the module, but also data stored in the application level definitions data table and complex table.

User Interface Definition Types

The definition types within Agentry that define the Agentry Client’s user interface are the screen set, platform, and screen.

- **Screen Set:** The screen set is the main Client user interface definition and defines what definition type its child screens display.
- **Platform:** The platform definition defines how the screens it uses within the same screen set appear on a specific device type.
- **Screen:** A screen definition defines how the property values in the definition being displayed are presented to the user on the Agentry Client. There are two possible screen types that may be defined, list screens and detail screens. Screen definitions have additional child definitions for the controls they display. These child definitions are dependent on the type of screen (list or detail) and the definition type displayed by the parent screen set.

Each of these definition types provide a separate portion of the UI functionality to the application and are broken out into these separate, but related definitions primarily to provide the separation of data and interface. This separation allows for the multi-device support by a single Agentry application. The overall structure of the definition hierarchy within Agentry, and the UI definitions’ place within it, allows the business logic of an application to be separate from the UI. This also allows the UI to be defined to take full advantage of the capabilities of each device type.

Screen Set

A screen set definition defines the Agentry Client's user interface. The screen set defines the definition type to be displayed, which can be an object, transaction, or fetch within the same module. The properties of this definition type can then be displayed by the screen definitions within the screen set. Screen sets contain the child definitions screen and platform.

The type of data definition a screen set is defined to display will have an effect on the types of screens it may contain and how those screens are presented on the Agentry Client. When a screen set is defined to display an object it may contain both detail and list screens. Each screen within the screen set is displayed within the same window, with the screens represented by tab controls. In most cases the fields displayed on these screens are read-only.

When the screen set is defined to display a transaction or fetch it can only contain detail screens. These screens are displayed in a wizard format, with each screen displayed one at a time and containing navigation buttons to advance, reverse, cancel, or complete the wizard. Note that this navigation will also be affected by the action that displays the screen set. The fields of these screens can be read-only or editable based on each field’s definition.

When a new module is added to an application project a single screen set will be defined within it automatically. This will be the main screen set for the module, making it the first screen set displayed on the Agentry Client when that module is viewed by the user. There is only one main screen set per module. This screen set definition can be altered but cannot be deleted.

Screen Set Child Definitions

The following definitions are child definitions to the screen set:

- **Platform:** The platform definition defines how the screens it uses within the same screen set appear on a specific device type.
- **Screen:** A screen definition defines how the property values in the definition being displayed are presented to the user on the Agentry Client.

Screen Set Attributes

- **Displays:** This is a two part attribute consisting of the definition type and the specific definition of that type the screen set will display. Screen sets can be defined to display objects, transactions, or fetches. The selection made here makes the data (properties) within that definition available to the screens defined within the screen set.
- **Name:** This is the unique internal name of the screen set that identifies the definition within the module. This value must be unique among all other screen sets in the same module and can contain no white space.
- **Main Screen Set:** This attribute cannot be set by the developer and is displayed in the add screen set wizard and properties screen in the Agentry Editor for reference purposes only. The main screen set for a module is created automatically by the Editor whenever a new module is defined.

Platform

A platform definition defines how a screen set's screens will appear on a specific device type. A platform is defined to use one or more screens within the same parent screen set. There are different platform types, each corresponding to a different type of client device. The platform affects the placement of buttons and the form factor of the screens it uses.

The most important attribute to the platform definition is the Platform Type. This attribute specifies the platform upon which the screens it uses will be displayed and how those screens will appear. A given screen set can contain one or more platform definitions. At least one platform must be defined before screens can be added to the screen set. During publish, at least one screen must be used by at least one platform within the screen set or an error will be returned and the publish will not be allowed to proceed.

A platform can use more than one screen within the same screen set. A screen can be used by more than one platform as well. At run time, when a screen set definition is sent to a client, the client's device type will determine which screens that client receives based on the platform using the screens.

Platform Attributes

General Attributes

- **Platform Type:** This is the type of device platform to be supported by the screens used by the platform and will affect the form factor and behavior of those screens.
- **Caption:** This is the title text displayed in the window on the Client at run time for the screen set. Since this is at the platform level, the screen set's window can contain a different caption on different target devices. This value may be set statically or via a rule definition for more dynamic text. A rule used here is expected to return a string value and is evaluated in the context of the object displayed by the screen set.
- **Size:** This attribute only applies to platform definitions for the Windows desktop, laptop, and tablet operating systems. For this type of platform the Size attribute specifies the initial display size of the screens it uses. For other **Platform Types** this attribute is disabled and all screens used by the platform are displayed in the full screen size of the device type. Note that this attribute may be affected or negated by the application definition's **Screen Size** attribute.
- **Button Placement:** This attribute contains four possible settings: Bottom, Top, Left, and Right. This attribute specifies where the buttons for all screens used by the platform are displayed.
- **List Navigation:** This attribute controls whether or not the object displayed by the screens used by the platform definition can be changed via navigation buttons drawn automatically on the Client. When true, these buttons will allow a user to change the object displayed in the current screen set based on a list of objects in the previous screen in the navigational flow. In this previous list, the previous or next item in the list is selected and the action executed to display the current screen set is executed again. This attribute has no effect on platforms for the module main screen set or for platforms within screen sets displaying a transaction or fetch. This behavior is applicable when the previous screen was a list screen, or when it was a detail screen containing a list view or list tile view field.
- **Screen Navigation:** By default a screen sets screens are displayed as tabs on the Client at run time. Selecting this option removes the tabs and instead displays a menu button containing the caption value of each screen definition to allow the user to select different screens.

Platform Screen Type

The following attributes are only valid for platforms of type iPad or Android and support the display of a pop up screen using the platform's screens.

- **Screen Types:** The options for this attribute include **Full Screen** and **Overlay View**. Full Screen creates standard screens on the Client. Overlay View creates popup screens that overlay the screen from which the user navigated. These screens can be used for both read only information as well as for data capture. When Overlay View is selected, the Height and Width attributes are enabled to specify the size of the overlay screen displayed.
- **Height:** The vertical size of the overlay screen in pixels.
- **Width:** The horizontal size of the overlay screen in pixels.

Platform Style Attributes

The style attributes of a platform specify the styles applied to different aspects of the screens used by the platform. Style definitions must exist before these attributes can be set. The final appearance of the screen will be affected by the overall application of styles according to the style hierarchy. There are three groups of style elements for the platform: Screen Styles, Detail Screen Styles, and List Screen Styles. Screen styles affect all screens used by the platform regardless of screen type. Detail screen and list screen styles affect only those screens of the corresponding type.

All style attributes for the platform definition may be set statically by selecting the style from a list, or by returning the name of a style to apply from a rule definition. Rules evaluated for style attributes are expected to return a string value containing the name of the style to apply and are evaluated in the context of the object displayed by the parent screen set.

Screen Styles

- **Tabs:** The style to apply to the tab controls representing each screen within an object screen set. Has no affect on screens within a transaction or fetch screen set.
- **Buttons:** The style to apply to all button definitions for screens used by the platform.
- **Focused Buttons:** The style to apply to the button that currently has the focus.

Detail Screen Styles

- **Screen:** The style to apply to the screen as a whole. This will affect all portions of the screen not displaying a field or button.
- **Fields:** The style to apply to all fields displayed on the screen.
- **Focused Fields:** The style to apply to the field that currently has the focus.
- **Hyperlinks:** The style to field labels defined to be hyperlinks.

List Screen Styles

- **Screen:** The style to the list screen as a whole. This will affect all portions of the screen not displaying a list, header label, detail pane, or button.
- **Header Label:** The style to apply to the list screen’s header label. If no header label is defined this attribute has no affect on the screen.
- **Column Labels:** The style to apply to the column labels on the screen’s list control.
- **Rows:** The style to apply to all rows on the list screen. The Hyperlinks optional style will override the Rows style for cells with hyperlinks.
- **Alternate Rows:** The style to apply to every other row in the list, beginning with the second row. The Hyperlinks optional style will override the Alternate Rows style for every other row, specifically cells containing hyperlinks within the row.
- **Highlight Rows:** The style to apply to a row for the purposes of drawing attention to that row. This style should always be returned via a rule definition that evaluates the object being listed. The optional Hyperlinks style will be applied to the highlighted row’s cells containing a hyperlink.
- **Selected Rows:** The style to apply to the row currently selected by the user in the list control. The optional Hyperlink style will be applied to any cells within the selected row containing a hyperlink.

- **Detail Pane:** The style to apply to both the foreground (text) and background of the list screen’s detail pane. If no detail pane is defined this attribute has no affect on the screen.

Platform Button Attributes

Platforms defined within a screen set displaying a transaction or fetch include an additional set of attributes related to the buttons displayed on screens used by the platform. Screens within this type of screen do not contain button definitions, but rather contain buttons added to each screen automatically by the Client based on the action that displayed the parent screen set and the position of each screen within the screen set.

- **Cancel Action Details:** The label for cancel buttons that will cancel the action currently being executed.
- **Previous Screen Details:** The label for buttons that allow users to navigate to the previous screen in the current screen set.
- **Previous Record Details:** The label for buttons that allow users to navigate to the previous transaction instance. This button is displayed on the first screen of a screen set when being displayed by an action with looping behavior.
- **Next Screen Details:** The label for buttons that allow users to navigate to the next screen in the current screen set.
- **Next Screen (no back up) Details:** The label for buttons that allow users to complete the current instance of a wizard in a loop and start the next iteration; or to move from one wizard to the next when multiple wizards are displayed by the action.
- **Complete Action Details:** The label for buttons displayed on the last screen of a screen set, when there are no additional screen sets displayed by the action and when the current screen set is not being displayed in a loop.
- **Complete Action Details:** The label for buttons displayed on the last screen of screen set being displayed in a loop and that will end that loop.

Platform Screens List

The Properties view for a platform definition within the Agentry Editor includes a Screens tab. This tab lists all screens within the same parent screen set of the platform. Within this list the screens to be used by the platform can be selected. The screens listed here are not child definitions to the platform, but rather a children of the screen set. If a new screen is added to the screen set by starting the Add Screen Wizard from the platform view, that screen will automatically be used by that platform.

List Screen

A list screen definition displays an object collection property on the Agentry Client. Object instances from the collection are displayed as rows in the list. A list screen contains the child definitions column and button. A column is defined to display the property value for each object instance in the collection. Buttons are defined to execute actions related to the object instances. List screens include definable behaviors related to filtering, scanning, and sorting, as well as other screen enhancements for displaying data stored in the object instances of the target collection property.

The list screen may or may not display a header label above the list control. A header label can contain static or dynamic text about the items displayed in the list. A list screen may also display a detail pane containing static or dynamic text. The detail panes intended usage is to display the property values of the currently selected object in the list control, reducing the need for horizontal scrolling on the Agentry Client.

List screens can be defined to include double-click actions, executed when the user double-clicks an item in the list control, scanning actions and scan filtering, and include rules to determine what items are displayed in the list. A list screen can also be enabled or disabled via a rule definition. Disabled screens are not displayed in the screen set on the Agentry Client.

List Screen Child Definitions

- **Column Definition:** A list screen column defines what object property is displayed for each record in a list control and how it is formatted on the screen.
- **Button Definition:** A button definition defines a button control to be displayed for the screen that will execute an action or display a menu when selected.

List Screen General Attributes

The General Screen attributes set the basic behavior of the List Screen, including how Styles can be applied to the List Screen.

General Attributes

- **Name:** The internal name of the list screen. This value must be unique among all screen definitions, regardless of type, within the same parent screen set.
- **Caption:** Labels the tab on the Agentry Client for the list screen. This value may or may not be displayed when there is only one screen displayed within the parent screen set, depending on the client device type.
- **Screen Icon:** This is a reference to an image definition within the application. This image is used as the icon displayed for this screen in tabs.
- **Collection:** References the object collection property the list screen is to display. This collection is normally a property of the object definition the parent screen set is defined to display.
- **Enable Rule:** References a rule definition expected to return a Boolean value and that is evaluated in the context of the object definition for the parent screen set. When false is returned, the screen will be disabled and no tab for it will be displayed within the screen set window. If all screens within a screen set are disabled, that screen set will not be displayed and any actions defined to display it will also be disabled. If the main screen set for a module is disabled, that module cannot be displayed on the Agentry Client.
- **Include Rule:** References a Rule definition expected to return a Boolean value and that is evaluated once for and in the context of each object in the collection displayed by the list screen. When an include rule is specified, only those objects for which the rule evaluates to true will be listed in the screen's list control.
- **Icons Image:** References an image definition to be displayed on the tab for the list screen, to the left of the screen's caption text, within the screen set window on the Agentry Client.

The name of this image may be selected from a list, or it may be returned from a rule. When a rule is referenced, it is expected to return a string value and is evaluated in the context of the object displayed by the parent screen set.

List Screen Styles

- **Screen:** The style to apply to the list screen as a whole. This will affect all portions of the screen not displaying a list, header label, detail pane, or button.
- **Header Label:** The style to apply to the list screen's header label. If no header label is defined this attribute has no affect on the screen.
- **Column Labels:** The style to apply to the column labels on the screen's list control.
- **Rows:** The style to apply to all rows on the list screen. The Hyperlinks optional style will override the Rows style for cells with hyperlinks.
- **Alternate Rows:** The style to apply to every other row in the list, beginning with the second row. The Hyperlinks optional style will override the Alternate Rows style for every other row, specifically cells containing hyperlinks within the row.
- **Highlight Rows:** The style to apply to a row for the purposes of drawing attention to that row. This style should always be returned via a rule definition that evaluates the object being listed. The optional Hyperlinks style will be applied to the highlighted row's cells containing a hyperlink.
- **Selected Rows:** The style to apply to the row currently selected by the user in the list control. The optional Hyperlink style will be applied to any cells within the selected row containing a hyperlink.
- **Selected No Focus Rows:** The style to apply to the selected rows in a list view control or list screen where the list control does not have the input focus. The optional Hyperlinks style will be applied to any cells within the selected row containing a hyperlink.
- **Detail Pane:** The style to apply to both the foreground (text) and background of the list screen's detail pane. If no detail pane is defined this attribute has no affect on the screen.
- **Buttons:** The style to apply to all button definitions on the screen.
- **Focused Buttons:** The style to apply to the button that currently has the focus.

Actions/Sorting Attributes

The Action/View/Selection attributes control how the user interacts with the List Screen, including double-clicking on or off an item in the list and behaviors related to sorting and reordering the columns.

- **Double-Click On Item - Action:** Specifies the action to execute when the user double-clicks a list control record.
- **Double-Click On Item - Target:** Specifies the target of the Double-Click On Item Action. A target must always be specified for the action and is typically the selected object in the list.
- **Double-Click Off Item - Action:** Specifies an action to be executed when the user double-clicks the list without clicking on an item. This is most commonly used to execute an action that instantiates an add transaction for the object type being listed.

- **Double-Click Off Item - Target:** Specifies the target of the Double-Click off Item Action. A target must always be specified for the action. Typically the target is the parent object of the object collection property displayed by the list screen.
- **Fixed Sort Property:** Specifies the property definition within the object type being listed used to sort the records in the list. The user will not be allowed to re-sort the list when this attribute is set. The Order option to this attribute is set to specify the sort order, either ascending or descending.
- **Allow Sort:** Specifies if the user can resort the list by clicking on a column header. This is enabled by default, and is disabled if a **Fixed Sort Property** is set.
- **Initial Sort Column:** Specifies a column definition by which the list will be sorted upon initial display of the screen. This attribute requires that a column definition exists before it can be set. The Order option to this attribute is set to specify the sort order, either ascending or descending. If the list screen allows the list to be sorted (**Allow Sort** is true) the list will be displayed sorted to the order of the last sort action. If a **Fixed Sort Property** is set, this attribute is disabled.
- **Allow Reorder:** Specifies whether or not the user can reorder the columns displayed in the list by dragging and dropping the column headers in the list. This is enabled by default.
- **Allow Filter:** Specifies whether or not the user can filter the items in the list. A filter icon is displayed at the bottom of the screen when enabled. The user can click this icon to select sorting options. This sets the filter behavior for the entire list screen. This is enabled by default. Individual column definitions may be defined to prohibit filtering on those columns.
- **Allow Multi-Row Select:** Specifies if the user can select more than one record in the list at the same time. If multiple items are selected in a list, actions that target the selected object in the list will be executed once for each selected object. The default for selecting multiple objects requires a `Ctrl+Click` combination or a click and drag operation by the user, depending on the device type. The **Enable Single Click** option may be set to allow multiple records to be selected with a single click by the user. Deselecting a record requires the user to click it again. This feature is normally most useful on touch screen devices using a stylus, as it allows non-sequential records in the list to be selected.

Header/Detail Pane Attributes

Using these attributes, you can display Header text and a Detail Pane in addition to the main list control of the List Screen.

Header and Detail pane attributes are set to display additional information about the list as a whole or about the currently selected item in the list. The Header Label is a static line of text displayed above the list. This text may be static, set via certain available format strings, or set via a rule. A rule referenced for this purpose is expected to return a string value and is evaluated in the context of the object displayed by the parent screen set.

The Detail Pane is redrawn each time a new object is selected in the list and almost always contains either format strings or is set via a rule’s return value. Rules are evaluated in the context of the selected object in the list and are expected to return a string value.

- **Header Label:** Specifies the Header text for the list screen. A common use for this header label is the total number of objects displayed in the list vs. the total number of objects in the collection, which may be different when a filter is enabled. The format strings used for this purpose are `%DisplayedCount` and `%TotalCount`.
- **Detail Pane:** Displays a text box on the list screen. The detail pane is updated each time the user changes their selection in the list screen.
- **Position:** Controls where the detail pane is displayed on the screen in relation to the list control. You can position the detail pane below it or to its right.
- **Size:** Sets the pixel size of the detail pane on the screen. The default is 50. If the **Position** is "Bottom" the detail pane will span the width of the screen and the **Size** will set its height. If the **Position** is "Right" the detail pane will span the height of the screen and the **Size** will set its width.
- **Word Wrap:** When enabled, lines of text longer than the width of the detail pane will be wrapped to the next line. When disabled, text will continue off the screen. The user will need to scroll the detail pane to view the text.
- **Format:** Sets the values displayed in the Detail Pane. This pane can be set to a combination of static text and format strings, which take the form *`%propertyName`*. The *`propertyName`* is the name of a property defined within the selected object and will be updated with the value of that property each time a different object is selected. It may also be set to the return value of a rule, which is evaluated in the context of the selected object instance and is expected to return a string.

Scanner Attributes

The scanner attributes for a list screen affect only those list screens used by a scanner platform within the screen set and only when the list screen is displayed on a client device with a barcode scanner. At least one column definition within the list screen must be defined to support scan filtering.

A scanned value will be compared to the column(s) defined for scan filtering and only those matching this value will then be displayed. Actions may be defined when only one record matches the scan filter and when no records match.

- **Single Match Action:** Specifies what action is executed when a scanned barcode value matches one of the records displayed in the list screen. The target of the action will always be the record found to match.
- **No Match Action:** Specifies what action is executed when the scanner filter criteria does not match any records on the list. This is optional. The target of the action is the object that is the parent to the collection property displayed by the list.
- **Label Type:** Specifies what barcode types are accepted by the Agentry Client. If no Label Type is specified, all types supported by the client device will be supported.
- **Minimum Value:** The minimum number of characters accepted by the Agentry Client from the device scanner.

- **Maximum Value:** The maximum number of characters to be accepted by the Agentry Client from the device scanner. If the value scanned in contains more characters, it will be ignored.

List Screen Column

A column definition defines what object property is displayed in a list control column. The column definition also controls behaviors such as formatting, sorting the list on the column, whether or not the column can be resized or moved, and whether or not the list can be filtered on the column. Columns may also be defined to execute an action via hyperlink control.

In addition to or in place of a property value, a column may also display an image definition as an icon, which can be different for each record based on a Rule definition.

Column Attributes

- **Name:** Internal name for the column definition. This value must be unique among all columns definitions in the list screen.
- **Label:** Specifies the label for the column header. This text is displayed at the top of the column on the Agentry Client to identify the contents of the column.
- **Object Property:** Specifies the property to display in the column on the list screen. Set this to None, to display either a value derived from a format string or only an icon image. Selecting both an Object Property and specifying an icon image will display both in the column.
- **Enable Rule:** References a rule definition evaluated in the context of the object displayed by the screen set and expected to return a Boolean value. When the rule returns true, the column is enabled and displayed on the Agentry Client. When it returns false, the column is disabled and not displayed.
- **Icon Image:** References an Image definition within the application to specify an icon for the column. The image name can also be returned using a rule definition to dynamically determine the image to display for each record. This rule is evaluated in the context of the object instance for the record and is expected to return the name of an image definition as a string. Note that not using a rule for this attribute will display the same image for all records in the list
- **List Filter:** Specifies if the column should be included in those listed in the filter dialog for the list. This attribute is ignored in filtering has been disabled for the list screen.
- **Scanner Filter:** Enables scan filtering functionality for the column. When this attribute is enabled, the value scanned in by the device will be compared to the values of the column to create a filter. Multiple columns can be defined for this behavior. However, the values in the columns should be mutually exclusive. The order of the columns evaluated against the scanned value is undefined. This attribute is only supported for screens used by a scanner platform and displayed on a scanner-enabled device.
- **Format:** Can contain a format string to display one or more property values from the object type being displayed by the list in a different format than the default for the property’s data type. This text can also be set via a rule definition, where the expected

return value is a string and is evaluated in the context of the object instance for the record in the list. To set the format attribute set the **Object Property** attribute must be set to None.

- **Column Width:** Specifies the initial size of the column on the Agentry Client. The user can resize the columns if the list screen definition has not disabled this behavior. If the user changes the width of a column, the new width is saved by the Agentry Client and will override the **Column Width** attribute.
- **Hyperlink:** Specifying a hyperlink action enables each cell within the column to execute an action when the user single or double clicks on the hyperlink drawn in that column. The text of the hyperlink will be the value the column is defined to display. This functionality can include columns with images. Hyperlink contains two attributes:
    - **Hyperlink Action:** Specifies the action that will be executed when a user single-clicks a column in a populated row in the list.
    - **Hyperlink Target:** Specifies the target of the Hyperlink Action.

Detail Screen

A detail screen definition displays a single instance of an object, transaction, or fetch on the Agentry Client. The properties of the definition instance are displayed in fields, a child definition to the detail screen. Definable behaviors of a detail screen are predominantly controlled by the screen’s child field and button definitions, which can include read-only or read-write values within the fields, as well as numerous field type behaviors. Detail screens for transactions and fetches do not have the child definition button.

The detail screen definition contains attributes for the screen's caption, enabling and disabling the screen, and the initial focus of the screen. The detail screen is separated into multiple rows and columns, based on the definition. These row and column positions are used to specify the location of fields on the screen.

The values of the definition instance displayed by the detail screen are exposed to the user via the field definitions.

Detail Screen Child Definitions

- **Detail Screen Fields:** A detail screen field defines field controls for display on a detail screen to display data to and capture data from the Agentry Client user.
- **Buttons:** A button definition defines a button control to be displayed for the screen that will execute an action or display a menu when selected. Detail screens only have button definitions when the parent screen set is defined to display an object.

Detail Screen Attributes

General Settings

- **Name:** Internal name for the screen definition. This value must be unique among all screen definitions within the same parent screen set, regardless of screen type.
- **Caption:** Labels the tab on the Agentry Client for the detail screen when a part of an object screen set. For transaction a fetch screen sets, the detail screen caption text is displayed in

the title bar of the window on the Agentry Client. This value may be set to a rule. This rule is evaluated in the context of the definition instance being displayed, and is expected to return a string value.

- **Screen Icon:** This is a reference to an image definition within the application. This image is used as the icon displayed for this screen in tabs.
- **Enable Rule:** References a rule definition evaluated in the context of the definition displayed by the parent screen set and expected to return a Boolean value. When the return is false, the screen will be disabled and will not be displayed to the user.
- **Rows:** Sets how many rows the screen will contain. This attribute is used to divide the screen into rows, which are referenced by the field definitions to determine the position of each field on the screen. The default settings will vary depending on the platform using the screen. The grid created by the **Rows** and **Columns** attributes is not displayed on the screen at run time, but is visible in the Agentry Editor for development purposes.
- **Columns:** Sets how many columns the screen will contain. This attribute is used to divide the screen into columns, which are referenced by the field definitions to determine the position of each field on the screen. The default settings will vary depending on the platform using the screen. The grid created by the **Rows** and **Columns** attributes is not displayed on the at run time, but is visible in the Agentry Editor for development purposes.s
- **Initial Focus:** Sets the field to be the initial focus when the screen is first displayed on the Agentry Client. This attribute requires that fields have been defined for the detail screen.
- **Label Position:** Specifies the position of the label text for all fields displayed on the detail screen. The options for this attribute are either Left or Top, with the Left being the default.

Detail Screen Style Attributes

- **Screen:** The style to apply to the screen as a whole. This will affect all portions of the screen not displaying a field or button.
- **Fields:** The style to apply to all fields displayed on the screen.
- **Focused Fields:** The style to apply to the field that currently has the focus.
- **Hyperlinks:** The style to apply to any labels define to be hyperlinks.
- **Buttons:** The style to apply to all buttons on the screen.
- **Focused Buttons:** The style to apply to the button that currently has the focus.

Images Attributes

- **Screen Background Image:** This attribute allows for the selection of an image definition within the application which is displayed as the background image for the detail screen. This behavior is currently only supported on detail screens used by iOS and Android platforms.
- **Fit to Screen (Lock Aspect Ratio):** This attribute specifies that the image should be resized to fit within the viewable area of the screen. The aspect ratio of the original image is maintained. This attribute is mutually exclusive from **Fit to Screen (Stretch)** and **Crop to Screen**.

- **Fit to Screen (Stretch):** This attribute specifies that the image is to be resized to fit within the viewable area of the screen. The aspect ratio of the original image is not maintained and the image will always fill the entire viewable area of the screen. This attribute is mutually exclusive from **Fit to Screen (Lock Aspect Ratio)** and **Crop to Screen**.
- **Crop to Screen:** This attribute specifies that the image is to be cropped to fit within the viewable area of the screen. Images larger than the viewable area of the screen will not be fully displayed if this attribute is selected. This attribute is mutually exclusive from **Fit to Screen (Lock Aspect Ratio)** and **Fit to Screen (Stretch)**.
- **Background Image Position:** This attribute specifies the position of the image within the viewable area of the screen. There are nine radio buttons displayed for this attribute, each corresponding to the position of the image on the screen both vertically and horizontally.
- **Field Opacity - Fields Cover Images:** This attribute sets the opacity of the image displayed on the screen. If selected, fields on the detail screen will always be displayed on top of the background image. If not selected, the image will overlay the fields on the screen.

Button

A screen button defines a button control to be displayed on a Client screen. The button may be displayed as a standard button control, a tool bar button, a menu or menu item, or as a separator. A button is defined to execute an action when clicked or tapped, unless defined as a menu or separator. When executing an action the button also defines the target object instance provided to the action for processing.

The button definition itself allows for different Button Types. These include a traditional button, called an Action Button; an item to be added to the Action menu displayed on the Agentry Client’s menu bar, called an Application Menu; a Toolbar Button, which is displayed on the Agentry Client’s tool bar; and a Separator button, which places extra space between other button definitions, or a separator line in a menu.

In addition, an Action Button can be defined to be a Popup Menu button. In this case, the button displayed on the screen will not execute an action, but instead display a menu when clicked. The contents of the menu are other button definitions for the same screen that will execute actions when selected. These other buttons must meet the criteria of, first, being positioned after the menu button, and second, the Popup Menu attribute must be set to true.

All button types except for the separator are defined to execute an action when clicked. The action definition to execute must exist before creating the button definition. Buttons also include a target attribute where the object instance targeted by the action being executed is specified. The selected target object type must be the same as the object type selected for the action being executed, with the exception of those actions defined with a For Object attribute setting of None.

Button Attributes

Following are the attributes for a button definition. Some of these attributes are not applicable to a button definition based on the selection made in the **Button Type** attribute. The attribute descriptions in this list specify this information:

- **Button Type**: This attribute specifies the type of button to define for the screen. The options are:
    - **Action Button:** Displays a button control on the screen at the position specified by the platform using the screen at run time.
    - **Application Menu:** Adds a menu item to the Agentry Client’s Action menu. This menu item will only be a part of the Action menu when the parent screen to the button has the focus.
    - **Separator:** Places extra space between Action Buttons, or a separator in a popup menu, depending on where the separator button is displayed Separators cannot be added to the Agentry Client’s Action menu.
    - **Toolbar Button:** Places a button on the Agentry Client’s toolbar. This button type must have an image as it will not have a label.
- **Name**: The unique name for the button definition. This value must be unique among all buttons within the same screen.
- **Image**: This attribute references an image definition within the application to be used as the icon for the button control displayed on the screen. For action button types the icon is displayed to either the left or right of the button’s label depending on the device’s OS shell. The image icon for toolbar buttons is required. This will be the image used to identify the toolbar button. For both Separator and Application Menu Button Types, or if the Action is set to Popup Menu, the image attribute field is disabled.
- **Label**: This attribute specifies the label to identify the button on the screen. This value is the label for Action Button Types, or the text listed as a menu item for both Action Menu Button Types or Action Buttons included on a popup menu. This attribute is disabled for both Toolbar and Separator Button Types.
- **Action**: This is the action to execute on the Agentry Client when the button is clicked or tapped by the user. This action must be defined before creating the button definition. At run time if this action is disabled, the button will also be disabled. This attribute may also be set to Popup Menu. In this case the button will not execute an action, but rather will display a popup menu when clicked or tapped. The items in this menu will be other button definitions within the same screen defined to be drawn on the popup menu. Popup menu buttons do not have an image or a target object instance. Also, the Popup Menu attribute is not available, as a popup menu button cannot be placed within another popup menu.
- **Target**: This attributes specifies the target object instance of the button to be passed to the action the button executes. The object type selected here must match the definition type defined in the For Object attribute in the action the button is defined to execute. At run time if the selected Target object instance is not currently in scope, the button will be disabled. As an example, if the target is the selected object in a list screen, and no object is currently selected, there is not valid target in scope and the button will be disabled. Separator Button

Types and buttons with an Action attribute setting of Popup Menu do not have a target as they do not execute an action.

- **Popup Menu**: This attribute specifies whether the button should be displayed in a popup menu on the Agentry Client. If this attribute is set to true, and of a button definition positioned before the current definition is defined with Action attribute of Popup Menu, the current button definition will be added as a menu item rather than a button control. This attribute may only be set for Action Button Types.
- **Style**: This attribute specifies a style to apply to the button definition. The Style attribute is only available for Action Button Types.
- **Focused Style**: This attribute specifies a style to apply to the button when the button has the focus. This attribute is not available for Separator Button Types.
- **Shortcut Key**: This attribute specifies whether a shortcut key is associated with the button and the specific key or key combination. This attribute includes the ability to set combinations of the `Ctrl`, `Alt`, and `Shift` keys, as well as any alphanumeric keys, function keys from `F1` through `F24`, or hardware buttons (Button 1 through Button 5) on mobile devices. When setting this attribute, verify the key combination selected is not configured for any other shortcut, either within the current screen of the mobile application or for any system shortcuts configured on the client device.

Detail Screen Fields

A detail screen field defines a field control for display on the parent screen. The field displays data to the user and, when displaying a transaction or a fetch, can capture data from the user. A field can be defined to have one of several edit types that will affect both the appearance and behavior of the field on the screen, especially when capturing data.

There are several different edit types that may be selected for a field definition. This edit type will significantly impact the field’s behavior on the Agentry Client. Despite this, however, there are several attributes that are common among most fields regardless of edit type. For many field edit types these common attributes are the only attributes. For others there are additional attributes specific to the edit type selected for the field definition.

In many use cases a field definition will target a property within the definition it displays. The value of that property will be displayed to the user and, for transaction and fetch screens, the user may be able to edit that value. In these situations, the value of the field will be assigned to the property when the user advances the wizard past that screen. This may be the case when the user clicks a next button or finish button.

The field may also target other definitions within the application. If the target of a field is not a property definition, the value of the field will not be copied to that definition. It will only use it as a data source for the value to display. These targets can be selected using the target browser and can include other fields on the current screen or other screens within the same screen set.

With the release of the Agentry Mobile Platform version 5.1, when the target is another screen field that is one of the edit types for displaying complex tables, it is possible to select a complex table record field from the currently selected record in that target screen field. In previous releases it was necessary to define an update rule for the field that would retrieve the complex table record and field to display in the screen field. The additional target browser behavior negates the need to define such a rule. Existing applications using an update rule for this purpose will still behave correctly, and can be modified to use the new behavior or left as is with the same result.

The target for a screen field can also include a field on screens in other screen sets, provided those screen sets currently exist on the Agentry Client, but are hidden from view due to the focus being on the current screen.

A field on a wizard screen displaying a property value will enforce the data limits of that property. This means minimum and maximum values or string lengths defined for the target property will be enforced by the field definition. For strings, no more than the maximum number of characters may be entered. For numeric values, the target property's attributes related to precision and maximum values will be enforced. For minimum values the user will receive an error message when trying to advance the wizard of either the minimum number of characters or the minimum numeric value has not been entered.

The labels for a field may be defined as static text or as a hyperlink. Hyperlink labels may only be defined for a field displayed on a detail screen that displays an object instance. When a label is defined as a hyperlink, an action is defined to be executed when the user clicks that label.

Fields may have their displayed value set through an update rule. These fields can still target a property, normally for transactions and fetches, in which case the value of the field as set by the update rule will be the value assigned to the property when the user advances the wizard. When displaying an object, there is normally no reason to target a property with a field definition whose value is set via an update rule.

Fields may also be hidden and/or disabled via rule evaluation. A hidden field will not be displayed on the detail screen. An optional behavior related to a hidden field is disabling that field when it is hidden. A field may also be disabled via a separate rule independent or in lieu of a hidden rule. A disabled field on a wizard screen will not enforce any required values as defined by the target property.

Fields are positioned and sized on the detail screen using the columns and rows into which the detail screen is broken up. The position of a field is set based on the upper left corner of the field and is specified using the row and column position. The width of the field is specified in columns, and the height is specified in rows, counting from the position in the field which its placed.

Common Field Attributes

The following attributes are common to most or all field edit types and result in the same behaviors for most of the different types of fields.

- **Object/Transaction Property:** Sets the property definition or other definition whose value is displayed by the field and/or that is updated with the field's value. This definition is said to be “targeted” by the field. This attribute can be set to “-- None --”, in which case the value displayed by the Field must come from some other source.

- **Name:** The unique internal name of the Field definition. It must be unique among all fields within the same detail screen. This is commonly set to match the name of the property the field targets.
- **Label:** Sets the label for the field. This text is displayed on the left side of the field. This label text will be rendered as a hyperlink if that behavior is also defined. This value is optional and if not set no label nor the space for one will be displayed on the screen.
- **Placeholder:** This attribute references a rule definition which returns a string value used as the place holder for the field definition.
- **Edit Type:** Sets the edit type for the field, selected from a list. This may also be set to “-- Default --”, in which case the edit type of the field will match the data type of the property being displayed.
- **Read-only:** Sets the field to be read-only or read/write. Fields targeting an object property are always read-only and are not affected by this attribute. Fields with any other target will respect the **Read-only** attribute setting.
- **Shortcut Key:** Sets a key or key combination that, when entered by the user, will set the focus to the field on the detail screen. This can include both keyboard keys and hardware keys on the client device.
- **Format:** Sets any format text for the value displayed in the field. If using a format string the Object/Transaction Property attribute should be set to “-- None --.”
- **Label Width:** Sets the number of characters that can be displayed in the space given to the label on the Screen. Character size will vary depending on the font used for the label text. The total size of a field on the screen does not change based on the Label Width. The amount of space within the specified size that is given to display the field itself is decreased as the Label Width is increased. Label text longer than the space provided based on the Label Width is word wrapped on the screen.
- **Position - Column & Row:** Sets where the upper-left corner of the field will be displayed. The column and row specified correspond to the number of columns and rows the detail screen is defined to contain.
- **Size - Width & Height:** The Size attributes specify the Width and Height of the field. The Width is set to the number of columns the field should span and the Height is set to the number of rows.

Rules/Hyperlink/Special Value Attributes

- **Change Focus:** Sets if the field will keep the focus of the screen. If this attribute is checked, when focus is set to the field, it will automatically be redirected to the next field on the screen. When unchecked, the field will keep the screen focus until the user selects another control.
- **Update Rule:** References a rule definition evaluated in the context of the definition being displayed and expected to return a string value. This rule is evaluated each time the user interacts with any part of the detail screen. The value returned by the rule is displayed in the

field. Note that this rule will not change the value of the field if it returns the same value for two or more consecutive evaluations.

- **Hidden Rule:** References a rule definition evaluated in the context of the definition being displayed and expected to return a Boolean value. If the rule returns true, the field will be hidden on the detail screen. If false, the field will be displayed.
- **Disable When Hidden:** When checked, the field will be disabled whenever the Hidden Rule returns true. The Enable Rule will not be evaluated. If unchecked, then the Enable Rule will determine whether the field is enabled independently of whether or not the field is hidden.
- **Enable Rule:** References a rule definition evaluated in the context of the definition being displayed and expected to return a Boolean value. When the rule returns true the field is enabled. A false return will disable the field. A disabled field will appear grayed out, and the user will not be able to interact with it. A disabled field will also not update its target property and no attributes related to the required property value will be enforced.
- **Clear When Disabled:** When checked, the field will clear any value in the transaction property it targets if the field is disabled. Disabled fields include are those disabled by the **Enable Rule**; or those with the **Disable When Hidden** attribute is true and the field is hidden by its **Hidden Rule**. This attribute only affects fields with the following edit types:
  - Boolean
  - Date
  - Date and Time
  - Decimal Number
  - Duration
  - Identifier
  - Integral Number
  - String
  - Time
- **Pattern Recognizer:** This attribute enables or disables the behavior of recognizing certain patterns within text values of the field; e.g., e-mail addresses or phone numbers. When set to true, the user can hold down the hyperlink text to invoke some OS-defined operation. Examples may include allowing the user to compose and send an e-mail; or to send a text message or place a call to a phone number. This functionality is available on iOS Agentry Clients version 6.0.6 or later and only when the application is configured in an Agentry Editor version 6.0.8 or later.
- **Hyperlink - Action:** This attribute references an action and when set will enable the hyperlink behavior for the field's label. The label itself will be displayed as hyperlink and the user will be able to click on the label to execute the defined action. This behavior is only valid for fields displayed on an object screen.
- **Hyperlink - Target:** Sets the target object for the Hyperlink Action.
- **Hyperlink - Shortcut Key:** Sets a shortcut key for the hyperlink. When this key combination is entered on the Client, the defined Hyperlink Action is executed.
- **Special Value:** Sets a default value for the field. When a field has a Special Value defined, a radio button is displayed on the detail screen as a part of the field's definition. It is drawn

between the label for the field and the actual field control. A second radio button is also drawn to the immediate left of the field control. When the first radio button is selected, the Special Value defined for the field is set as the field's actual value, which will then update the property targeted by the field. When the second radio button is selected, the field control itself becomes enabled, and the user can enter a value.

- **Display Value:** The value to display in the field when the property value is equal to the field's special value. This only impacts fields on detail screens displaying an object instance.
- **Auto Label and Width:** This attribute can contain a label for the first radio button displayed for special value fields. This label is displayed to the right of the radio button and can indicate to the user that they are selecting the default value.
- **Edit Label and Width:** This attribute can contain a label for the second radio button that enables the field control on the Agentry Client. This label is displayed to the right of the second radio button and can indicate to users that its selection requires them to enter a value.

Detail Screen Field Edit Types

Following are the different field edit types that may be selected from the Edit Type attribute's list. All field edit types include the Common Field Attributes as a part of their definition. Many edit types also include additional attributes related to their edit type-specific behavior. These edit types are denoted as such with an asterisk(*) in this list. Look to the additional information provided for these field edit types for information on their type-specific attributes.

- **Default:** Selecting this edit type option will force the field to take on the edit type matching the data type of the property it targets. If a field has a default edit type and does not target a property, the field will be a string field.
- **Barcode Scan*:** The barcode scan field edit type receives input from a barcode scanner. Use of this field type requires the device to have a barcode scanner, and for the parent detail screen to be used by a platform that supports scanning. This type of field may also accept manual input from the user, depending on how it is defined.
- **Button*:** The button field edit type defines a detail screen field with button behaviors to execute actions and capture values. This field type will draw a button control on the detail screen in any position where a field can be placed. For object screens this button may execute an action. For wizard screens the button can set the value of a property. The type of button displayed may be a push button, check box, or radio button. Check boxes and radio buttons may be grouped (meaning only one can be selected at a time) by all targeting the same property. A value can be defined for this field that will be set to the property the Field targets when the user clicks this button. There are three types of buttons: Radio Button; Check Box; and Push Button. This is the default for displaying a Boolean Property.
- **Calendar View*:** The calendar view field edit type provides an interactive calendar to display an object collection property, with each object treated as a calendar event. The objects in the collection property displayed by this field must include properties for start and end date and times, as well as other calendar related values.

- **Complex Table Drop Down*:** The complex table drop down field edit type displays unique values from a defined record field from a complex table in a drop down list. Using a succession of fields with this edit type can create a cascade. This is a representation of parent-child values where the users will be required to select a parent value first, and then select from only those values that are children of the selected parent. Use of this edit type requires the supporting structure be first defined in the complex table the field displays. A cascade can also be created using a combination of this field edit type and Complex Table List fields.
- **Complex Table List*:** The complex table list field edit type displays the records of a complex table in a list control on the detail screen. Using a succession of fields with this edit type can create a cascade. This is a representation of parent-child values where the users will be required to select a parent value first, and then select from only those values that are children of the selected parent. Use of this edit type requires the supporting structure be first defined in the complex table the field displays. A cascade can also be created using a combination of this field edit type and Complex Table Drop Down fields.
- **Complex Table Search*:** The complex table search field edit type displays the records of a complex table in a searchable list of records. This screen is displayed when the user clicks the associated button for this field type. This is a built-in screen within the Client and will display the records of the complex table in rows and columns. The user may select any index for a string field and enter search text to locate a record within the table.
- **Complex Table Tree*:** The complex table tree field edit type displays the records of a complex table in a tree control, providing a parent-child relationship to the records. This screen is displayed when the user clicks the associated button for this field type. This is a built-in screen within the Client and will display the records of the complex table in a tree control. The records are organized in this tree using the parent-child index relationships defined in the complex table.
- **Data Table Selection*:** The data table selection field edit type lists the records of a data table in a drop down list control on the detail screen. The code value of the record selected by the user is returned to the field. If the number of records is too large to fit in a drop down control, a popup dialog will display the records in a list box.
- **Date: The date field edit type allows the user to enter a date value selected from a calendar control.** The user may also manually type a date value into this field. When using the calendar control, the user clicks the ellipsis button drawn to the right of the field on the detail screen. This will display the calendar where the user can select a date by scrolling through the months. It is recommended that this edit type only be used with properties defined to be date values.
- **Date And Time:** The date and time field edit type allows the user to enter a date and time value selected from calendar and time controls, respectively. In this type of field, the user can enter a date value by selecting it from a calendar and enter a time value in the time portion of the field. It is recommended this field edit type only be used with date and time properties.

- **Decimal Number:** The decimal number field edit type captures decimal values, allowing only numeric values, a single decimal value, and a negative sign. Any other characters will not be accepted by this field type.
- **Duration:** The duration field edit type allows the user to enter a duration value in hours, minutes and seconds. This field displays a control similar to a time entry, but the values entered represent a duration of time, rather than a time of day.
- **Embedded Image*:** The embedded image field edit type displays an image definition on the detail screen that can be interactive. A transparent grid can overlay this image and each section, or “cell” within this grid can have an action associated with it. When a given cell is clicked on the Client that action will be executed. Fields with this edit type have a child definition called Cell that represents each cell in the grid overlaying the image.
- **External Data**: The external data field edit type displays controls to show the Windows File Dialog on the client to allow a file to be selected for an external data property.
- **External Field - Active X Control*:** The external field-ActiveX Control field edit type is defined to call out from a field to an active X control, passing values to the control. Use of this Edit Type also requires use of the Active X interface available with the Agentry Mobile Platform.
- **HTML:** The HTML field edit type supports the formatted display of HTML markup text, or the display of a defined URL for internet navigation.
- **Identifier:** The identifier field edit type requires the user to enter only positive integers. This edit type is intended to support the capture and storage of values intended to uniquely identify some business entity.
- **Image Capture*:** The image capture field edit type provides integration with the client device’s built-in digital camera, allowing for images to be captured and stored in properties of the application.
- **Integral Number:** The integral number field edit type allows the user to enter only whole numeric values and an optional negative sign. Any other characters will not be accepted in this field.
- **Label:** The label field edit type displays only the label portion of a field definition, excluding any actual field control. The Label edit type prevents any editing, and no field is drawn on the detail screen. As will be readily apparent in the Agentry Editor, many of the common field attributes are disabled for fields with an edit type of Label.
- **List Selection*:** The list selection field edit type displays a drop down list of values, the source of which may be an object collection, data table, or complex table. Using an include rule, you can also list a sub-set of the source items. The values listed in this field edit type are treated as a temporary data table that exists only in working memory and only for as long as the field is displayed.
- **List Tile View:** The list tile view field edit type displays an object collection property in a tiled view allowing for add and edit interaction with the collection through the field. This field type will use other screen sets containing detail screens within the same module to display each object in the collection in a list with each object displayed in it’s own tile. This can include different screen sets to display, add, or edit objects within the collection.

- **List View*:** The list view field edit type displays an object collection property in a list control on a detail screen with the same functionality as provided by a list screen. A field of this edit type contains the child definition column, matching the column child definition to the list screen definition.
- **Location:** The location field edit type is intended to display the value of a location property, displaying the latitude and longitude in degrees for the location. This field can be read-only or editable, allowing the user to manually enter latitude and longitude values. When the field targets a transaction property, it will automatically retrieve the latest location value from the GPS unit. When this field edit type displays an object property, or if it does not target any property, a rule can be written to update the field using the @GPS_LOCATION rule function to update the field.
- **Password Validation*:** The password validation field edit type requires users to enter their client password and validates the value entered against the password for the client. This entered value is hidden with character placement, displaying asterisks in place of each entered character. Includes the ability to define a message to the user when the password entered is not valid.
- **String:** The string field edit type allows the user to enter any printable character values. This field type can be used to provide a large text field to capture user input by spanning multiple columns and rows on the detail screen.
- **Signature Capture:** The signature field edit type allows for the entry of a signature on a client's screen that is stored as a bitmap image. The signature edit type cannot be selected from the edit type list. This is, rather, the default edit type for a field when that field targets a property with a data type of Signature. To display such a property the correct edit type selection is "Default." No other edit type should be used when targeting a signature property as the behavior of such a combination is undefined.
- **Tile Edit*:** The tile edit field type displays object properties in a tiled view allowing for add and edit interaction without starting a wizard screen.
- **Tile Display*:** The tile display edit type displays an object instance in a tiled view.
- **Time:** The time field edit type allows the user to enter a time of day value using a time control. This edit type will display three controls for the field to display and capture the hours, minutes, and seconds portions of the time value. It is recommended that this field edit type only be used with time property types.

Field Edit Type - Property Data Type Cross Reference

When the edit type of a field is set to "Default," the field will take on the behavior of the edit type that matches the data type of the property the field targets. The table provided here contains the cross reference between the field's edit and the property data type. This will then be the type of field displayed on the detail screen when it targets a property of the type listed and the edit type of the field is set to "Default." This refers only to the field edit type on wizard detail screens. Object detail screens will always display property values in read-only string fields.

Property Data Type	Default Field Edit Type
Boolean	Button (check box type)
Collection	read-only string field (field should have edit type specified).
Complex Table Selection	Complex Table Search List
Data Table Selection	Data Table Selection
Date	Date
Date and Time	Date and Time
Decimal Number	Decimal Number
Duration	Duration
External Data	External Data
Identifier	Identifier
Integral Number	Integral Number
Object	read-only string field (field should have edit type specified)
Signature	Signature
String	String
Time	Time

Field Definitions With Edit Type-Specific Attributes

Many of the field edit types available include attributes beyond those common to all fields. These edit type-specific attributes are necessary to define the behaviors specific to a given field’s edit type. As an example, a field defined to display complex tables will require attributes that specific the complex table to be displayed, the index used to sort the records, and so on.

The following section lists each of these field edit types and describes their type-specific attributes.

Barcode Scan

The barcode scan field edit type receives input from a barcode scanner. This type of field may also be defined to behave like a string field, accepting input from the device's keyboard. The scanning functionality is only available on detail screens used by scanner platforms on devices equipped with a barcode scanner.

Barcode Scan Attributes

Following are the attributes specific to a barcode scan field edit type. These are in addition to the common field attributes:

- **Label Types**: This attribute specifies the name of the barcode label type or types to support for this field. If the barcode being scanned is not one of these types, it will not be scanned. If this attribute is left blank, any label type supported by the device will be scanned.
- **Minimum Length**: This attribute specifies the minimum number of characters to scan in. If the number of characters is less than this minimum, the value will be ignored. The default for this attribute is no minimum. This value must be less than or equal to the **Maximum Length** attribute value.
- **Maximum Length**: This attribute specifies the maximum number of characters to scan in. If the number of characters is greater than this maximum, the value will be ignored. The default for this attribute is no maximum. This value must be equal to or greater than the **Minimum Length** attribute value.
- **Allow Typing**: This attribute specifies whether or not the user can type a value into the field in addition to scanning one in. If true, the user can type a value directly into the field.
- **Show Scan Button**: This attribute specifies whether or not a **Scan** button is drawn to the right of the barcode scan field. This button is labeled “Scan” and will activate the device’s scanner just as if the hardware scanner button is pressed.
- **Maintain Scan Focus:** This attribute specifies whether or not the scan focus should always exist for the field when displayed on the current screen. When selected, and when the user scans a barcode the value scanned in will be set to the barcode field regardless of where the current input focus may be on the screen.

Button Field Edit Type

The button field edit type defines a detail screen field with button behaviors to execute actions and capture values. A value can be defined for this field to set the value of the target transaction property of the field. When displaying an object the button can execute actions. Part of the definable behavior is the type of button to display, which may be a radio button, check box, or push button. This is the default edit type for fields targeting Boolean properties.

Included in the functionality of a button field is the ability to group multiple button fields on a detail screen. If multiple button fields are defined for the same detail screen, and they also target the same property, these button fields are then grouped. The resulting behavior of such a configuration is that only one of the buttons may be selected at the same time. This is most commonly the case when the button fields are defined to display radio button controls. However, this same behavior will be exhibited for any of the button display types.

Buttons fields defined for object detail screens should be used to execute actions. Button fields for wizard detail screens displaying a transaction or fetch should be defined to set the value of a property.

An image can be defined for display in place of one of the available button controls. In this situation, the image referenced can be an image list, with each image in the list being a square and the same size. Which image is used is based on the state of the button field. These images are then used based on their position, as follows:

1. Enabled, not selected
2. Enabled, selected
3. Disabled, not selected
4. Disabled, selected

Button Field Edit Type Attributes

Following are the attributes specific to a Field with an Edit Type of Button.

- **Button Type**: This specifies the kind of button control to be drawn for the field. This can be a radio button, check box, or a push button.
- **Value When Selected**: This attribute specifies the value to be assigned to the target property of the field when that field is displayed on a transaction or fetch detail screen. For a button field on a transaction, fetch, or object detail screen, if the target property matches the value set for this attribute, the button will be in a selected state.
- **Action When Selected**: This attribute is only enabled for button fields when the parent detail screen displays an object. The action referenced is executed when the button is selected. Normally the **Button Type** for this situation is a Push Button. The **Target** attribute references the target object of the action referenced in the action attribute.
- **Button Image**: This attribute references an image definition to be displayed for this field. In this situation the **Button Type** attribute will be ignored and the image selected in the **Button Image** attribute will be displayed in its place. The image itself will behave as if it is a button and will either execute an action when selected or set the value of the field’s target property.

Calendar View

The calendar view field edit type provides an interactive calendar to display an object collection property, with each object treated as a calendar event. Properties of the object type used should represent an event title, an event start date and time and an event end date and time. Definable behaviors include allowing users to change viewing options, setting start and end times for a work day, and days included in a work week.

As a part of the calendar view field edit type, actions may be defined for double-clicking a calendar event as well as for double-clicking a calendar day and time that currently contains no events. When defining this field edit type, it is likely the field will comprise most, if not all of the viewable screen area.

The resulting display will be a calendar control that may include events. A given event is represented by an object instance containing the event’s title, and its start and end date and times. The event will then be represented in the calendar view as a block spanning the days and times as provided by the start and end date and time values. Users will be able to interact with these events and/or with empty time periods on the calendar based on double-click actions that can be set for the field.

Note that this field edit type was only supported on the desktop builds of the Windows operating system supported by the Agentry Client in versions of the Agentry Mobile Platform prior to 5.1. Release version 5.1 and later of the Agentry Mobile Platform provide support for the mobile Windows operating systems supported by the Agentry Client.

Calendar View Data/Style Attributes

The following attributes for a field with a calendar view edit type define the collection property to use, and the properties within each object instance contained in the object collection to use for displaying events in the calendar view. They also include the styles that may be applied to the calendar view events.

Calendar Data

- **Collection:** Sets the object collection property that will be used as the data source for calendar events. Each object instance in this collection will be treated as a calendar event by the field.
- **Event Title:** This attribute specifies the value to display as the title of a given event. This may be either an object property within the objects of the collection, or the return value of a rule. If set to a rule, the rule is evaluated in the context of the object instance for the event and is expected to return a string value. This rule will be evaluated once for each object currently displayed in the calendar view. Changing the view options of the calendar on the Agentry Client at run time will result in the rule being evaluated again for each object displayed.
- **Event Start:** This attribute specifies the value to treat as the start date and time for an event. This may be either an object property within the objects of the collection, or the return value of a rule. If set to a rule, the rule is evaluated in the context of the object instance for the event and is expected to return an integral number treated as an Agentry date and time value. This rule will be evaluated once for each object currently displayed in the calendar view. Changing the view options of the calendar on the Agentry Client at run time will result in the rule being evaluated again for each object displayed.
- **Event End:** This attribute specifies the value to treat as the end date and time for an event. This may be either an object property within the objects of the collection, or the return value of a rule. If set to a rule, the rule is evaluated in the context of the object instance for the event and is expected to return an integral number treated as an Agentry date and time value. This rule will be evaluated once for each object currently displayed in the calendar view. Changing the view options of the calendar on the Agentry Client at run time will result in the rule being evaluated again for each object displayed.
- **Include Rule:** Sets the name of the rule that can be used to limit the objects in the collection that will be displayed.
- **Tool Tip Rule:** Sets the rule that can be used to format a tool tip text when a user hovers the cursor over an event displayed in the calendar as an event.

Calendar Styles

- **Highlight Events:** This attribute provides the style to apply to events that should be highlighted in the calendar view. This style should be set based on a rule in any real-world use cases. The rule is evaluated in the context of the object instance representing the event and is expected to return a string value containing the name of the style to apply to those events that should be highlighted. An empty string will use the default style.

- **Selected Events:** This attribute provides the style to apply to a selected event in the calendar view. The style to be applied may be selected from those that are defined, or the name of the style can be returned from a rule. If a rule is used, it will be evaluated in the context of the object instance representing the event when the user selects the event. The rule is expected to return a string value containing the name of the style to apply.

Calendar View Options Attributes

The calendar view options attributes define behaviors related to the options a user may set, including whether or not to allow the user to set those options. These include the different views the calendar supports, the days of the week to treat as work week days vs. weekends, and whether or not those weekend days should be compressed in the month view.

Calendar Options

- **Allow User to modify Options:** This attribute controls whether or not users can change the calendar options on the Agentry Client. If set to true, the remaining attributes set here are treated as the defaults for the calendar behavior, which the users can then override. If this attribute is set to false, the options set here will define the behaviors exhibited at all times for each view supported by the calendar.
- **View:** Sets the default view for the Calendar:
  - **Day:** Displays the current day
  - **Month:** Displays the entire month, including weekends.
  - **Week:** Displays the entire week, Sunday through Saturday, including both work and not work days.
  - **Work week:** Displays the current work week. The days in the work week are defined in the Work Week section.

Day View - These attributes affect the appearance and behavior of the calendar view when set to the Day View on the Agentry Client.

- **Time Scale:** Sets the time scale, in minutes, for the time rows in the calendar. Options for Time scale are in minutes, can be in increments of:60, 30, 15, 10, 6, or 5.
- **Start Time:** Sets the calendar start time in increments of 30 minutes. The calendar uses a white background color when displaying times between the start time and end time.
- **End Time:** Sets the calendar end time in increments of 30 minutes. The calendar uses a white background color when displaying times between the start time and end time.

Work Week View - These attributes affect the appearance and behavior of the calendar view when set to the Week or Work Week view on the Agentry Client.

- **Sunday - Saturday check boxes:** Sets which days of the week are included in a Work Week. Unchecked days are treated as weekends.

- **First Day of Week:** Sets the first day of the week displayed on the calendar as the left-most day.

Month View - This attribute affects the calendar only when the field is set to the month view.

Compress weekends: Check to compress Saturday and Sunday in the calendar month view. Unchecked, and Saturday and Sunday will display like the other days in the week.

Calendar View Actions Attributes

The attributes for actions in a calendar view allow for the definition of actions to be executed when the users double-click an event in the calendar, and when double-clicking an open time slot within the calendar.

- **Double-Click on Event Action and Target:** These attributes allow for the definition of an action to be executed when a user double-clicks an event in the calendar, and the target object of that action. This target is normally the object representing the event just selected. In most use cases the action may display that event and/or allow the user to edit that event.
- **Double-Click off Event Action and Target:** These attributes allow for the definition of an action to be executed when a user double-clicks a time slot within the calendar that does not currently contain an event. This target is normally the parent object of the collection being displayed by the calendar view field. In most use cases the action will allow the user to add a new event starting at the day and time slot double-clicked in the calendar. The selected begin or end date may be retrieved via the rule function `SCREENFIELDVALUE` by passing the name of the calendar view field and either of the parameters `SelectedBeginDate` or `SelectedEndDate`.

Complex Table Drop Down

The complex table drop down field edit type displays a drop down list of unique values from a defined complex table field. This screen field edit type is normally used in a cascade control series, allowing users to drill down through records within the table that have a parent-child relationship.

A cascade is a series of multiple fields all displaying the same complex table. Each cascade field displays a different complex table field. The records displayed in a field are the child records to the selected record in the field before it in the cascade. The parent-child relationship is determined by the structure of the indexes for the complex table being displayed.

The overall behavior of a cascade will force a user to make a selection in the first field in the cascade, which is a top-level parent record in the complex table. The next field in the cascade will not be enabled until this selection is made. At this point, the values listed in the second field will be only those complex table records that are children to the record selected in the first screen field.

This behavior repeats for each field in the cascade. Defining such a cascade requires that the complex table displayed by these fields have the needed parent-child indexes defined. Each cascade screen field must then have a matching child table index at the same level.

As an option to displaying the values of a complex table, this field type can display a complex table search dialog. Within this dialog the records of the complex table will be listed within multiple columns, one for each complex table field. The records displayed in this dialog will be dependent on the selections made in previous fields in the cascade control series. Also as definable behaviors in this dialog are options to specify which indexes to allow a user to search on and the specific complex table fields to display in the list.

This field edit type also supports scanning as input. When this behavior is enabled, the value scanned in must be one that can return a record using the index specified for the screen field.

Both the complex table drop down and complex table list field edit types support cascade behavior. Fields of both types may used in the same cascade series of fields on a given detail screen.

Complex Table Drop Down Attributes

Following are the attributes specific to fields with an edit type of complex table drop down. These are in addition to the common field attributes:

- **Complex Table**: This attribute specifies the complex table the field is to display. In a cascade it is possible for each field to display a different complex table, provided the values selected in one can be used to search the next. This is not the recommended method for using cascades, as it is more efficient to use a single complex table for all cascade fields.
- **Table Index**: This attribute specifies the complex table index that should be used to search that table. For the first field in a cascade, this index should be the top-level index, that is, an index that does not have a parent index. For subsequent cascade fields, the selected index should be the child index to the index selected in the field that precedes it in the cascade.
- **Cascade Parent**: This attribute references another screen field on the same detail screen to use as the cascade parent for the field. If left set to Auto, the cascade will be determined based on the selected Table Index. The field whose Table Index is set to the parent index of the selected Table Index for the field will be treated as the cascade parent screen field. If the proper index structure is in place in the complex table, the cascade parent can be left set to auto.
- **Display Field**: This attribute specifies the complex table field to display for the table records. Leaving this attribute set to Auto will display the field upon which the index selected in Table Index is defined. This is the recommended selection for this attribute. Only unique values of this field will be listed.
- **Return Field**: This attribute contains the complex table field to return from the selected record. When left set to Auto, the return field will the field upon which the index selected in Table Index is defined. This is the recommended selection for this attribute, as this value will be the one passed to the next field in the cascade.
- **Selection Method**: This attribute specifies how the records of the complex table being displayed by the screen field are displayed to the user. Following are the options for this attribute:
  - *Always Drop Down Menu* - Always list the records of the complex table in a drop down list.

    - *Always Open Dialog* - Always list the records in a popup dialog.
    - *Always Open Dialog with Search* - Always open a complex table search list dialog that provides the user with the ability to enter search text to locate the desired record. This dialog will list all fields, or only those selected in the Search Dialog Indexes list, for each record in the complex table.
    - *Open Dialog if Needed* - Allows to the specification of record threshold. For this option, the default is to display the records in a drop down list. If the number of records to list exceeds the defined threshold, a popup dialog is displayed listing the records in a list box control.
- **Open Threshold:** This attribute can be set only of the **Selection Method** is set to Open Dialog if Needed. This attribute can be set to the Default, which will vary from one client device to another, or to a specific number of records. When the Open Threshold is exceeded, the popup dialog is displayed, listing the records from the table.
- **Scanning:** This attribute, when set to true, will enable the client device's scanner (if present). The user can scan a barcode value, which will be used to search the complex table by that value using the defined Table Index. If a single matching record is found, that will be the selection for the field. This attribute only has an impact if the client device has a scanner, and if the parent detail screen is used by a scanner platform.
- **Handle Special Value By:** This attribute is available only if a special value has been defined for the screen field. This attribute specifies what to do when the cascade parent value changes. You can define the field to then set itself back to the special value, change to the default text of "Please select," or to change to the "Please select text" only if the current selection is not the special value.

Search Dialog Indexes

For detail screen fields defined with an edit type of complex table drop down, there are two lists of items shown in the properties view of the Editor. The first is the Search Dialog Indexes tab. Listed in this tab will be one item for each top-level index defined in the complex table the screen field is displaying. For complex table drop down lists the selected search indexes will only affect behavior when the **Selection Method** attribute is set to "Always Open Dialog with Search." Each item contains a check box which, when selected, will display that index to the user as one that can be searched on. Those indexes not checked in this list will not be displayed to the user.

Search Dialog Fields

For detail screen fields defined with an edit type of complex table drop down, there are two lists of items shown in the Properties view of the Editor. The second is the Search Dialog Fields tab. Listed in this tab will be one item for each field defined in the complex table being displayed by the screen field. For complex table drop down lists the selected fields will only affect behavior when the **Selection Method** attribute is set to "Always Open Dialog with Search." Each item contains a check box which, when selected, will display that field to the user in the list of records. Fields that are not selected will not be shown to the user.

Complex Table List

The complex table list field edit type displays the records of a complex table in a list control on the detail screen. Definable behaviors include the complex table fields to display, the complex table field value to return when a record is selected, and an action to execute when a record is double-clicked. This field edit type is normally used in a cascade control, though this is not a requirement.

A cascade is a series of multiple fields all displaying the same complex table. Each cascade field displays a different complex table field. The records displayed in a field are the child records to the selected record in the field before it in the cascade. The parent-child relationship is determined by the structure of the indexes for the complex table being displayed.

The overall behavior of a cascade will force a user to make a selection in the first field in the cascade, which is a top-level parent record in the complex table. The next field in the cascade will not be enabled until this selection is made. At this point, the values listed in the second field will be only those complex table records that are children to the record selected in the first screen field.

This behavior repeats for each field in the cascade. Defining such a cascade requires that the complex table displayed by these fields have the needed parent-child indexes defined. Each cascade screen field must then have a matching child table index at the same level.

Both the complex table drop down and complex table list field edit types support cascade behavior. Fields of both types may used in the same cascade series of fields on a given detail screen.

Complex Table List Attributes

Following are the attributes specific to the complex table list field edit type. These attributes are in addition to the common field attributes:

- **Complex Table**: This attribute specifies the complex table the field is to display. In a cascade it is possible for each field to display a different complex table, provided the values selected in one can be used to search the next. This is not the recommended method for using cascades, as it is more efficient to use a single complex table for all cascade fields.
- **Table Index**: This attribute specifies the complex table index that should be used to search that table. For the first field in a cascade, this index should be the top-level index, that is, an index that does not have a parent index. For subsequent cascade fields, the selected index should be the child index to the index selected in the field that precedes it in the cascade.
- **Cascade Parent**: This attribute references another screen field on the same detail screen to use as the cascade parent for the field. If left set to Auto, the cascade will be determined based on the selected Table Index. The field whose Table Index is set to the parent index of the selected Table Index for the field will be treated as the cascade parent screen field. If the proper index structure is in place in the complex table, the cascade parent can be left set to auto.

- **Fields to Display**: This attribute can contain the name of each complex table field to display in the list. Each table field name is listed here, separated by a comma. If no fields are listed here, all fields are displayed in the list. Each field displayed in the list is represented by a list column.
- **Return Field**: This attribute contains the complex table field to return from the selected record. When left set to Auto, the return field will the field upon which the index selected in Table Index is defined. This is the recommended selection for this attribute, as this value will be the one passed to the next field in the cascade.
- **Double-Click Action**: This attribute references an action to execute if the user double-clicks a record in the list. The target of this action will always be the object instance the parent detail screen is displaying. The complex table record the user double-clicks is the current record and can be accessed as such through the target browser. This attribute can only be set when the field is displayed on an object detail screen and will have no affect on a wizard detail screen for a transaction or fetch instance.
- **Handle Special Value By**: This attribute is available only if a special value has been defined for the Field. In this case, this attribute specifies what to do when the cascade parent value changes. You can define the Field to then set itself back to the special value, change to the default text of Please select, or to change to the Please select text only if the current selection is not the special value.

Complex Table Search

The complex table search field edit type displays the records of a complex table in a searchable list. This field edit type displays a field with an ellipses button. When the ellipses button is clicked, the searchable list screen is displayed. By default users may search the records of the complex table on any defined top-level index for a string field. Alternately, a single search index may be specified as a part of the screen field's definition. This field edit type also supports scanner functionality to select a record.

When scanner functionality is enabled, the scanned value will be used to search the complex table on the selected search index. Only those records that match will be listed and the user may make a selection from this filtered list.

Complex Table Search Attributes

Following are the attributes specific to the complex table search field edit type. These attributes are in addition to the common field attributes:

- **Complex Table:** This attribute specifies the complex table whose records will be listed in the search screen.
- **Search Index:** This attribute can be set to restrict the index used to search the complex table. If an index is selected for this attribute, the user will only be able to search the complex table using that index. By default, all top-level indexes on string fields can be used to search the records. The Search Index can be defined any index, parent or child, which will then be used for all searches of the complex table when using this field.

- **Parent Value:** This attribute can be set to the value by which the records should be filtered when the **Search Index** is set to an index that is a child to another index.
- **Initial Value:** This attribute can be set to a property of the definition. This will set the initial value of the field to the value of this property. The user can still select a complex table record to change this value. By default, there is no Initial Value.
- **Display Field:** This attribute can be set to any field within the complex table and specifies the field value to display in the screen field for the selected table record. By default the value displayed in the screen field is the field upon which the complex table's primary index has been defined, i.e., the field containing the unique value for each table record.
- **Return Field:** This attribute can be set to any field within the complex table and specifies the table field to return to the screen field from the selected table record. By default the value returned is the table field upon which the complex table's primary index has been defined, i.e., the field containing the unique value for each table record.
- **Allow Scanning as Input:** This attribute can enable scanner functionality for the search screen. When enabled, the user can scan a value that will be used to search the complex table using the selected search index. Only those records matching this search will be listed. This attribute will only affect screen fields for detail screens used by a scanner platform displayed on client devices with barcode scanners.

Search Dialog Indexes

For detail screen fields defined with an edit type of complex table search, there are two lists of items shown in the properties view of the Editor. The first is the Search Dialog Indexes tab. Listed in this tab will be one item for each top-level index defined in the complex table the screen field is displaying. Each item contains a check box which, when selected, will display that index to the user as one that can be searched on. Those indexes not checked in this list will not be displayed to the user.

Search Dialog Fields

For detail screen fields defined with an edit type of complex table search, there are two lists of items shown in the Properties view of the Editor. The second is the Search Dialog Fields tab. Listed in this tab will be one item for each field defined in the complex table being displayed by the screen field. Each item contains a check box which, when selected, will display that field to the user in the list of records. Fields that are not selected will not be shown to the user, unless all fields are not selected, in which case all fields of the complex table are displayed as columns in the list.

This dialog also allows the developer to specify the order in which columns should be displayed in the list control of the search dialog. For the selected fields, a position value is assigned and can be adjusted by moving the field up or down in the list.

Complex Table Tree

The complex table tree field edit type displays the records of a complex table in a tree control, providing a parent-child relationship to the records. Each node in the tree control represents a complex table record. This edit type displays a field on the detail screen with an ellipses button. When this button is clicked the screen containing the tree control is displayed. Definable behaviors include the table indexes to be treated as the parent and child indexes, the starting point of the records, the number of levels below the start point to display, and the complex table field values to display from each record in each node of the tree control.

The complex table tree field edit type allows for the creation of parent-child relationships that do not exist in the complex table's structure. Part of the definition of a field of this type is the selection of two indexes in the complex table, both of which are top-level indexes. One will be used as the parent index and the other the child. This relationship will only exist while the tree control screen is displayed.

Complex Table Tree Attributes

Following are the attributes specific to the complex table tree field edit type. These attributes are in addition to the common field attributes:

- **Complex Table**: This attribute references the complex table whose records will be displayed in the tree control screen.
- **Parent Index**: This attribute references the index within the complex table to use as the parent index. Records will be organized in the tree control according to their common parent based on this index. Each node will contain child nodes with the same value in the field for which this index is defined.
- **Child Index**: This attribute references the index within the complex table to use as the child index. Each record with a unique value in the field upon which this index was created will be listed in the tree control under the parent record.
- **Search Index:** This attribute can be set to restrict the index used to search the complex table. If an index is selected for this attribute, the user will only be able to search the complex table using that index. By default, all top-level indexes on string fields can be used to search the records.
- **Parent Root**: This attribute can be set to a value found in the complex table field for which the selected Parent Index is defined. Any records with the Parent Root value in this field will be treated as the top-level parent records by the complex table tree field. The resulting behavior will be that these records will be listed as the root nodes in the tree control.
- **Display Field**: This attribute references a complex table field whose value will be displayed for each node in the tree control. This same value will also be the one returned to the field for display on the detail screen containing the complex table tree field definition. If this value is not set, the default is to display the field for which the complex table's primary index was defined. The Display Format attribute to this screen field edit type can also affect the appearance of the nodes in the tree control.
- **Return Field**: This attribute specifies the complex table field to return for the selected record for the purpose of setting the property targeted by the screen field. By default, the complex table field for which the primary index was defined is the value returned.
- **Display Type**: This attribute can specify how each node in the tree control will be displayed. The default is to display the table field value from the **Display Field** attribute for the record each node represents. The other alternatives are to display not only the value for that record, but also the values of each ancestor to that record. This can be in either a

parent-to-child order (Root to Selected Item), or in a child-to-parent order (Selected Item to Root). The value from each record displayed in the node can be separated in the display using the Connect Items With attribute (discussed below). The **Display Field** attribute contains the field value displayed for each record.

- **Display Format**: This attribute can contain format strings to format the display of the selected record in the detail screen field. To access the values of the selected record, use the format string syntax of `%`*`fieldName`* where *`fieldName`* is the name of the complex table field whose value is to be displayed.
- **Tree Format**: This attribute can contain format strings to format the display of the nodes in the tree control. Using these format strings, you can display additional complex table fields for each record in its respective node. This will be in addition to the value selected in the display field attribute. To reference a complex table field, the syntax is `%`*`fieldName`* where *`fieldName`* is the name of the complex table field.
- **Connect Items With**: This attribute can contain a character that will be placed in between each of the values displayed in a single node. This attribute is only available if the **Display Type** is set to either Root to Selected Item, or Selected Item to Root. The character(s) contained in the **Connect Items With** attribute will be placed between the values for each record in the hierarchy within a single node of the tree control.
- **Word Wrap**: This attribute, when set to true, will wrap the text of the nodes to the next line, if it spans beyond the viewable area of the screen. The default is to not wrap the node values, requiring the user to horizontally scroll the tree control for longer values.
- **Sort**: This attribute controls how child nodes are sorted in relation to nodes that begin with a hyphen. In many cases, a complex table will contain a default record, such as `"--None--"`. The Sort attribute can specify that such records are sorted either before or after the other nodes. The default setting will place these items wherever they may be sorted according to the locale settings of the client device.
- **Depth**: This attribute can specify how many levels of the hierarchy within the complex table you to display in the tree control. The number of levels refers to the number of descendents to display below the root node. This value is relative to the Parent Root, if one is specified. Note that this value does not specify the actual level, but rather the number of levels counting from the Parent Root.
- **Search**: If this attribute is enabled the search controls in the complex table tree screen will be hidden.
- **Scanning**: This attribute can enable barcode scan searches of the complex table records within the tree control. The complex table will be searched using the specified Child Index for the value scanned in. The first matching record will be selected in the tree control. This attribute only affects complex table tree fields defined for detail screens used by a scanner platform and displayed on a device equipped with a barcode scanner.

Data Table Selection

The data table selection field edit type lists the records of a data table in a drop down list control on the detail screen. Definable behaviors of this list include the data table field to display, the sort order for display, and the value by which to sort. A popup screen may be displayed based on the number of records in the data table. This threshold is different for each supported device type.

When displaying the records from the data table in the drop down list, the code, value, or both may be displayed. Additionally, format strings may be used to format the text for each record. When a selection is made in the list, the value returned to the field is always the code portion of the selected record. This will be the value set to the target property of the field.

Data Table Selection Attributes

The following attributes are specific to the data table selection field edit type and are in addition to the common field attributes.

- **Data Table Name**: This attribute references the data table whose records will be listed in the drop down list for this field.
- **Sort By**: This attribute allows you to sort the values listed in the drop down list by one of several options: Code, which is the code field in each table record; Value, which is the value field in each record; Displayed Text, which is the text displayed for each record in the field; and Order in Data Table, which is the order in which the records are listed in the data table itself.
- **Sort Order**: This attribute specifies whether the records displayed are sorted in ascending or descending order. This is a string sort.

Field Attributes

- **Display Type**: This attribute specifies which fields from the data table records should be displayed in the drop down list. The options are: Code, meaning the code field in each table record; Value, which displays the value field from each record; Code - Value, which displays both fields from each record, separating them with a hyphen; and Format Text, which allows you to specify format strings to format the values displayed in the list for each record.
- **Format Text**: If the **Display Type** attribute is set to “Format Text” this attribute will be enabled. Format strings can then be entered in this attribute to format each record from the data table. The valid format strings for this field are `%code`, `%value` and `%position`. This last will display the position number of each record as stored in the data table. This last option is used mostly for testing purposes and is generally not found in the production version of an application. This attribute can also contain any other printable characters, excluding tabs and carriage returns, to format the display of the table’s records.
- **Editable:** This attribute specifies whether or not users can manually enter text values in the field for values not found in the data table displayed by the field. When this attribute is set users can either select from the list or enter a value manually. When not set, users will be required to select an item from the list. This attribute can be set if the field is defined to be read-only if the field also has an update rule defined, if that rule can return values not found in the data table.

Popup Dialog Attributes

- **Define separate display type for popup dialog**: This attribute allows you to display the records from the data table differently in the popup dialog vs. the drop down list for the field. If set to false, the display and format attributes listed above will also affect the popup dialog. If set to true, the attributes listed next will provide separate display behaviors for the popup dialog.
- **Display Type** (Popup Dialog): The options for this attribute are the same as the Display Type options listed previously. The option selected here will impact the appearance and behavior of the popup dialog displayed for larger data tables.
- **Format Text** (Popup Dialog): The format strings for this attribute are the same as the Format Text options listed previously. The format text entered here will impact the appearance and behavior of the popup dialog displayed for large data tables.

Embedded Image Field

The embedded image field edit type displays an application-level image definition on the detail screen that can be interactive. Definable behaviors include whether or not to resize the image to fit in the space allocated for the field, the cropping behavior of the image displayed, and the ability to divide the image into cells to elicit different behaviors when different portions of the image are selected.

Each cell in an embedded image field is represented by a child definition to the field in the Editor. This definition type is called an image cell. There will be as many of these image cells as there are cells in the image, which is a multiple of the rows and columns defined for the field.

Note that the embedded image field edit type was named the image field edit type in versions of the Agentry Mobile Platform. Starting with version 5.1 and going forward, this field edit type has been named embedded image. This is to distinguish this field edit type from the image capture field, which displays the contents of image properties. The embedded image field is provided to display image definitions at the application level of the application project hierarchy.

Child Definitions

- **Image Cell:** The image cell definition is a child definition to fields with an edit type of embedded image and represents a specific portion of the image being displayed. An image cell defines the action to execute or the value to set when the corresponding cell of the image field is selected by the user.

Image Field Edit Type Attributes

The following attributes are specific to the embedded image field edit type. These are in addition to the common field attributes:

- **Image**: This attribute specifies the image definition within the application that this field is to display.

- **Grid**: This attribute contains two numeric values, rows and columns. The product of these two values determines the number of image cell definitions for the field. A one by one grid will create a single cell representing the entire image field.
- **Resize to Fit**: This attribute specifies whether the image should be resized to fit in the space allotted to the field.
- **Lock the Aspect Ratio**: Available only when resize to fit is true, this attribute specifies whether or not the aspect ratio of the image should remain the same. If true, the aspect ratio will be locked. If false, an image too large for the field will be resized to the size and shape of the field, regardless of its affect on the appearance of the image.
- **Crop to Fit**: This attribute is only available when the resize to fit attribute is false. If **Crop to Fit** is true, the image will be cropped on its right and bottom edges to fit within the field.
- **Position**: This attribute specifies the position of the image within the space allotted to the field on the detail screen. This can be one of: upper-left; top; upper-right; left-center; center; right-center; bottom-left; bottom; or bottom-right.
- **Highlight**: This attribute specifies whether or not the currently selected cell on the image field is highlighted. If this attribute is true, you can specify how to highlight the cell(s). This may be either in 3D, or by specifying a mask color to be applied to the selected cell(s). Cells are considered selected if either the user selects them on the screen or if the value the cell is defined to set is equal to the value of the property targeted by the field.
- **Highlight Cells on Hover**: This attribute specifies whether or not the cell over which the mouse cursor is currently hovering is highlighted. If this is true, you can specify whether to highlight the cell in 3D or by specifying a mask color to be applied to that cell. This attribute only affects the Windows PC platforms.

Image Cell

The image cell is a child definition to a field with an edit type of image. The image cell definition represents a cell for the parent image field. A cell definition can define an action to execute or a value to assign to the field's target property when the cell is selected. Actions may be executed from detail screens for objects. Values may be assigned to properties from detail screens for transactions or fetches.

The editor allows for a single image cell to be edited within the image, or to edit multiple cell images at the same time. This is accomplished using the layout view for the image field. The grid will overlay the defined image in this view, and the cells may be selected and edited via right-clicking a cell. Multiple cells may be selected using `Ctrl+Click`. A single cell can be defined to set a value or execute an action, and then additional cells can be defined to be the same as that cell. This allows for multiple cells to be combined to define a region of the image, based on its appearance.

Image Cell Attributes

The following attributes define the behavior of the image field child definition image cell:

- **Cell**: This attribute specifies which cell the definition represents. This is a numeric value displayed in the format (*Row*, *Column*), where *Row* and *Column* are the points where the row and column intersect to create the cell.
- **Name**: This attribute is the name of the cell definition, which is set by default to `Cell_R_C`, where *R* and *C* are the row and column that make up the Cell.
- **Value When Selected** (Transactions and Fetches): This attribute is available for transaction and fetch detail screens and defines the value to be set to the property targeted by the cell's parent field definition when the cell is selected.
- **Action** (Objects): This attribute references the action to execute on object detail screens when the user selects the cell.
- **Action Target**: This attribute specifies the object instance that is targeted by the action executed.
- **Tooltip When Hovered Over**: This text field can contain any text value. This will be the text displayed on the client when the user hovers the mouse cursor over the cell. This attribute only impacts image fields displayed on detail screens for the Windows PC platforms.

External Field - ActiveX Control

The external field-ActiveX control edit type is defined to call out from a field to an ActiveX control. Values may be passed to this control from the Agentry Client.

Use of this field requires an ActiveX control exist on the client devices and that control be built using the Agentry ActiveX Control API, including the implementation of all Expected Methods.

Using the **Agentry Data** and **Actions** tabs allows an ActiveX control to query Agentry for data and for an ActiveX control to call for Agentry to execute actions. Agentry can also query the ActiveX control for any values listed in the **External Values** tab.

External Field - Active X Control Attributes

The following attributes are specific to the External Field - ActiveX control field edit type. These are in addition to the common field attributes:

- **ActiveX Class Name (Prog ID):** This attribute contains the class name that the Agentry Client will interface with for the ActiveX control.
- **Allow Scanning as Input:** This attribute specifies whether or not the field displayed will accept barcode scan values as input. This attribute will only impact fields displayed on detail screens used by a platform that supports scanner behavior and on client devices equipped with a barcode scanner. When value is scanned for the field, the ActiveX control expected method `AgentryUpdateScanData` to pass the barcode value to the ActiveX control.
- **External Values Tab:** The External Values tab is a list of values provided by the ActiveX Control. This will allow the Agentry Client to query the control for data. From the tab, you can add and delete value names from the list. The ActiveX control referenced by the detail screen field must include the proper processing within the

`AgentryGetSpecificValue` method to return the value(s) associated with each of the External Values listed in this tab.

- **Agentry Values Tab:** The Agentry Values tab is a List of names and target paths for values within Agentry, made available to the ActiveX Control. From the tab, you can link Agentry data with the external values for the ActiveX Control. Both primitive data types as well as object instances and collection properties can be made available to the ActiveX control. The name associated with the selected data item is the identifier exposed to the ActiveX control, which can call the `GetPropertyFromMappings` Agentry Client-Side API method, passing the name to retrieve the desired value.
- **Actions:** Allows the ActiveX control to call for Agentry to execute actions. The Properties tab gives you a list of Actions and target paths. Within this list actions can be added and deleted. When an action is added it must also specify a target object for the action. The ActiveX control can call the `ExecuteAgentryAction` Agentry Client-Side API method, passing the name of the action to be executed.

HTML

The HTML field edit type supports the formatted display of HTML markup text, or the display of a defined URL for internet navigation. Definable behaviors for this field include whether or not to display the navigation toolbar, a list of parameters to be pased to a URL, and the ability to provide either a list of permitted or prohibited URL's to restrict the navigation allowed by the user.

Included in this field edit type are two child definitions, which are the Domain List and the URL Parameters. The domain list items can be used to specify to which URL's users can navigate. The URL parameters can define the parameters to pass to a URL. These values are derived from a rule and can therefore be dynamic.

The HTML field edit type can also display HTML pages or text retrieved from some source, such as the back end system. This allows for a web page to be displayed within the field on the detail screen, which can then provide links to internal or external pages.

HTML Child Definitions

- **Domain:** The domain definition is a child definition to detail screen fields with an edit type of HTML, and can specify the URL's to which users can navigate or those they should be prevented from viewing.
- **URL Parameter:** The URL parameter is a child definition to detail screen fields with an edit type of HTML, and can specify an argument value to be passed to the URL the field is defined to display.

Attributes

Navigation Bar

- **Initial State-Show Navigation Bar:** This attribute specifies whether or not to display the navigation bar for the HTML field when the parent screen is initially displayed. If this option is not set, the user can display the navigation bar by right-clicking the field and selecting the popup menu item to display it.

Domain List

- **Domain List Contains:** This attribute specifies whether the domains added to the HTML field as child definitions specify those URL's to which the user is allowed to navigate, or those URL's to which they should be prevented from viewing.

Domain and URL Parameter

The domain definition is a child definition to detail screens with an edit type of HTML, and can specify the URL's to which users can navigate or those they should be prevented from viewing. This definition type contains a single attribute of Name, which contains the URL for the domain definition. The parent HTML field then specifies whether all child domain definitions are those that are allowed to be viewed, or those that should be blocked.

The URL parameter is a child definition to detail screen fields with an edit type of HTML, and can specify an argument value to be passed to the URL the field is defined to display. The value for each URL parameter is specified via a rule definition, making the values dynamic.

Image Capture

The image capture field edit type provides integration with the client device's built-in digital camera, allowing for images to be captured and stored in properties of the application. Using this field type it is also possible to select an image file on the client device to store in the property. This field edit type is intended for use only with properties of type image.

The image capture field edit type interacts with the client device's camera, if one is available. When displayed on the detail screen, the field will include up to two buttons. One will allow the user to select a file from the client device's file system. The other will interact with the camera, taking a picture that will be captured and displayed in the field. This behavior is exhibited only on detail screens displaying a transaction or fetch. For object screens, the image capture field will display the image stored in an image property in a read-only field.

When the camera button for the image capture field is clicked, a dialog is displayed allowing the user to take a picture using the device's camera. When the image is captured it is displayed as a thumbnail in the image capture field. The user can then click this image to display a popup screen of the image. The size of the image displayed in this popup is dependent on the Initial Popup Mode attribute for the image capture field. This option allows for the image to initially be displayed in either full size or to be scaled to fit within the popup screen. This screen will be no larger than the viewable area of the client device's display. Within this popup screen the user can click the image to switch between the scaled and full size image views. The dialog will contain scroll bars to allow the user to scroll the image if it is larger than the viewable display area.

Image Capture Attributes

- **Allow Image Camera Capture:** This attribute specifies whether or not to allow the field to interact with the device's camera, if one is available. When this attribute is set, the field will include a button control that will activate the camera to take a picture
- **Allow Image File Capture:** This attribute specifies whether or not to allow the field to capture an image file stored on the device's file system. When this attribute is set, the field will include button control that will display the Windows file dialog, allowing the user to select an image file from the file system.
- **Initial Popup Mode:** This attribute specifies how the image should be initially displayed in the popup screen when the image capture field is clicked by the user. The options are display the captured image in full size or to scale the image display to the size of the popup dialog. This is the initial display mode and the user can switch between the two by clicking the image in the popup dialog.
- **Image Location:** Specifies the location in which the image should be stored once it has been captured.
- **Image Name Prefix:** Specifies a string value to affixed to the beginning of the image file name.

List Tile View

The list tile view field edit type displays an object collection property in a tiled view allowing for add and edit interaction with the collection through the field. For a given object, the properties of that object can be displayed in the list tile view in tiles within that object's record. The values of a given object can be edited directly in this list, and new object instances can also be added. This field edit type also supports scan filter functionality.

The list tile view will make use of an existing screen set defined to display the same object type as is stored in the collection being listed by the field. As a part of a list tile view's definition, a screen set is selected to display the objects within the collection. This screen set must be defined to display the same object type as is found in the collection, and must contain a single detail screen. When the list tile view field is displayed, each object instance within the collection will be displayed within the field in a list. Each tile within the list will be shown in the detail screen from the selected screen set. Two screen sets can be used for read-only display of the objects within the collection. One is used for all rows within the list. The selected row screen set can be defined, with a detail screen containing more fields. This will then be the screen set used to display the selected object.

Similar to this behavior is the ability to add and edit objects for the collection from within the tile view. A transaction for the add and edit behaviors must exist, as must a screen set to be used to display the transactions. When an item is selected in the list, the user can click the add icon button. A new tile will be displayed at the bottom of the list. The screen set in which it will be displayed will be the one defined for the add behavior. For an edit, the user can select an object in the list tile view and click the edit icon button. In this case the currently selected tile will change to use the edit screen set, displaying the edit transaction. The user can change the values on the screen. They can then either cancel or accept the changes they have made.

As alternatives to this behavior, an action can be specified for both add and edit behaviors. When the action is executed it will dictate the behavior, displaying the add or edit transaction in the wizard screen set just as with any other action.

Filtering can be enabled or disabled for the entire list tile view field. When enabled, the properties of the object type being listed are selected. The user will then only be able to filter the list on these properties. When a property is selected for this purpose, a Tile Filter child definition is added to the list tile view. Users will then only be able to filter the items in the list on one of the selected property values.

Related to the manual filtering, this field edit type also supports scanner filtering. A tile filter can be defined to support scanner filtering. When a barcode value is scanned in it will be compared to the values of that tile filter's property. Only those items that match will be listed. The parent field can then be defined to execute an action when a single item in this list matches the scan filter, and a separate action to execute when no items match.

List Tile View Child Definitions

- **Tile Filters:** The tile filter is a child definition to a detail screen field with an edit type of list tile view, defining the values upon which the items listed in the parent field can be filtered.
- **Sort Properties:** This child definition is a simple list of the object properties by which the list tile view can be sorted at run time on the Agentry Client. When adding a sort property a selection is made from the properties defined in the object type for the collection which the List Tile View field is defined to display.

List Tile View - Collection/Styles Attributes

The list tile view field edit type does not support the following general field attributes:

- Object/Transaction Property
- Format
- Field Style
- Focused Field Style
- Change Focus
- Update Rule
- Special Value

The list tile view data and style attributes set the basic behavior of the view, including how styles can be applied to the list tile view field.

General Settings

- **Collection:** References the object collection property the list tile view is to display. This collection is normally a property of the object definition the parent screen set is defined to display.
- **Include Rule:** References a rule definition expected to return a Boolean value and that is evaluated once for and in the context of each object in the collection displayed by the list view. When an include rule is specified, only those objects for which the rule evaluates to true will be listed in the list tile view.

Styles Settings

- **Header Label:** The style to apply to the list tile view's header label. If no header label is defined this attribute has no affect on the screen.
- **Rows:** The style to apply to all rows on the list tile view.
- **Alternate Rows:** The style to apply to every other row in the list, beginning with the second row.
- **Highlight Rows:** The style to apply to a row for the purposes of drawing attention to that row. This style should always be returned via a rule definition that evaluates the object being listed.
- **Selected Rows:** The style to apply to the row currently selected by the user in the list control.

List Tile View - Settings Attributes

Selection Settings

- **Allow Multi-Row Select:** Specifies if the user can select more than one record in the list at the same time. If multiple items are selected in a list, actions that target the selected object in the list will be executed once for each selected object. The default for selecting multiple objects requires a `Ctrl+Click` combination (mouse input) or a click and drag operation (stylus input) by the user, depending on the device type. The **Enable Single Click** option to this attribute may be set to allow multiple records to be selected with a single click by the user. Deselecting a record requires the user to click it again. This feature is normally most useful on touch screen devices, as it allows non-sequential records in the list to be selected. If this option is enabled the attributes related to editing the objects in the list tile view will be disabled. These objects may still be edited as the selected object in the list tile view, but the action must be executed from a control on the same screen as the list tile view field, rather than from within the list tile view itself.

Action Settings

- **Allow Tile Adds:** This attribute specifies whether or not users will be able to add a new object to the collection being displayed by the list tile view field from within the field. When this option is selected, the **Add Screen Set** and **Add Transaction** attributes must also be set.
- **Allow Tile Edits:** This attribute specifies whether or not uses will be able to edit an object within the collection displayed by the list tile view field. When this option is selected, the

Edit Screen Set and **Edit Transaction** attributes must also be set. **Allow Tile Edits** is disabled if the attribute **Allow Multi-Row Select** is enabled for the list tile view field.

- **Allow Single Click Action:** This attribute specifies whether to allow for an action to be executed when a tile is selected in the list with a single click. If this attribute is set to true, all default behaviors of the list tile view field for a single click of a tile are disabled, as are the related attributes within the definition. This includes the following attributes:
  - Allow Tile Edits
  - Allow Multi-Row Select/Enable Single Click Selection
  - All attributes in the section Edit Actions/Tiles

Screen Sets

- **Row:** This attribute specifies the screen set containing the detail screen to display each object contained in the collection being listed in the list tile view field. The screen set selected here will be used for each tile in the list that is not currently selected. The screen set referenced must be defined for the same object type as is contained in the collection being listed by the list tile view. The screen set must also contain a single detail screen used by the same platform as the parent screen of the list tile view.
- **Selected:** This attribute specifies the screen set containing the detail screen to display each selected tile in the list tile view field. The screen set selected here will be used only for a selected tile in the list. The screen set referenced must be defined for the same object type as is contained in the collection being listed by the list tile view. The screen set must also contain a single detail screen used by the same platform as the parent screen of the list tile view.

Add Actions/Tiles

- **Add Screen Set:** This attribute is enabled when the **Allow Tile Adds** attribute is set. Add Scree Set is set to the screen set in which the **Add Transaction** will be displayed within the list tile view field. This screen set is displayed when the user selects clicks the add icon button for the field, allowing the user to add the values for the new object instance.
- **Add Transaction:** This attribute is enabled when the **Allow Tile Adds** attribute is set. **Add Transaction** is set to the transaction that will capture the values from the user for the new object instance to be added to the collection being displayed by the list tile view field. The transaction will be displayed in the list tile view field, with the tile using the screen set selected in **Add Screen Set**.
- **Add Action:** This attribute is enabled when the **Allow Tile Adds** attribute is not set. **Add Action** can be set to the action to execute when the user clicks the add icon button for the list tile view field. This action will be executed, targeting the object selected in **Add Target**. The purpose of the **Add Action** attribute is to execute an action that will add a new object instance to the collection being displayed by the list tile view field.
- **Add Target:** This attribute is enabled when the **Allow Tile Adds** attribute is not set. **Add Target** is set to the object instance that the **Add Action** should target when executed. In

almost all scenarios the **Add Target** should be set to the parent object of the collection being listed by the list tile view field.

- **Add Shortcut Key:** This attribute is set to the shortcut key combination that will allow the user to add an object to the collection being displayed by the list tile view field. The shortcut key will exhibit the same behavior as if the add icon button for the list tile view field were clicked or tapped by the user, meaning either the defined **Add Action** will be executed, or the defined **Add Screen Set** and **Add Transaction** will be displayed in a new tile in the list tile view field.

Edit Actions Tiles

- **Edit Screen Set:** This attribute is enabled when the Allow Tile Edits attribute is set. Edit Screen Set is set to the screen set in which the Edit Transaction will be displayed within the list tile view field for the selected tile. This screen set is displayed when the user selects a tile in the list and clicks the edit icon button for the field, allowing the user to edit the values of the selected object instance.
- **Edit Transaction:** This attribute is enabled when the **Allow Tile Edits** attribute is set. **Edit Transaction** is set to the transaction that will capture the values from the user to modify the object instance selected in the list tile view field. The edit transaction will be displayed in the list tile view field with the tile using the screen set selected in **Edit Screen Set**.
- **Edit Action:** This attribute is enabled when the **Allow Tile Edits** attribute is not set. **Edit Action** can be set to the action to execute when the user clicks the edit icon button for the list tile view field. This action will be executed, targeting the object selected in **Edit Target**. The purpose of the **Edit Action** attribute is to execute an action that will allow the user to edit the selected object instance in the collection being displayed by the list tile view field.
- **Edit Target:** This attribute is enabled when the **Allow Tile Edits** attribute is not set. **Edit Target** is set to the object instance that the **Edit Action** should target when executed. In almost all scenarios the **Edit Target** should be set to the selected object instance of the collection being listed by the list tile view field.
- **Edit Shortcut Key:** This attribute is set to the shortcut key combination that will allow the user to edit the selected object in the list tile view field. The shortcut will exhibit the same behavior as if the edit icon button for the list tile view field were clicked or tapped by the user, meaning either the defined **Edit Action** will be executed, or the defined **Edit Screen Set** and **Edit Transaction** will be displayed in the selected tile of the list tile view field.

Single Click Action - These attributes are enabled only of the attribute **Allow Single Click Action** is set to true.

- **Single Click Action:** This attribute specifies the action to be executed when the user selects a tile in the list.
- **Single Click Target:** This attribute specifies the object to be targeted by the Single Click Action when it is executed.

List Tile View - Filter/Sort Attributes

General Settings

- **Fixed Sort Property:** Specifies the property definition within the object type being listed by which to sort the objects in the list tile. The **Order** option to this attribute is set to specify the sort order, either ascending or descending. For the list tile view it is recommended that this attribute be set, as the list tile view cannot be sorted by the user. If a Fixed Sort Property is not set, the order of the objects in the list will be the order in which they are stored in the collection.
- **Enable Groups:** Enables or disables the group and indexing behavior available in iOS Agentry Clients. When selected, the defined **Fixed Sort Property** is used to group the objects listed in the List Tile View field. Tiles will be sorted based on this selection and grouped by those with the first *x* number of characters (defined in No. Chars option) sorted relative to each other and exclusive to those in other groups. If the **Fixed Sort Property** is a string property, the **No. Chars** option is enabled where the number of characters to group on is defined. For numeric types, grouping is based on the first (highest order) digit. This value should be less than the maximum length of the selected string property. When **Enable Groups** is set to true, the attribute **Allow Filter** is disabled.
- **Show Group Index:** This attribute is only available when Enable Groups is selected. When set to true, this attribute will result in the display of a group index on the right side of the List Tile View field. The user can select one of the items in this list to filter the List Tile View to only the matching items.
- **Allow Sort:** This attribute enables or disables sorting of the List Tile View's tiles on the Agentry Client by the user. When enabled, a button is displayed on the top of the list tile view field that displays a sort dialog when clicked by the user. The user can select a property within the object type being listed and the sort order of either ascending or descending. This attribute is disabled if a Fixed Sort Property is defined.
- **Initial Sort Property:** This attribute allows for the selection of property to sort the list tile view field on during its initial display on the Agentry Client. If a property is selected for initial sorting, the option Order is available to define whether or not the initial sort order should ascending or descending. This attribute is not available unless **Allow Sort** is set to true.
- **Allow Filter:** Specifies whether or not the user can filter the items in the list tile view. A filter icon is displayed for the list tile view field when enabled. The user can click this icon to select filter options. Only those properties for which tile filters have been defined within the list tile view field can be selected by the user in the displayed filter dialog.
- **Shortcut Key:** This attribute specifies the shortcut key combination the user can enter on the Agentry Client to display the filter dialog for the list tile view field. This attribute will have no affect if **Allow Filter** is not set.

Header

- **Header Label:** Specifies the header text for the list tile view. A common use for this header label is the total number of objects displayed in the list vs. the total number of objects in the collection, which may be different when a filter is enabled. The format strings used for this purpose are `%DisplayedCount` and `%TotalCount`.

List Tile View - Scanner Attributes

The scanner attributes for a list tile view affect only those list tile view fields defined for a detail screen that is used by a scanner platform within the screen set and only when the screen set is displayed on a client device with a barcode scanner. At least one tile filter must be defined within the list tile view to support scan filtering.

Single Match

- **Use Edit Row:** This attribute specifies whether or not to use the defined edit behavior for the single object that matches the scan filter settings. When set, the selected object will be edited via either the defined **Edit Action**, or the defined **Edit Screen Set** and **Edit Transaction** in the List Tile View Settings attributes. If this attribute is set, the Single Match Action attribute will be disabled.
- **Single Match Action:** Specifies what action is executed when a scanned barcode value uniquely matches an object in the list tile view. The target of the action will always be the object instance found to match. This attribute will be disabled if the Use Edit Row attribute is set.

No Match

- **Use Add Row:** This attribute specifies whether or not to use the defined add behavior of the list tile view field. When set, defined Add Action will be executed, or the defined Add Screen Set and Add Transaction in the LIst Tile VIew Settings attributes will be displayed in a new tile added to the list. If this attribute is set, the No Match Action attribute will be disabled.
- **No Match Action:** Specifies what action is executed when the scan filter criteria does not match any records in the list. The target of the action is the parent object to the collection property displayed by the list tile view. This attribute will be disabled if the Use Add Row attribute is set.

Label

- **Label Types:** Specifies what barcode types are accepted by the Agentry Client. If no **Label Type** is specified, all types supported by the client device's scanner will be supported. To restrict the label types, enter the name of each label type to support, separated by a comma. Barcodes not listed will not be processed by the Agentry Client.
- **Minimum Value:** The minimum number of characters accepted by the Agentry Client from the device scanner. If the value scanned in contains fewer characters, it will be ignored.
- **Maximum Value:** The maximum number of characters to be accepted by the Agentry Client from the device scanner. If the value scanned in contains more characters, it will be ignored.

- **Shortcut Key:** This attribute allows a shortcut key combination to be defined to activate the device’s barcode scanner. This should be set to a key combination not already defined as a shortcut for any other items on the current screen or any system-level shortcut keys.

Tile Filter

The tile filter is a child definition to a detail screen field with an edit type of list tile view. A tile filter defines the property within the object type being listed upon which the items listed in the parent field can be filtered. This includes both manual, user defined filters as well as barcode scan filters.

A tile filter targets an object property within the object type being listed by the parent list tile view field. For this property, the tile filter then defines whether or not the user can filter on this property manually, and whether or not scan filtering is enabled for this value within the list tile view.

Tile Filter Attributes

- **Object Property:** This attributes contains the target path to the object property for the tile filter.
- **Allow Filter:** This attribute specifies whether or not the user can select this property from the list of object properties displayed in the filter dialog on the Agentry Client.
- **Scan Filter:** This attribute specifies whether or not the value scanned in by the client device’s barcode scanner should be compared to the value of the property targeted by the tile filter. It is considered a best practice to set this attribute to true for only one tile filter within the same list tile view field, but this is not a requirement.

List Selection

The list selection field edit type displays a drop down list of values, the source of which may be an object collection, data table, or complex table. This list is treated as a temporary data table created at run time. Part of the definition of this edit type is to specify the values to be treated as the code and value fields for each record. Definable behaviors include whether to display the code, value, or both for each item listed. The code field is always the value returned from the selected item in the list.

A field with this edit type is displayed as a drop down list on the detail screen. If the number of records displayed in the list is large, a popup dialog will displayed when the user selects the field.

To use this edit type, either an object collection property or complex table is selected as the source for the items listed. Within this selected source two data members (object properties or complex table fields) are selected as the code and value for the records in the temporary data table.

Another aspect of this edit types behavior is the option to define an include rule. If used, this rule will be evaluated for each object or record in the defined source and only those items for which the rule returns true will be listed in the field. Note that this rule evaluation should be made as efficient as is possible when working with complex tables with large numbers of records.

List Selection Attributes

The following attributes are specific to the list selection field edit type and are in addition to the common detail screen field attributes.

- **Source**: This attribute specifies the source object collection property or complex table for the field. The collection or complex table may be returned via a rule, or it may selected from the target browser. Within the target browser, options exist for selecting object instances or a range of complex table records via a rule. Note that this is separate from using an include rule, which is another attribute to the field type. In most situations simply the object collection or complex table is selected here. It may be desirable for complex tables to return a range of records based on a table index. In this case the search value is provided in the target browser, which may come from an object property or a rule. Specifying a search value for the complex table to reduce the number of records for the field can significantly reduce the number of evaluations needed for the include rule, if one is used.
- **Include**: The Include attribute can reference a rule definition that will filter the records listed to the user. This rule is evaluated in the context of each data instance (object or complex table record) returned by the Source attribute and is expected to return a Boolean value. Only those data instances for which this rule returns true will be listed in the field. If no Include rule is selected, all instances of the selected source will be listed.
- **Key**: This attribute specifies the data definition within the **Source** instances to be used as the code (also known as the Key) field for each record in the temporary data table. This will either be an object property or complex table field, depending on the selected source.
- **Value**: This attribute specifies the data definition within the Source instances to be used as the value field for each record in the temporary data table. This will be either an object property or complex table field, depending on the selected source.
- **Sort By**: This attribute allows you to sort the values listed in the drop down list by one of several options: Code, which is the code field in each table record; Value, which is the value field in each record; Displayed Text, which is the text displayed for each record in the field; and Order in Data Table, which is the order in which the records are listed in the data table itself.
- **Sort Order**: This attribute specifies whether the records displayed are sorted in ascending or descending order. This is a string sort.
- **Display Type**: This attribute specifies which fields from the data table records should be displayed in the drop down list. The options are: Code, meaning the code field in each table record; Value, which displays the value field from each record; Code - Value, which displays both fields from each record, separating them with a hyphen; and Format Text, which allows you to specify format strings to format the values displayed in the list for each record.
- **Format Text**: If the **Display Type** attribute is set to “Format Text” this attribute will be enabled. Format strings can then be entered in this attribute to format each record from the

data table. The valid format strings for this field are `%code`, `%value` and `%position`. This last will display the position number of each record as stored in the data table. This last option is used mostly for testing purposes and is generally not found in the production version of an application. This attribute can also contain any other printable characters, excluding tabs and carriage returns, to format the display of the table's records.

- **Editable - Allow User-Entered Values:** This attribute specifies whether or not the user can manually enter values not found in the source of the list. When set users will be able to either select an item from the list or manually enter a text value in the field. When not set the users will be required to select from the items in the list. Note that if the field is set to read-only, this attribute should still be set if the field has an update rule defined, and if that rule can return a value not found in the data source for this field.
- **Define separate display type for popup dialog**: This attribute allows you to display the records from the data table differently in the popup dialog vs. the drop down list for the field. If set to false, the display and format attributes listed above will also affect the popup dialog. If set to true, the attributes listed next will provide separate display behaviors for the popup dialog.
- **Display Type** (Popup Dialog): The options for this attribute are the same as the Display Type options listed previously. The option selected here will impact the appearance and behavior of the popup dialog displayed for larger data tables.
- **Format Text** (Popup Dialog): The format strings for this attribute are the same as the Format Text options listed previously. The format text entered here will impact the appearance and behavior of the popup dialog displayed for large data tables.

List View

The list view field edit type displays an object collection property in a list control on a detail screen. This list contains all of the same definable behaviors as a list screen, but is contained within a detail screen. Multiple list views may be displayed on a single detail screen. This is the default edit type for a field targeting an object collection property.

When a field is defined with a list view edit type, that field will have column child definitions. A list view field can be defined on a wizard detail screen for a transaction or fetch. However, the attributes for the double-click actions will be disabled, as actions may not be executed from a wizard.

List View Child Definitions

- **Column Definition:** A list screen column defines what object property is displayed for each record in a list control and how it is formatted on the screen.

List View Data/Styles Attributes

The list view data and style attributes set the basic behavior of the view, including how styles can be applied to the list view field.

List Data

- **Collection:** References the object collection property the list view is to display. This collection is normally a property of the object definition the parent screen set is defined to display.
- **Include Rule:** References a Rule definition expected to return a Boolean value and that is evaluated once for and in the context of each object in the collection displayed by the list view. When an include rule is specified, only those objects for which the rule evaluates to true will be listed in the list view.
- **Icons Image:** References an image definition containing an image list to be displayed in a column on the list view. This is an image list with the positions of the images in each list then referenced by the child column definition's Icon attribute. Note that columns may also reference image definitions to use for this same purpose, though they may not be image lists.

List Styles

- **Header Label:** The style to apply to the list view's header label. If no header label is defined this attribute has no affect on the screen.
- **Column Labels:** The style to apply to the column labels on the screen's list control.
- **Rows:** The style to apply to all rows on the list view. The Hyperlinks optional style will override the Rows style for cells with hyperlinks.
- **Alternate Rows:** The style to apply to every other row in the list, beginning with the second row. The Hyperlinks optional style will override the Alternate Rows style for every other row, specifically cells containing hyperlinks within the row.
- **Highlight Rows:** The style to apply to a row for the purposes of drawing attention to that row. This style should always be returned via a rule definition that evaluates the object being listed. The optional Hyperlinks style will be applied to the highlighted row's cells containing a hyperlink.
- **Selected Rows:** The style to apply to the row currently selected by the user in the list control. The optional Hyperlink style will be applied to any cells within the selected row containing a hyperlink.
- **Detail Pane:** The style to apply to both the foreground (text) and background of the list view's detail pane. If no detail pane is defined this attribute has no affect on the screen.

List View Actions/Sorting Attributes

The list view actions and sorting attributes control how the user interacts with the list view, including double-clicking on or off an item in the list and behaviors related to sorting and reordering the columns. Note that the double-click attributes will be disabled for list view fields defined on wizard detail screens.

Double-Click Actions

- **Double-Click On Item - Action:** Specifies the action to execute when the user double-clicks a list view record.

- **Double-Click On Item - Target:** Specifies the target of the **Double-Click On Item - Action**. A target must always be specified for the action and is typically the selected object in the list view.
- **Double-Click Off Item - Action:** Specifies an action to be executed when the user double-clicks the list view without clicking on an item. This is most commonly used to execute an action that instantiates an add transaction for the object type being listed.
- **Double-Click Off Item - Target:** Specifies the target of the **Double-Click off Item - Action**. A target must always be specified for the action. Typically the target is the parent object of the object collection property displayed by the list view.

Sorting and Selection

- **Fixed Sort Property:** Specifies the property definition within the object type being listed by which to sort the items in the list. Selecting a property here prevents the user from resorting the list on any other column. The **Order** option to this attribute is set to specify the sort order, either ascending or descending.
- **Allow Sort:** Specifies if the user can sort the list by clicking on a column header. This is enabled by default, and is disabled if a **Fixed Sort Property** is set.
- **Initial Sort Column:** Specifies a column definition by which the list will be sorted upon initial display of the list view. This attribute requires that a column definition exist before it can be set. The **Order** option to this attribute is set to specify the sort order, either ascending or descending. If the list view allows the list to be sorted (**Allow Sort** is true) the list will be displayed sorted in the order of the last sort action. If a **Fixed Sort Property** is set, this attribute is disabled.
- **Allow Multi-Row Select:** Specifies if the user can select more than one record in the list at the same time. If multiple items are selected in a list, actions that target the selected object in the list will be executed once for each selected object. The default for selecting multiple objects requires a `Ctrl+Click` combination (mouse input) or a click and drag operation (stylus input) by the user, depending on the device type. The **Enable Single Click** option to this attribute may be set to allow multiple records to be selected with a single click by the user. Deselecting a record requires the user to click it again. This feature is normally most useful on touch screen devices using a stylus, as it allows non-sequential records in the list to be selected.
- **Allow Reorder:** Specifies whether or not the user can reorder the columns displayed in the list view by dragging and dropping the column headers. This is enabled by default.
- **Allow Filter:** Specifies whether or not the user can filter the items in the list. A filter icon is displayed at the bottom of the list view field when enabled. The user can click this icon to select filter options. Individual column definitions may be defined to prohibit filtering on those columns.

List View Header/Detail Pane Attributes

Using these attributes, a header label or a detail pane may be added to the list view field. Header label and Detail pane attributes are set to display additional information about the list as a whole or about the currently selected item in the list. The Header Label is a static line of text displayed above the list view, within the area given to the field. This text may be static, set via certain available format strings, or set via a rule. A rule referenced for this purpose is expected to return a string value and is evaluated in the context of the object displayed by the parent screen set.

The Detail Pane is redrawn each time a new object is selected in the list and almost always contains either format strings or is set via a rule's return value. Rules are evaluated in the context of the selected object in the list and are expected to return a string value. The detail pane drawn on the screen is a multi-line, read-only text box that may be scrolled horizontally or vertically if needed. The detail pane is drawn within the are given to the list view field and will reduce the amount of space for the list items.

Header

- **Header Label:** Specifies the header text for the list view. A common use for this header label is the total number of objects displayed in the list vs. the total number of objects in the collection, which may be different when a filter is enabled. The format strings used for this purpose are `%DisplayedCount` and `%TotalCount`.

Detail Pane

- **Detail Pane:** When true, a text box on the list view. The detail pane is updated each time the user changes their selection in the list view.
- **Position:** Controls where the detail pane is displayed on the screen in relation to the list control. This may below the list or to its right.
- **Size:** Sets the pixel size of the detail pane within the list view field. The default is 50. If the **Position** is "Bottom" the detail pane will span the width of the space given to the field and the **Size** will set its height. If the **Position** is "Right" the detail pane will span the height of the space given to the field and the **Size** will set its width.
- **Word Wrap:** When enabled, lines of text longer than the width of the detail pane will be wrapped to the next line. When disabled, text will continue off the detail pane. The user will need to scroll the detail pane to view the text.
- **Format:** Sets the values displayed in the detail pane. This pane can be set to a combination of static text and format strings, which take the form *`%propertyName`*. The *`propertyName`* is the name of a property defined within the selected object and will be updated with the value of that property each time a different object is selected. It may also be set to the return value of a rule, which is evaluated in the context of the selected object instance and is expected to return a string.

List View Scanner Attributes

The scanner attributes for a list view affect only those list view fields defined for a detail screen that is used by a scanner platform only when the detail screen is displayed on a client device with a barcode scanner. At least one column definition within the list view must be defined to support scan filtering.

A scanned value will be compared to the column(s) defined for scan filtering and only those matching this value will then be displayed. Actions may be executed automatically when a single record matches the scan filter, or when no records match.

- **Show Button:** This attribute specifies whether or not a button is displayed to activate the device’s barcode scanner.
- **Single Match Action:** Specifies what action is executed when a scanned barcode value matches one of the records displayed in the list view. The target of the action will always be the object instance found to match.
- **No Match Action:** Specifies what action is executed when the scan filter criteria does not match any records in the list. The target of the action is the parent object to the collection property displayed by the list view.
- **Label Types:** Specifies what barcode types are accepted by the Agentry Client. If no Label Type is specified, all types supported by the client device’s scanner will be supported. To restrict the label types, enter the name of each label type to support, separated by a comma.
- **Minimum Value:** The minimum number of characters accepted by the Agentry Client from the device scanner. If the value scanned in contains fewer characters, it will be ignored.
- **Maximum Value:** The maximum number of characters to be accepted by the Agentry Client from the device scanner. If the value scanned in contains more characters, it will be ignored.
- **Shortcut Key:** This attribute can define a shortcut key combination to activate the device’s barcode scanner. This shortcut cannot be the same as any other shortcut defined for the current screen or any system level shortcuts configured on the client device.

List View Column

A column definition defines what object property is displayed in a list control column. The column definition also controls behaviors such as formatting, sorting the list on the column, whether or not the column can be resized or moved, and whether or not the list can be filtered on the column. Columns may also be defined to execute an action via hyperlink control.

In addition to or in place of a property value, a column may also display an image definition as an icon, which can be different for each record based on a rule definition.

Column Attributes

- **Object Property:** Specifies the property to display in the column on the list view. Set this to None, to display either a value derived from a format string or only an icon image. Selecting both an Object Property and specifying an icon image will display both in the column.
- **Name:** Internal name for the column definition. This value must be unique among all columns definitions in the list view.
- **Label:** Specifies the label for the column header. This text is displayed at the top of the column on the Agentry Client to identify the contents of the column.
- **Enable Rule:** References a rule definition evaluated in the context of the object displayed by the screen set and expected to return a Boolean value. When the rule returns true, the

column is enabled and displayed on the Client. When it returns false, the column is disabled and not displayed.

- **Format:** Can contain a format string to display one or more property values from the object type being displayed by the list in a different format than the default for the property's data type. This text can also be set via a rule definition, where the expected return value is a string and is evaluated in the context of the object instance for the record in the list. To set the format attribute set the **Object Property** attribute must be set to None.
- **Icon Image:** References an Image definition within the application to specify an icon for the column. The image name can also be returned using a rule definition to dynamically determine the image to display for each record. This rule is evaluated in the context of the object instance for the record and is expected to return the name of an image definition as a string. Note that not using a rule for this attribute will display the same image for all records in the list
- **Column Width:** Specifies the initial size of the column on the client. The user can resize the columns if the list view definition has not disabled this behavior. If the user changes the width of a column, the new width is saved in the registry on the client device and will override the Column Width attribute.
- **List Filter:** Specifies if the column should be included in those listed in the filter dialog for the list. This attribute is ignored in filtering has been disabled for the list view.
- **Scanner Filter:** Enables scan filtering functionality for the column. When this attribute is enabled, the value scanned in by the device will be compared to the values of the column to create a filter. Multiple columns can be defined for this behavior. However, the values in the columns should be mutually exclusive. The order of the columns evaluated against the scanned value is undefined. This attribute is only supported for screens used by a scanner platform and displayed on a scanner-enabled device.
- **Hyperlink:** Specifying a hyperlink action enables each cell within the column to execute an action when the user single or double clicks on the hyperlink drawn in that column. The text of the hyperlink will be the value the column is defined to display. This functionality can include columns with images. Hyperlink contains two attributes:
  - **Hyperlink Action:** Specifies the action that will be executed when a user single-clicks a column in a populated row in the list.
  - **Hyperlink Target:** Specifies the target of the Hyperlink Action.

Password Validation

The password validation edit type requires users to enter their password on a detail screen. The value entered is validated against the password stored for the Agentry Client for the current user. The characters entered in this field are replaced with asterisks. This field edit type is used primarily with transaction authentication functionality.

The value entered in this field is validated against the user's password when the wizard or authentication screen set is advanced. If the value entered is not a valid password a message will be displayed. This message may be the default message provided by the Agentry Client, or it may be defined as a part of the screen field using the **Message** attribute.

Password Validation Attributes

The following attributes are specific to the password validation field edit type. These are in addition to the common field attributes:

- **Password Failure Message:** This attribute specifies the message to display to the user if the password entered in the field is invalid. This may be the default message, which is displayed when Auto is selected, or it may be a message entered in the text box for this attribute field.

Tile Edit

The tile edit field type displays object properties in a tiled view allowing for add and edit interaction without starting a wizard screen. For a given object, the properties of that object can be displayed in the tile edit view in tiles within that object instance. The layout of the tile edit is defined in a separate screen set and detail screen, used by the tile edit field. The values of a given object can be edited directly, and new object instances can also be added.

Prior to defining a field with this edit type, the screen set and the transaction it is to use must be defined. Both are required information when defining a new tile edit field. The screen set must be defined to display the transaction to capture the data. The screen set must contain a single detail screen displaying the properties from the transaction.

At run time this field type is displayed within its own detail screen. Within the field is then the single detail screen from the separate screen set. The fields of this detail screen are displayed within the tile edit field in the same manner in which they are laid out in the detail screen. The user can edit any fields defined in the separate detail screen that are not read-only. Within the tile edit field, if the detail screen displayed is larger than the space given to the field, a vertical scroll bar is displayed to allow the user to scroll up and down to display fields not immediately shown.

The tile edit field type does not support the following common field attributes:

- Object Property
- Read-only
- Format
- Change Focus
- Update Rule
- Special Value

Tile Edit Attributes

The following attributes are specific to the tile edit field type. These are in addition to the common field attributes:

- **Tile Edit Screen Set:** This attribute specifies the screen set to display for edits. This screen set should be defined to display the transaction definition specified in the Tile Edit Transaction attribute.

- **Tile Edit Transaction:** This attribute specifies the edit transaction instantiated to capture data entered by the user in the Tile Edit Screen Set. An instance of this transaction is created, applied, and saved as a pending transaction when the user enters data changes.
- **Tile Target:** This attribute specifies the object instance that is targeted by the transaction.
- **Modify Row Height By:** This attribute allows for all rows displayed on the screen within the Tile Edit field to be modified by the value set in this attribute.
- **Hide Buttons:** This attribute will hide the OK and Cancel buttons displayed when the tile is being edited. These are displayed by default. When hidden, values entered by the user are automatically applied, as is the define transaction, when the Tile Edit field no longer has the input focus.
- **In Progress Edit:** This attribute will enable the In Progress Edit style to be applied to the field when it is currently being edited and the changes it contains have not been applied. This is a visual indicator to the user that the Tile Edit field currently has the focus and is actively being edited.

Tile Display

The tile display edit type displays an object instance in a tiled view. The layout and appearance of the values is defined in a separate screen set and its detail screens. This separate screen set is used by the tile display field, with its detail screens displayed within the tile display field as a tab control.

Prior to defining a field with an edit type of tile display, the separate screen set it is to display must be defined. This screen set must be defined to display the object type desired for display in the tile display field. The separate screen set can contain a single detail screen. The fields of this detail screen display the property values of the selected object type.

When the detail screen containing the tile display field is displayed on the client, the separate screen set and its detail screen are displayed within the viewable area of the tile display field. These values are read-only.

The tile display edit type does not support the following general field attributes:

- Object Property
- Read-only
- Format
- Change Focus
- Update Rule
- Special Value

Tile Display Attributes

The following attributes are specific to the tile display edit type. These are in addition to the common field attributes:

- **Tile Display Screen Set:** This attribute specifies the screen set to display the object instance within the tile display field. This screen set can contain one detail screen. The fields of this detail screen are displayed within the tile display field. The screen set must be defined to display the object definition specified in the Tile Target attribute.

- **Tile Target:** This attribute specifies the object instance targeted by the tile display field. This object instance must be of the type the Tile Display Screen Set is defined to display.
- **Modify Row Height By:** This attribute specifies the rows within the screen being displayed by the Tile Display field be modified by the value set in this attribute.
- **Display Single Screen:** This attribute forces the Tile Display field to display only a single screen from the selected screen set. Otherwise each screen in the screen set is displayed within the Tile Display, with a tab control displayed for each screen.

Detail Screen Fields With Implicit Edit Types

In addition tot hose detail screen field edit types already addressed, there is a small handful of edit types which are implicitly set based on the data type of the property the field is defined to display. These implicit field edit types do not contain any edit type-specific attributes. The general field behaviors, e.g., position, size, read-only, etc., are defined just as any field would be. The edit type-specific behaviors are typically take driven by the definition of the property being displayed.

These edit types cannot be selected from the Edit Type attribute for the field definition. Instead, when a field is defined to display one of the property data types with which the field edit type corresponds, the field definition’s Edit Type should be left set to --Default--. As an example, when displaying a signature capture property type, the field to display this property will not contain a corresponding Signature Capture edit type. Rather, it is left set to an Edit Type of --Default--. At run time, the field displayed on the client will be a signature capture field, and the behavior of the field is driven by the definition of the signature property.

Signature

The signature field edit type allows for the entry of a signature on a client’s screen that is stored as a bitmap image. This is an implicit edit type in that it cannot be selected when defining a field definition. Any detail screen field with an edit type of “Default” and targeting a property with an data type signature will be a signature field. This field edit type has no additional attributes beyond those of the common field attributes. Much of the behavior of this field is dictated by the signature property it targets.

Action

An action defines navigation and user interaction for the Agentry Client. Actions are composed of a series of action steps of varying types. An action is defined to allow the user to interact with the application in some way.

The action defines the object its steps will act upon, any may also define whether or not users will be permitted to cancel the action once it has been executed, and also an optional separate action to execute if the action is cancelled.

The behavior of the action is dictated primarily by its child action step definitions. There are different types of action steps for different types of Agentry Client behaviors. Each action step defines a specific task to be performed on the Agentry Client.

Actions can be referenced by several different components of the user interface, such as buttons, list screens, and others. Whenever an action is executed it will be passed an object instance. This object instance is determined by the user interface component executing the action. The action must be defined for an object type. This object type for the action and the type of object passed to the action on the Agentry Client must be the same. The exception this is when the action is not defined for any object. Such actions are limited in use and normally pertain to performing transmits between the Agentry Client and Agentry Server, or actions that close screen sets but do not open others.

Action Child Definitions

Action Step: An action step defines a single task within an action that is a part of the overall action execution on the Agentry Client.

Action Attributes

- **Name:** This is the unique internal name for the action within the application project. This value must be unique among all actions defined within the same module.
- **Display Name:** This attribute contains the name displayed for the action on the Agentry Client.
- **Group:** This attribute specifies the group into which the action will be organized within the application project. This attribute has no impact on the action’s behavior at run time.
- **For Object:** This attribute specifies the object for which the action is defined. An instance of this object must be passed to the action by the Agentry Client user interface component executing the action. Therefore, both this attribute and UI component must have the same type of object defined or in scope when the action is executed.
- **Enable Rule:** This attribute references a rule definition called in the context of the object currently in scope on the user interface and is expected to return a Boolean value. When the rule returns true the action will be enabled and can be executed. When the returns false the action will be disabled and cannot be executed. Any buttons defined to execute a disabled action will be displayed as disabled controls.
- **Disable Cancel:** This attribute specifies whether or not users can cancel an action once its execution begins. This setting primarily affects the behavior of screen sets defined to display transactions or fetches and are displayed by the action. When Disable Cancel is set to true, screen sets will not contain a cancel button, preventing the user from canceling the action. When set to false (default) the wizards will contain a cancel button.
- **Cancel Action:** This attribute references another action within the same module to be executed when the parent action is canceled. The cancel action will be executed in the same context as the action that was executed first and then canceled by the user. The action selected here must exist prior to making a selection.

Action Step

An action step defines a single task within an action that is a part of the overall action execution on the Agentry Client. There are multiple action step types. Each type of step is defined for a different type of task. These can include navigation, transaction instantiation and display, transmit initiation, and other behaviors. Each action step type contains its own type-specific attributes.

The action step definition encapsulates a single task to be performed within the action as a whole. A given action can contain one or more action steps. Much of the client-side functionality and behavior that may be defined for a mobile application is exposed in the action steps.

Action Step Types

Following are the different types of action steps that may be defined:

- **Apply:** The apply action step type applies all transactions instantiated and completed before it in the same action.
- **Exit Application:** The exit application client action step will close the Agentry Client application when executed.
- **External Field Command:** The External Field Command action step issues a command to an ActiveX control when executed.
- **List Selection:** The List Selection action step type selects the specified row or item in the selected screen set and screen.
- **Message:** The message action step type displays a message screen on the Agentry Client to the user that can contain one or two buttons.
- **Navigation:** The navigation action step type displays an object screen set on the Agentry Client.
- **Open URL:** The Open URL action step type defines a URL to be opened by the client device’s web browser.
- **Print Report:** The print report action step type will print the defined report definition on a printer connected to the client device.
- **Save Tile Transactions:**
- **SubAction:** The SubAction action step type executes an action definition from within another action.
- **Transaction:** The transaction action step type instantiates a transaction on the Agentry Client and defines what screen set to display the transaction instance in.
- **Transmit:** The transmit action step type initiates communications between the Agentry Client and Agentry Server.
- **Windows Command:** The Windows command action step type executes a command on the client device.

Action Step Type: Apply

The apply action step type applies all transactions instantiated and completed before it in the same action. An apply step is required in any action containing one or more transaction steps in order for those transactions to affect their target objects and to be saved on the Client.

The apply step definition itself contains no attributes other than a name. However, it is an important part of transaction behavior within the Agentry Client. The absence of an apply step within an action that also includes a transaction step will result in the transaction not being save or applied on the Agentry Client.

The intended purpose of the separate apply step to apply and save a transaction is to allow for actions continuing multiple transaction steps followed by a single apply step. This allows for the requirement that multiple transactions be instantiated and completed by a user within a single action, and to not save any data until all transactions have been finished.

Apply Step Attributes

This action step type has only a **Name** attribute, which must be unique among all steps within the same parent action.

Action Step Type: Exit Application

The exit application client action step will close the Agentry Client application when executed. When this step is performed within an action it will result in the same behavior as if the Exit menu item is selected in the File menu of the Client. A step of this type should only be defined as the last step to be executed within an action, as no other action steps that follow it will be executed.

The primary purpose of this action step type is to support the true and clean shutdown of the Agentry Client when running on client devices that do not support this behavior easily. Many client devices and the shells they run will no truly exit an application when the user clicks the title bar close button. Rather, the application is simply hidden from view. It remains running on the client device. Additionally, the application does not exhibit any behaviors defined to occur when the application exits, such as check for, and notifying the user of any pending transactions.

To support a cleaner shut down, the users should always be instructed to use either the Exit menu item in the client's File menu, or to execute an action defined with an Exit Application action step type.

Exit Application Step Type Attributes

This action step type has only a **Name** attribute, which must be unique among all steps within the same parent action.

Action Step Type: External Field Command

The External Field Command action step issues a command to an ActiveX control when executed. It references the External Field - ActiveX Control field to specify the control to which the command is to be issued. The action step passes the value of the defined command string to the ActiveX control, which is then responsible for receiving and processing the string command accordingly.

The defined command string within this action step type is passed by the Agentry Client to the ActiveX control through the expected method `AgentryExecuteCommand`. This method should be implemented to process the provided command string in the manner deemed appropriate for that control.

External Field Command Step Attributes

- **Step Name:** This attribute contains the unique internal name of the action step definition. This must be unique among all steps within the same parent action.
- **Screen Set:** This attribute specifies the screen set containing the detail screen within which the External Field - ActiveX Control field is defined. Valid selections for this attribute include any screen set defined to display an object definition. Screen sets for transactions and fetches are not valid.
- **Screen:** This attribute specifies the detail screen containing the External Field - ActiveX Control field.
- **External Control:** The External Field - ActiveX Control detail screen field that references the ActiveX control to which the command string is to be issued.
- **Command:** The string to be passed to the ActiveX control's `AgentryExecuteCommand` method. This attribute value can be entered into the attribute field directly, or can be set to the return from a rule definition. A rule referenced by this attribute is evaluated in a string context and in the context of the action to which the action step is being added and the object for which that action is defined.

Action Step Type: List Selection

The List Selection action step type selects the specified row or item in the selected screen set and screen. The specific list control on the screen must also be specified if more than one type of list field is defined for that screen. The action allows for the specification of record to select by one of several options, as described in the Select Rows attribute of the field definition.

List Selection Step Attributes

- **Step Name:** This attribute contains the unique internal name of the action step definition. This must be unique among all steps within the same parent action.
- **Screen Set:** This attribute specifies the screen set containing the screen and list in which a selection is to be made.

- **Screen:** This attribute specifies the screen containing the list in which a selection is to be made.
- **List Control:** This attribute specifies the list on the selected detail screen in which a selection is to be made.
- **Select Rows:** This attribute specifies how the item in the list is to be selected. Options for this attribute include:
  - **By Position:** Selecting this option enables the Position attribute, where the position number of the item to be selected can be specified. This is a numeric value that must be one or greater and indicates the item to select from either the top or bottom of the list.
  - **By Rule:** This option specifies a rule is to be used to determine the item to be selected. The object being listed sets the context for the rule being evaluated, with either a true or false value returned by the rule. The first item for which the rule returns true will be the one selected in the list.
  - **First Row:** The first row in the list based on it’s current sort order.
  - **Last Row:** The last row in the list based on it’s current sort order.
  - **Next Row:** The row immediately following the row currently selected in the list.
  - **None (Clear selection):** This clears the selection state of any items that may be currently selected in the list.
  - **Previous Row:** The row immediately preceding the row currently selected in the list.

Action Step Type: Message

The message action step type displays a message screen on the Agentry Client. This screen can contain a defined title, message text, and either an OK or an OK and Cancel button. When a Cancel button is clicked in a message screen the parent action of the step is canceled. No subsequent steps within the action will be executed.

This step type can provide the user with the ability to cancel an action based on some decision. This step type is commonly used within actions that will delete an object instance on the client. A message step with two buttons can be defined to confirm the delete of the object prior to executing the transaction step that will delete it.

When a message step is displayed with only a single button, the user will not be able to cancel the action within the message displayed. Rather, the single button is displayed for the user to confirm they have read the message. Once clicked, the action will continue execution with its next defined action step.

Message Step Attributes

- **Step Name:** This attribute contains the unique name for the action step. This value must be unique among all steps within the same parent action.
- **Caption:** This attribute contains the text to display in the title bar of the message dialog displayed by the Message Step.
- **Message Text:** This attribute contains the text to display in the main portion of the message dialog displayed by the Message Step. Format strings may be used within this

text, or the entire message may be built and returned by a rule. A rule definition referenced here is evaluated in the context of the object passed to the step by the action. The rule is expected to return a string value.

- **OK Label:** This attribute contains the text to label the OK button in the message dialog. Regardless of the label, clicking this button will always confirm the message, or be considered a positive response to the message, continuing execution of the action.
- **Cancel Label:** This attribute can enable or disable the cancel button behavior in the message log. When disabled, no cancel button is displayed. When enabled, the cancel button will be displayed and this attribute also then contains the label for that button. Regardless of the label text, clicking this button will always be considered a negative response and cancel the parent action's execution.

Action Step Type: Navigation

The navigation action step type displays an object screen set on the Agentry Client. It includes optional definable behaviors to specify the screen and control on the screen to which the initial focus is set. It may also be defined to close the previous screen set displayed. Screen sets may be defined to display an object, transaction, or fetch. A navigation action step is defined to display only those screen sets defined to display an object.

When selecting a specific screen within a screen set to be the first one displayed, you will typically select any of the screens not at position one within the screen set. The screen at position one within the screen is displayed first by default. If the navigation step displays a screen set with multiple platforms, and the selected screen definition is not used by one or more of the platforms, the screen at position one within the screen set is displayed first on those device platforms.

Similar behavior is exhibited when a specific field is selected within a detail screen to have the initial focus. If the field does not exist on a screen for a given platform, the default focus field, as defined within the screen, will contain the initial focus.

If the initial screen defined to be displayed is a list screen, the optional behavior of selecting one or more rows by default within that list can be defined. There are several options for selecting the rows, including: by position within the list; conditionally based on a rule; or to not select any record within the list.

The definable behavior of closing the previous screen set, or closing all open screen sets, can be used when navigating from one screen set to the next. However, it is recommended that closing the previous screen behavior not be defined when navigating from a module main screen set, as closing the main screen set is considered undesirable user interface behavior except in rare circumstances. Closing all open screen sets will never close the current module's main screen set.

Closing the previous screen set or all non-main screen sets may also be defined in a navigation step that does not display a new screen set. This definition option results in the user being returned to the previously displayed screen set, i.e. the one from which they navigated to the current screen set, or the module main screen set. These behaviors are supported to ensure that a screen set and its screens are truly destroyed when the user wishes to close them. Certain client devices and their shells do not close a screen once opened. Rather, when a user clicks the close button (or sometimes an OK button) displayed in a title bar, the screen itself is simply hidden, but still remains in the background. The navigation step will close a screen set by destroying the screen object in memory. Closing all open screen sets other than the module main screen set provides an easy means of returning the user to the module main screen set if they are multiple levels deep into the application's screen flow.

Navigation Step Attributes

- **Step Name:** Contains the unique internal name of the action step definition. This must be unique among all steps within the same parent action.
- **Screen Set:** Specifies the object screen to be displayed by the navigation step. This screen must be defined and exist within the module prior to defining the navigation step. This may be set to "Do No Display Screen Set," which will also set the **Close Screen** attribute to "Close the screen you are leaving when this navigation step runs."
- **Close Screen:** Specifies that the screen set currently displayed when the action is executed be closed, or alternately that all screen sets other than the module main screen set be closed. The default setting for this attribute is to not close any screen sets. Valid options for this attribute include:
  - *None:* Do not close any screen sets. Display the screen set defined in the **Screen Set** attribute.
  - *Close the screen you are leaving when this navigation step runs:* The currently displayed screen set will be closed when the action step is executed. The screen set defined in the **Screen Set** attribute, if any, will then be displayed.
  - *Close all screens except main when this navigation step runs:* All open screen sets except the module main screen set will be closed. The screen set defined in the **Screen Set** attribute, if any, will then be displayed. In an application with multiple modules, the main screen set for all modules except the current module will be closed.
- **Screen:** Specifies the screen within the screen set to display first. By default, the first screen displayed is the screen at position one within the screen set. If a detail screen is selected, the Initial Focus attribute field is enabled. If a list screen is selected, the Select Rows attribute field is enabled.
- **Initial Focus:** Available only when the Screen attribute is set to a detail screen. The Initial Focus can specify a field on the selected detail screen that will have the initial focus when the navigation step displays the screen set. By default, the field at position one within the screen will have the initial focus.
- **Select Rows:** Available only when the Screen attribute is set to a list screen. The following are the options for this attribute, each of which specifies which row or rows within the list screen should be selected automatically when the list screen is displayed:
  - *All Rows:* This selection will result in all rows in the list screen being selected initially. This selection is only applicable to list screens for which the multi-row select behavior has been enabled.

  - *Auto:* This selection is the default and will not change the selected row on the list screen.
  - *By Position:* This selection will enable the Position attribute field where the selected row in the list is specified by its position within the list.
  - *By Rule:* This selection specifies that the initially selected row or rows in the list will be determined by a rule. When this option is selected, the Row attribute field will be enabled where the rule can be selected.
  - *First Row:* This selection specifies that the first row in the list will be selected. This is always the first row from the top of the list.
  - *Last Row:* This selection specifies that the last row in the list will be selected. This is always the last row from the bottom of the list.
  - *None (Clear Selection):* This selection specifies that no rows will be selected in the list.
- **Position:** Available when the Select Rows attribute is set to By Position. The Position attribute can then be set to a numerical value specifying the row at this position will be selected. As options to this attribute, the position can be determined by counting from the top of the list down or counting from the bottom up.
- **Rule:** Available when the Select Rows attribute is set to By Rule. The Rule attribute can then be set to the First, Last, or All Rows, where the selected Rule returns either True or False. All rows is applicable only to list screens for which the multi-row select behavior has been enabled. The rule referenced here is evaluated once for and in the context of each object listed on the screen. It is expected to return a Boolean value.

Action Step Type: Open URL

The Open URL action step type defines a URL to be opened by the client device’s web browser. As a option to this defined URL it is possible to also specify one or more URL parameters to be passed to the URL. These parameters can be set via a rule, which allows for the specification of dynamic values obtained from the applications data.

Child Definitions

- URL Parameters: Contains a value to be passed to the defined URL as a parameter (such as a CGI argument or similar). This can be set via a rule definition to expose access to any value within the mobile application.

Open URL Attributes

- **Step Name:** Contains the unique internal name of the action step definition. This must be unique among all steps within the same parent action.
- **URL:** Specifies the URL to be passed to the client device’s default web browser. This can be either a constant value set directly in the field, or returned by a rule definition.

Action Step Type: Print Report

The print report action step type will print the defined report definition on a printer connected to the client device. This step can control which objects for the report are printed via an include rule. It can also be defined to allow the user to skip printing the report. At least one report definition must exist within the same module before a print report action step can be defined.

Print Report Step Attributes

- **Step Name:** This attribute contains the unique internal name of the action step definition. This value must be unique among all steps within the same action.
- **Report:** This attribute references the report to be printed by the action step. The report definition must be defined prior to selecting it for this attribute.
- **Include Rule:** This attribute allows for optionally including only certain objects within the collection targeted by the report being printed by the action step. When a rule is referenced here it is evaluated once for, and in the context of each object instance in the report's targeted collection. The rule is expected to return a Boolean value. Only those objects for which the rule returns true will be printed in the report.
- **Allow Skip:** This attribute can allow the user to skip printing the report. When this attribute is set to false (default) the report will always be printed. When set to true, the user will be prompted to continue with the print job or to cancel.

Action Step Type: Save Tile Transactions

The Save Tile Transactions action step applies all transactions begun in a tile edit or tile list detail screen field that have not been applied. This can occur based on the overall screen flow and navigational options defined within the application project.

Save Tile Transactions Action Step Attributes

- **Step Name:** This attribute contains the unique internal name of the action step definition. This value must be unique among all steps within the same action.
- **Save Option:** This attribute specifies which transactions are to be applied when the step is executed. The options include:
  - **Active Screen Set:** Any unapplied transactions from any tile controls on any screen in the current screen set.
  - **All Screen Sets:** Any unapplied transactions from any tile controls on any screen in any screen set within the current module.

Action Step Type: SubAction

The SubAction action step type executes an action definition from within another action. When the sub-action has completed execution the parent action will continue. A SubAction step can execute an action once or iteratively based on various available criteria. A SubAction step is also used to execute an action in a different module.

The SubAction step type supports modularity within the actions of an application, providing for the reuse of actions that provide behaviors applicable to multiple areas of functionality. SubAction steps are also the primary means by which iterative processing can be implemented within the client application's behavior. This step type is also the primary means of providing cross-module functionality. Using a SubAction step an action in one module may be executed from an action in another module.

A primary part of a SubAction step's definition is the object the action it executes targets. This object should normally be within the context of the parent action's object. As an example, if the parent action is defined for Object A, which contains a collection of Object B, the SubAction step can target an instance of Object B within that collection. The exception to this is cross-module action execution.

To execute an action across modules, the target object for the SubAction step must be an object defined in the other module. When an object from a different module is defined as the target, the actions that may be selected for the SubAction step will be those defined in that module. Execution of the parent action on the Agentry Client will then result in the action in the second module being executed as defined. The parent action will then proceed as defined after the SubAction step has completed execution.

SubAction Step Attributes

General Attributes

- **Name:** Contains the unique internal name for the step definition. This value must be unique among all steps within the same parent action.
- **Execution Type:** Specifies how the sub-action should be executed. There are several options available for this attribute, many of which providing iterative behavior. When one of these selections is chosen, the SubAction step is referred to as a Looping SubAction step. Lopping SubAction steps will have additional attributes that will differ depending on how the SubAction step loops. Following are the available items for this selection.
  - *Always - Execute until stopped:* This selection will define the SubAction step to execute repeatedly until the user explicitly ends the processing. This item should only be selected when the SubAction executes and action that allows the user to either cancel or finish the processing, normally within a transaction wizard screen set.
  - *Execute Once:* This selection will execute the defined action a single time when the SubAction step is executed.
  - *Execute once if rule is true:* This selection will execute the defined action a single time only when the rule referenced in the **Execution Rule** attribute returns true. If the rule returns false, the defined sub-action will not be executed and the parent action will continue execution as defined.
  - *Execute until rule is false:* This selection will execute the defined action until the rule referenced in the **Execution Rule** attribute returns false. This rule will be evaluated after each iteration of the sub-action. This behavior means the SubAction step will always execute the defined action at least once, as the rule will not be evaluated until after execution has completed.
  - *Execute while rule is true:* This selection will execute the defined action while the rule referenced in the **Execution Rule** attribute returns true. This rule will be evaluated before the first iteration of the sub-action and before each additional iteration. This

behavior means the SubAction step may or may not execute the defined action, as the rule will be evaluated to determine if the sub-action should be executed.

- *Loop over collection:* This selection will execute the sub-action once for each object instance referenced in the **Collection** attribute. This may be limited by referencing a rule in the **Execution Rule** attribute. In this case, the **Execution Rule** will be evaluated once for, and in the context of each object instance in the collection. The rule is expected to return a Boolean value. The sub-action will then only be executed for each object instance where the rule returns true.
- *Loop over list screen:* This selection will execute the sub-action once for each object listed in the current list screen. This may be limited by referencing a rule n the **Execution Rule** attribute. In this case, the **Execution Rule** will be evaluated once for, and in the context of each object currently displayed in the list screen. The rule is expected to return a Boolean value. The sub-action will then only be executed for each listed object instance where the rule returns true.
- *Loop over selected list screen objects:* This selection will execute the sub-action once for each selected object in the current list screen. This selection is provided in support of the multi-select behavior that may be enabled for list screens. If no items are selected in the list screen, the sub-action will not be executed by the SubAction step.

- **Collection:** This attribute is only enabled when the **Execution Type** is set to “Loop over collection.” The **Collection** attribute references the object collection property the SubAction step is to loop over.
- **Execution Rule:** This attribute is enabled when the **Execution Type** is set to “Loop over collection,” “Loop over list screen,” or to one of the execution types involving a rule. The **Execution Rule** references the rule definition to be evaluated to determine the execution behavior of the sub-action.
- **Act on Object:** This attribute references the object instance the sub-action is to target. This selection is normally a child object to the object for which the parent action is defined, or an instance of the object type for the parent object. It may also be an object defined in another module. When this last type of object is selected, the available items listed for the Actions attribute will those actions within the same module as the selected object.
- **Action:** This attribute references the action the SubAction step will execute as a sub-action to the step’s parent action. The action selected here must be defined prior to the definition of the SubAction step. The selected action must be defined for the object type selected in the **Act on Object** attribute.
- **Begin Loop with Selection:** When the Execution Type is set to Loop over displayed list items, this attribute is enabled allowing for the specification of the first item to be executed on. When selected, the first item is the one currently selected in the list. When not specified, the first item is the one at the beginning of the list.

Looping Attributes - These attributes are available only when the Execution Type attribute is set to one of the iterative options. Depending on the type of iteration, different attributes listed here will be enabled or disabled.

- **Dialog:** This attribute is available only when the **Execution Type** is set to “Loop over collection” or “Loop over list screen.” The Dialog attribute specifies whether or not to display a message when there are no items for the SubAction to loop over. This can occur if the selected collection contains no object instances, or if the list screen currently lists no items. When Dialog is set to true, the Dialog Message attribute will be enabled allowing for the definition of the message to display.

- **Dialog Message:** This attribute is available only when the Dialog attribute is set to true. The Dialog Message can contain the message text to display when the item to loop over is empty. The default message displayed is “No valid records found.”
- **Back Up:** This attribute specifies whether to complete the SubAction step when the user clicks the Back button in a screen set displaying a transaction or fetch.
- **Allow Done:** This attribute specifies whether to display a Done button in last screen of a wizard screen set. The Done button differs from the standard Finish button in that the Done button will break out of the SubAction’s loop and return execution control to the parent action. When a SubAction step’s **Execution Type** is set to “Always - Execute until stopped,” the **Allow Done** attribute should be set to true. In other looping SubAction steps, the Done button will allow the user to end the loop regardless of any other constraints related to the looping behavior.

Action Step Type: Transaction

The transaction step type instantiates a transaction on the Agentry Client. A transaction step also defines the screen set in which the transaction instance should be displayed, if any. A transaction step can also define a target for the transaction as well as a sub-action to execute after the transaction has been completed.

A transaction step can define a transaction to be instantiated but not displayed in a screen set. This is a common occurrence when the transaction is a Delete transaction type. Other transaction types may also be defined in this manner if it is not necessary to capture any data from the user for the transaction. To not display a transaction instance, the Screen Set attribute should be set to No Screen Set.

The target object and target property for the transaction may be set in the transaction step. By default, the target object is passed to the transaction step from the action. The transaction step can then change the target to a different object, provided it is a valid item within the context of the object passed in by the action. In many cases it is not necessary to change the target of the transaction within the transaction action step. This target is normally set when there are multiple transaction steps within the same action, and one or more of those transactions is defined for the object type for which the action is defined. In the situations where the target is specified, it is normally a child object to the object type for which the parent action is defined.

The option of defining a SubAction to execute when the transaction has been completed provides a means of executing a second action from the transaction step. This sub-action will be executed only when the transaction step completes the transaction processing; i.e., if the user clicks cancel in the wizard screen set displaying the transaction the defined SubAction will not be executed. The action executed as a SubAction to the transaction step is executed as the last task of the transaction step. This results in the condition that the sub-action is executed before any apply step within the parent action. The transaction will, therefore, not yet be applied to the object it targets. This will impact the current data values that will be accessible within that target object and may, therefore, impact how the sub-action itself is defined, as well as how any other definitions it references will be defined, specifically as to which data values the sub-action will have access.

Transaction Step Attributes

- **Step Name:** This attribute contains the unique internal name for the step definition. This attribute must be unique among all step definitions within the same parent action.
- **Transaction:** This attribute references the transaction to be instantiated by the transaction step. The transaction selected here must exist prior to defining the transaction step.
- **Screen Set:** This attribute references the screen set in which the transaction will be displayed. Only screen sets defined to display transactions may be selected for this attribute. The screen set must exist prior to defining the transaction step.
- **Target Object:** This attribute can be set to change the target of the transaction from the object instance passed in by the action, to a different object instance. This attribute is optional and, if left set to its default, the target of the transaction will be the object instance passed to the transaction step by the action. The selection of a Target Object should be to an object instance that is easily related to the action's object instance wherever possible.
- *Target Property:* This attribute is obsolete in current versions of the Agentry Mobile Platform. It exists as a result of behaviors exhibited in early versions of the platform and in current implementations is no longer necessary. It is still provided for backwards compatibility and may be deprecated in a future release.
- **SubAction:** This attribute can specify an action to be executed as a sub-action to the parent action of the transaction step. The transaction selected for this attribute will only be executed when the transaction instantiated by the transaction step is completed successfully. Note that this sub-action is executed prior to the transaction being applied.

Action Step Type: Transmit

The transmit action step type initiates communications between the Agentry Client and Agentry Server. This includes displaying the Client's built-in transmit dialog where users can select a transmit configuration and begin the transmission. Alternately the transmit step can be defined to begin the transmission automatically and to hide the transmit dialog unless an error is encountered. The transmit step also defines the non-main fetches to be processed, if any.

Each module will contain at least on action with a defined transmit step. Additional actions may defined as needed that include transmit steps for various purposes.

The transmit step can be defined to start transmission between the client and server automatically. The default behavior is to display the Client's Transmit Dialog, where the user can select a transmit configuration and then start the transmit. When the transmit step is defined to automatically start the transmission, the transmit will begin when the step is executed. The transmit configuration used will be the last one selected by the user; optionally the transmit step can define the transmit configuration to be used.

The transmit step can also be defined to automatically finish the transmit. By default when a transmit is complete, the user must close the Transmit Dialog by clicking the finish button. The transmit step can close this screen automatically when the transmit completes successfully.

If the transmit step is defined to automatically start and finish the transmit, it can also be defined to hide the Transmit Dialog. In this case, the dialog will not be displayed to the user unless an error occurs during the transmission.

The transmit step is where non-main fetches must be selected for processing. The main fetches of a module will always be run when a transmit occurs. A non-main fetch must be explicitly selected in a transmit step and will only be processed when that transmit step is executed.

A transmit step can be defined to skip fetch processing altogether. This can only be defined when the step is first defined to use a transmit configuration for which real-time communications have been defined. The transmit step can then be defined to simply connect the user to the Server, process any pending transactions, and the remain connected to receive push data and/or to allow for background sending. Note that this behavior can negatively impact the push functionality if the push is defined to use exchange data initially generated by a fetch. This exchange data will not exist as the fetch will not be processed.

Transmit Step Attributes

- **Step Name:** Contains the unique internal name of the step definition. This value must be unique among all step definitions within the same parent action.
- **Transmit Config:** This attribute can be set to a specific transmit configuration within the application. If a transmit configuration is selected here, the user will not be able to change the transmit configuration when the transmit step is executed.
- **On-line/Off-line:** This attribute specifies whether to change the on-line state of the client when the transmit step is executed. This will override any selection the user makes on the client for this state. It will also override the on-line state of the client if it is set to Off-line as the result of a disconnect.
- **Initiate Asynchronous Transmit Only:** This attribute is only available if the Transmit Config attribute is set to a transmit configuration defined to support real-time communications. If this attribute is set to true, no fetches will be processed during the transmit. Pending transactions will be sent to the Server to be processed and complex table and data table definitions will be synchronized.
- **Allow user to skip:** This attribute allows the user to skip the transmit. This is normally only set when the parent action contains multiple step definitions, including the transmit step. The user may skip the transmit behavior when this attribute is set to true and when the client is in an Off-line state.
- **Automatically start transmission:** When this attribute is true, the transmission between the client and server will begin automatically when the transmit step is executed. The default is to require the user to click the Start button in the Transmit Dialog to being the transmission.

- **Automatically finish transmission:** When this attribute is true, the Transmit Dialog will be closed automatically when the transmit has completed successfully. The default is to require the user to click the Finish button in this dialog when the transmission has completed.
- **Hide transmission screen:** This attribute is available only when the Automatically start transmission and Automatically finish transmission attribute are both true. The Hide transmission screen can then be set to true, which will result in the Transmit Dialog not being displayed when the transmit step is executed. The Transmit Dialog is always displayed if an error occurs during transmission, regardless of this attribute setting.
- **Hide Screen Timeout:** This attribute is available only if the Hide transmission screen attribute is set to true. This timeout value is set in minutes and seconds. If the transmission takes longer than the duration entered in Hide Screen Timeout, the Transmit Dialog will be displayed to the user indicating the progress of the transmission. This timeout value should be selected based on the typical duration of a transmit for the application.

Action Step Type: Windows Command

The Windows command action step type executes a command on the client device. This step type can be defined to wait for the command to complete execution, to capture the return code of the external process, and to display an error message based on a non-zero return code. The Windows command step type is also used to display external files on the client device by setting the full path and file name as the command. This will result in the file being opened by the default application for the file type.

The command executed by the Windows command step must include the full path and file name of the executable to be run or file to be opened. When waiting for the command to return, the step will block action execution until the command completes, or until the defined wait period expires. An expired wait periods is treated as a timeout error by the Windows command step.

Additional error conditions include a non-zero return value by the command to the operating system. If a non-zero value is returned, the Windows command step will treat this as an error condition.

The timeout and the error conditions each have associated messages that may be displayed as defined in the Windows command step. This step type allows for providing the user with the option to continue or cancel the parent action’s execution. Alternately, the step can be defined to not allow action execution to continue, or to not allow the user to cancel the action regardless of the error.

Windows Command Step Attributes

- **Step Name:** Contains the unique internal name for the step definition. This value must be unique among all step definitions within the same action.
- **Command Line:** This attribute contains the command to execute or pass to the operating system. This may be a string value set within the attribute field, or it may be returned from a rule definition. The command may contain one or more format strings consisting of the

property names for the object passed to the command step form the action. These format strings take the form `%propertyName`. Note for properties of type External Data, the format string will return the full path and file name of the file referenced by the property. If a rule is referenced for the command, it may not return a string containing format strings. The rule is evaluated in the context of the object passed to the Windows command step by the action. The rule is expected to return a string value.

- **Wait:** This attribute specifies whether the Windows command step should wait for the command it executes to return. The default is to not wait, in which case the command line will be executed and the step will end execution. The timeout message will not be displayed. The only error captured by the step will be if the command line cannot be executed by the operating system, e.g. if the command referenced does not exist, or the file cannot be found. When the Wait attribute is set to true, the Wait Period Limit attribute is enabled.
- **Wait Period Limit:** This attribute is enabled only when the **Wait** attribute is set to true. In this case, the **Wait Period Limit** specifies the duration of time the Windows command step should wait for the command it executes to complete processing and return. If this duration is exceeded without a return from the command, the step's defined **Timeout Message** will be displayed.
- **Error Message:** This attribute contains the text to display when an error occurs. This may be displayed if the command fails to execute, or if the command returns a non-zero value after completing execution.
- **Timeout Message:** This attribute contains the text to display when the Wait Period Limit is exceeded without a return from the command executed by the step. This behavior also requires the **Wait** attribute to be set to true.
- **Continue Label:** This attribute contains the label for the Continue button that is a part of the dialog that displays the Error Message and Timeout Message. At run time, when this button is clicked the Windows command step will complete execution and the action will execute the next defined step. This button may be hidden by selecting the option Not Allowed, preventing the user from allowing the action to continue the action's execution when an error occurs executing the defined command.
- **Cancel Label:** This attribute contains the label for the Cancel button that is a part of the dialog that displays the Error Message and Timeout Message. At run time, when this button is clicked the Windows command step will complete execution and the parent action will be canceled. This button may be hidden by selecting the option Not Allowed, preventing the user from cancelling the action when an error occurs executing the defined command.

Report

A report defines a printed tabular format for the contents of an object collection on the Agentry Client. Reports can be generated for any object collection within the application data. A report can then be printed on the client device, provided it is equipped with a printer.

The report definition defines the object collection for the report and the property values for the collection's object type to include in the printed report. The report definition does not include any behaviors related to when to print the report. To print a report on the Agentry Client the action step type Print Report must be defined within an action.

A report defines the point size for the values it contains. It can also define the header and footer text to display in the report. Three separate header and footer values may be defined to be displayed on the left, center, and right of the page across the top and bottom of the report. Separate point sizes may be defined for the header and footer text. Note that the header and footer within the report definition are not the same as the report column headers. They are intended for general information about the report as a whole, not to label individual values within the report.

The child definition Report Columns defines the which properties to display from the object type within the target collection, as well as the column header labels within the report table. The order of the columns within the report definition will specify the order in which the columns are printed in the report from left to right.

Report Child Definitions

Report Column: A report column defines which property values are listed in the corresponding printed column of a report.

Report Attributes

General Attributes

- **Name:** Contains the unique internal name for the report definition. This must be unique among all reports within the same module.
- **Display Name:** Contains the default name for the report definition displayed on the Client.
- **For Object:** This attribute references the parent object of the collection for which the report will be generated.
- **Collection:** This attribute references the object collection property whose contents will be printed in the report.
- **Point Size:** This attribute specifies the font point size for the data printed in the report. This excludes the report header and footer, which specify their own point sizes.
- **Gridlines:** This attribute specifies whether or not to print grid lines in the report to separate columns and rows in the table. When set to true these lines will be printed in the report table. When false they will be omitted.

Header/Footer Attributes

- **Left, Center, and Right Text:** These three text boxes within the definition contain the text to display at the left, center and right sides of the report page. These values are not column labels, but are intended for general information to display in the header and/or footer of the report.
- **Point Size:** This attribute specifies the font point size of the header or footer text.
- **Bold:** This attribute specifies whether or not to display the text in bold. When true, the text will be in bold. When false it will not be.

Report Column

A report column defines which property values are listed in the corresponding printed column of a report. The column definition includes attributes for formatting the values of the column and the order of the columns within the report.

Each column defines a property of the object type in the parent reports target collection to be printed in the report. Included in the column definition is the label for the column in the report table. Basic formatting can also be defined for the column, including whether or not the column label should be in bold text, whether or not to word wrap the text within the column, alignment of the values within the column, and the width of the column as a whole.

As an alternative to selecting a property whose value will be displayed in the column, format strings or format text may be specified. To make use of this behavior a column should not be selected, but rather the Format attribute should be set to specify the value to be displayed for each object. This attribute may contain format strings referencing the properties of the object, as well as plain text. This attribute may also be set via the return value of a rule, which will be expected to build the entire string to be displayed in the column for each object.

Report Column Attributes

- **Name:** Contains the unique internal name for the report column definition. This value must be unique among report columns within the same report.
- **Label:** Contains the column label for the header row of the report table.
- **Object Property:** This attribute references the object property to be printed in the column for each object instance in the target collection.
- **Bold Label:** This attribute specifies whether or not he label for the column should be printed in bold text. When true the label will be printed with bold text.
- **Wrap:** This attribute specifies whether or not the values of the column should be word wrapped. If true the values printed in the report will be word wrapped to fit in the space of the column. If false, the column width will be expanded to allow for the size of the text.
- **Alignment:** This attribute specifies the alignment of the text within the column. The options for this attribute are “Left Justified,” “Right Justified,” or “Centered.”
- **Column Width:** This attribute specifies the width of the column. The units for this attribute are the number of average sized characters. If left set to Auto, the width of the column will be set by evenly spacing all Auto Width columns within the report, after space is allocated for all columns with a defined width.
- **Format:** This attribute can contain a combination of format strings and standard text to specify the format of the values printed in the column. Alternately the value printed in the column can be the return from a rule definition. If a rule is used for this attribute, it will be evaluated once for, and the context of each object instance within the reports target collection property. It is expected to return a string value. If this attribute is set either format text or a rule definition, the **Object Property** attribute should be set to None.

Rule Function Terms Overview

Rule functions terms are the heart of most rule definitions within an application. While there are situations where a rule may be defined to contain a single rule term that returns the value of a global or other such data definition type, most rules are more complex than this and consist of multiple function calls.

Most rule functions take one or more arguments, each of which contains a data value for the function. When the function is evaluated, these data values are processed in some manner. The result of this processing is a single return value that is passed to the function's caller. A function will always provide the caller with a value in the data type the caller asks for. Not all function support all data types for their return values. If a data type is not one supported by the function, that function will return the null-equivalent of that data type.

There are over a hundred different functions available for a rule definition. These are organized into Function Categories. These categories denote the general types of behavior for the functions. The rule editor presents the functions to the developer organized into one of these categories.

- **Conversion Functions** - Conversion functions set the context of a given term to a specified data type. A conversion function supports all return types within the rule definition. The names of conversion functions dictate what data type they will set for the context of a function call.
- **Logical Functions** - Logical functions are those that provide the comparison and decision making functionality to a rule. This includes if-then-else and comparison operations and behaviors.
- **Mathematical Functions** - Mathematical functions provide math operations to rules. This includes addition, subtraction, multiplication, division, and modulus operations, as well other mathematical functions, such as rounding, and working with significant digits.
- **Property Functions** - Property functions are those that operate on properties, usually of a certain data type. Most property functions are provided for the intended purpose of working with a given type of property, such as an object collection or external data property.
- **String Functions** - String functions provide behaviors for manipulating string values, including concatenation and parsing operations, string search and replacement, and other string-related operations.
- **System Functions** - System functions are those that provide access to information about the Agentry Client's host system, or information that is general to the client. This can be information such as the system's time and date, or the user ID of the current user. This category also includes functions to access hardware components of the client device such as barcode scanners and GPS units.

- **Table Functions** - The table functions provide access to the records of complex table and data tables stored on the Agentry Client.

Note that the function categories do not directly impact where a function can be used, or which rules can use a given function. The categories are an organizational aid built into the rule editor to aid the developer in locating the rule function that is needed.

Conversion Functions for Rules

The Conversion functions category of rule function terms provide the means for changing the context in which a function or data term within a rule is called. Within this category of conversion functions there is one function for the integral number, string, and property data types. The decimal number data type has two conversion functions, one of which is for use with significant digit math.

The name for each conversion function represents the data type to which it will set the context of the term that is its argument. The function term name, then, does not represent the data type *to* which a value will be converted, but the data type *from* which a value will be converted. Each conversion function supports all return types.

Conversion functions are most commonly used when it is necessary to obtain the return value of a function that may exhibit different behaviors in different contexts, or when the desired return data type does not match the supported return type of a given function. The caveat to this is that the conversion desired is type safe.

An example of this is the string function @FIND. This function searches a source string for a given sub-string. The function supports three return types, string, integral number and Boolean. The context in which this function is called will then dictate what type of value it will return. In a string context the function returns the sub-string when found within the source string. When called in an integral number context, the function returns the position, as a number, of the first character within the source string of the found sub-string. In a Boolean context the function will return true if the sub-string is found and false when it is not. For this function call a conversion function may be used to change the data type of the context in which it is called in order to obtain the desired value.

@FROM_DECIMAL_NUMBER

The FROM_DECIMAL_NUMBER function sets the context of its single parameter to a data type of decimal number. It supports the decimal number, integral number, string, and property return types. The value of its single parameter will be converted from the decimal number data type to the data type of the context of the FROM_DECIMAL_NUMBER function call.

One of the main uses of this function is to set the context of another function call to a decimal number. Certain functions do not directly support non-numeric data types for return. The FROM_DECIMAL_NUMBER function allows for these other functions to be called in a decimal number context and to then return that value in a data type such as string. While this function supports the decimal number return type, it is unnecessary to call this function in this context.

Parameters

@FROM_DECIMAL_NUMBER (Convert Parameter, [Precision, [Rounding Method]])	
Convert Parameter	Required decimal number parameter, contains the value to be converted to the data type of the function's context.
Precision	Optional integral number parameter, contains the precision to which the returned decimal number should be rounded. Positive values specify the number of digits after the decimal place. Negative numbers specify number of digits before the decimal.
Rounding Method	Optional integral number parameter, specifies how the return value should be rounded. The default is to round to the nearest value. If this parameter is set to 1, the rules pertaining to NIST rounding will be used to round the value returned by the function.

Supported Return Types

- Integral Number
- Decimal Number
- String
- Property

@FROM_INTEGRAL_NUMBER

The FROM_INTEGRAL_NUMBER function sets the context of its single parameter to a data type of integral number. It supports the decimal number, integral number, string, and property return types. The value of its single parameter will be converted from the integral number data type to the data type of the context of the FROM_INTEGRAL_NUMBER function call.

One of the main uses of this function is to set the context of another function call to integral number. Certain functions do not directly support non-numeric data types for return. The FROM_INTEGRAL_NUMBER function allows for these other functions to be called in an integral number context and to then return that value in a data type such as string. While this function supports the integral number return type, it is unnecessary to call this function in this context.

Parameters

@FROM_INTEGRAL_NUMBER (Convert Parameter)	
Convert Parameter	Required integral number parameter, contains the value to be converted to the data type of the function's context.

Supported Return Types

- Integral Number
- Decimal Number
- String
- Property

@FROM_STRING

The FROM_STRING function sets the context of its single parameter to a data type of string. It supports the decimal number, integral number, string, and property return types. The value of its single parameter will be converted from the decimal number data type to the data type of the context of the FROM_STRING function call.

One of the main uses of this function is to set the context of another function call to string. The FROM_STRING function allows for other functions to be called in a string context and to then return that value in a data type such as integral or decimal number, depending on the context of the FROM_STRING function call. While this function supports the string return type, it is unnecessary to call this function in this context.

Note that converting from a string to a numeric data type is not considered type safe. Such operations should be limited in use and all reasonable precautions should be made to ensure the string value being converted to a numeric data type contains only numeric characters. The FROM_STRING function processes each character of a string one at a time and, in a numeric conversion, will stop processing with the first non-numeric character found in the source string. The value returned will then be the numeric value at the point the processing ended, which is not likely to be a useful value.

Parameters

@FROM_STRING (Convert Parameter)	
Convert Parameter	Required string parameter, contains the value to be converted to the data type of the function's context.

Supported Return Types

- Integral Number
- Decimal Number
- String
- Property

@FROM_SIG_DECIMAL_NUMBER

The FROM_SIG_DECIMAL_NUMBER function sets the context of its single parameter to a data type of decimal number. It supports the decimal number, integral number, string, and property return types. The value of its single parameter will be converted from the decimal number data type to the data type of the context of the FROM_SIG_DECIMAL_NUMBER function call. The decimal number of this parameter will respect the rules of significant digit math.

One of the main uses of this function is to set the context of a decimal value that is either not stored in a decimal property, or one that is stored in a decimal number property but that does not have the significant digits math attribute set.

An optional parameter to this function is `Precision`. This parameter will specify the number of digits after the decimal to keep, with the last digit being rounded. If the precision is greater than the number of digits after the decimal, the value will be padded with zeros up to the specified precision.

Parameters

@FROM_SIG_DECIMAL_NUMBER (Convert Parameter [, Precision])	
Convert Parameter	Required decimal number parameter, contains the value that will be treated as a decimal number with respect of significant digit math.
Precision	Optional integral number parameter, specifies the number of digits to keep after the decimal in the value provided by `Convert Parameter`. If this value is greater than the number of digits after the decimal the value will be padded with zeros up to the specified precision.

Supported Return Types

- Integral Number
- Decimal Number

- String
- Property

@FROM_PROPERTY

The FROM_PROPERTY function takes a variable number of arguments. Each argument is evaluated as a property and this evaluation is within the context dictated by the argument that precedes it in the arguments to the function. The overall purpose of this function is to provide a kind of drill-down access to the value of a property that may be a descendent of the current object.

One of the main uses of this function is to set the context of another function call to property. Certain property functions do not directly support other data types for return. The FROM_PROPERTY function allows for these other functions to be called in a property context and to then return that value in another data type.

When taking multiple parameters, the FROM_PROPERTY function is likely to be used in an overall search of a collection for a given object based on a property value, returning another property within the same object instance.

Parameters

@FROM_PROPERTY (Property 1 [, ..., Property N])	
Property 1	Required property parameter, contains the value to be evaluated as a property in the context of the function call. This parameter sets the context of the next parameter to the function, if present. If this is the only parameter, it will be returned in the context of the function call.
Property N	Optional property parameter(s), contains the value to be evaluated as a property in the context of the preceding parameter to the function. This parameter sets the context of the next parameter to the function, if present. If this is the last parameter, it will be returned in the context of the function call.

Supported Return Types

- Boolean
- Integral Number
- Decimal Number
- String
- Property

Logical Functions for Rules

The Logical category of rule function terms within the rule definition provide the decision making and comparison logic to a rule. Many, though not all, of these functions support the Boolean return type and will return true or false in this context based on some decision or comparison. The functions within this category provide behaviors including conjunctions, value comparisons, and if-then-else if and switch-case logic.

Unlike similar constructs in other development tools, many of the functions within the logical category support more than two arguments. As an example, the @AND function will take two or more arguments, returning true only when all of its arguments are true. In other development languages to provide similar logic, multiple operators may be needed, as in:

```
if (value1 && value2 && value3 && value4)
```

Here the presence of multiple and operators are required. In the rule functions, the same logic would use a single @AND function call, with each value passed as an argument to the function:

```
AND (value1, value2, value3, value4)
```

Many of the other functions provide similar support within the context of their behavior.

@AND

The AND function performs a logical conjunction between its parameters, returning true or false based on this conjunction. Each of its parameters is evaluated in the order provided as Boolean values. If any parameter is evaluated as false, the return value is false. Otherwise the function returns true. The function must have at least one parameter and may contain as many more as is needed.

Parameters

@AND (Expression 1 [, ..., Expression N])	
Expression 1	Required Boolean parameter, the first to be evaluated by the function. If false, evaluation stops and the function returns false. If true, the function will return true if no other parameters are provided or evaluate the next parameter.
Expression N	Optional Boolean parameters, each evaluated by the function in the order provided. Evaluation stops for the first false value found and the function returns false. Otherwise the function returns true.

Supported Return Types

Boolean

@CASE

This function has been deprecated and will not be supported in future releases. It should be replaced with one of the following: CASE_INT, CASE_STRING, CASE_DEC, or IF. The CASE function provides switch-case logic, allowing for the evaluation of a single test value for the purpose of returning one of a multiple number of possible values. The CASE function takes a variable number of parameters, but with a minimum of three. The first parameter is evaluated as an integral number. This value is then treated as a positional value for one of the other parameters to the function, with the second parameter at position 1. The parameter at the position specified by the position parameter is then returned. The data type of the other parameters varies depending on the function's context. For example, if the context of the function call is a string, the parameters `Position1` through `PositionN` of the function will be treated as strings.

Parameters

@CASE (Position To Match, Position1 [, ..., PositionN])	
Position To Match	Required integral number parameter, indicating which of the case parameters the function should return.
Position 1	Required parameter, evaluated when the `Postion To Match` parameter evaluates to 1. The data type is dictated by the context in which the function is called.
Position N	Optional parameter, evaluated when the `Position to Match` parameter evaluates to N, where N is the position of the parameter in the function's parameter list. The data type is dictated by the context in which the function is called.

Supported Return Types

- Boolean
- Integral Number
- Decimal Number
- String
- Property

@CASE_INT

The CASE_INT function is used to return a single value from a variable list of multiple possible returns, with a comparison made between a switch and 1 or more case values, both of type integral number. The first parameter to the function is evaluated as an integral number. The last parameter to the function is the default return value and is optional. Parameters between the first and last are provided in pairs. The first parameter in a pair is the value to which the switch value is compared. If these two values are equal, the second value of the pair is returned by the function. The comparison between the switch value and the first parameter of a pair is performed as an integer comparison. The data type for the second parameter in each pair is dependent on the context in which the function is called. If none of the case parameters match the switch parameter and a default parameter is provided, that parameter is evaluated and returned by the function. If no default parameter is provided, the default null equivalent of the context's data type is returned, e.g. 0, False, null string, etc.

Parameters

@CASE_INT (Integer To Match, Integer 1, Return 1 [, ..., Integer N, Return N] [, Otherwise])	
Integer To Match	Required integral number parameter, contains the value upon which the function will switch, i.e. compare against each of the `Integer N` parameters in turn until a match is found.
Integer 1	Required integral number parameter, contains the value to which `Integer To Match` is compared. This parameter must be followed by the `Return 1` parameter, which is the value returned when `Integer To Match` matches the `Integer 1` value.
Return 1	Required parameter with a context-dependent data type, contains the value returned if `Integer To Match` matches `Integer 1`.

@CASE_INT (Integer To Match, Integer 1, Return 1 [, ..., Integer N, Return N] [, Otherwise])	
Otherwise / Integer N	Optional parameter(s), the data type in which it is evaluated is dependent on whether it is the last parameter to the function, or if it is followed by another parameter. When this is the last parameter to the function, it will be evaluated in the data type corresponding to the context of the function call. In this situation the parameter is the default `Otherwise` parameter, evaluated by the function when `Integer To Match` does not match any of the `Integer N` parameters. If this parameter is followed by another function parameter, it is evaluated as an integral number. In this situation it is evaluated by the function to determine if the value matches the `Integer To Match` parameter value. If it matches, the subsequent parameter is evaluated by the function. If not, the function evaluates the next parameter. Multiple `Integer N` parameters can be provided with the requirement that they are paired with corresponding `Return N` parameters. Only one `Otherwise` parameter may be provided.
Return N	Optional parameter(s) with a context-dependent data type, contains the value returned if `Integer To Match` matches the corresponding `Integer N` parameter.

Supported Return Types:

- Boolean
- Integral Number
- Decimal Number
- String
- Property

@CASE_STRING

The CASE_STRING function is used to return a single value from a variable list of multiple possible returns, with a comparison made between a switch and one or more case values, each of type string. The first parameter to the function is evaluated as a string. The last parameter to the function is the default return value and is optional. Parameters between the first and last are provided in pairs. The first parameter in a pair is the value to which the switch value is compared. If these two values are equal, the second value of the pair is returned by the function. The comparison between the switch value and the first parameter of a pair is performed as a case-sensitive string comparison. The data type for the second parameter in each pair is dependent on the context in which the function is called. If none of the case parameters match the switch parameter, and a default, non-paired parameter is provided, that parameter is evaluated and returned by the function. If no default parameter is provided, the default null equivalent of the context's data type is returned, e.g. 0, False, null string, etc.

Parameters

@CASE_STRING (String To Match, String 1, Return 1 [, ..., String N, Return N,] [, Otherwise])	
String To Match	Required string parameter, contains the value upon which the function will switch, i.e. compare against each of the `String N` parameters in turn until a match is found.
String 1	Required string parameter, contains the value to which `String To Match` is compared. This parameter must be followed by the `Return 1` parameter, which is the value returned when `String To Match` matches the `String 1` value.
Return 1	Required parameter with a context-dependent data type, contains the value returned if `String To Match` matches `String 1`.
Otherwise / String N	Optional parameter(s), the data type in which it is evaluated is dependent on whether it is the last parameter to the function, or if it is followed by another parameter. When this is the last parameter to the function, it will be evaluated in the data type corresponding to the context of the function call. In this situation, the parameter is the default `Otherwise` parameter, evaluated by the function when `String To Match` does not match any of the `String N` parameters. If this parameter is followed by another function parameter, it is evaluated as a string. In this situation, it is evaluated by the function to determine if the value matches the `String To Match` parameter value. If it matches, the subsequent parameter is evaluated by the function. If not, the function evaluates the next parameter. Multiple `String N` parameters can be provided with the requirement that they are paired with corresponding `Return N` parameters. Only one `Otherwise` parameter may be provided.
Return N	Optional parameter(s) with a context-dependent data type, contains the value returned if `String To Match` matches the corresponding `String N` parameter.

Supported Return Types:

- Boolean
- Integral Number
- Decimal Number
- String
- Property

@EQBOOL

The EQBOOL function takes two or more parameters, each of which is evaluated as a Boolean value and returning true if all parameters are either true or all are false. If all parameters have the same Boolean value, the function will return true. Otherwise, it will return false. The function will end evaluation and return false upon the first parameter found to be different than others passed to it.

Parameters:

@EQBOOL (Boolean 1 [, ..., Boolean N])	
Boolean 1	Boolean required parameter, evaluated for comparison to all other parameters to the function.
Boolean N	Optional additional Boolean parameter(s), evaluated for comparison to `Boolean 1`.

Supported Return Types
Boolean

@EQDEC

The EQDEC function takes two or more parameters, each evaluated as a decimal value, compares them for equality, returning true if all are equal or false if any are found to be different. The function will end evaluation of all parameters at the point where the first different value is found. If only a single parameter is provided, the function returns true.

Parameters

@EQDEC (Decimal 1 [, ..., Decimal N])	
Decimal 1	Required decimal number parameter, evaluated by the function for comparison to all other parameters.

@EQDEC (Decimal 1 [, ..., Decimal N])	
Decimal N	Optional decimal number parameter(s), each evaluated by the function for comparison to `Decimal 1`.

Supported Return Types
Boolean

@EQNUM

The EQNUM function takes two or more parameters, each evaluated as an integral number, compares them for equality, and returns true if all values are equal, or false if one or more are different. This function will end evaluation of all subsequent parameters after the first parameter is found to be different. If only a single parameter is provided, this function will return true.

Parameters

@EQNUM(Integer1 [, ..., IntegerN])	
Integer 1	Required integral number parameter, evaluated by the function for comparison to all other function parameters.
Integer N	Optional integral number parameter(s), evaluated by the function for comparison to `Integer 1`.

Supported Return Types
Boolean

@EQSTR

The EQSTR function takes one or more parameters, each evaluated as a case-sensitive string, and compares them for equality, returning true if all values are equal, or false if one or more values are different. The function will end evaluation of all subsequent parameters when the first different value is found. The function will return true if only a single parameter is provided.

Parameters

@EQSTR(String1 [, ..., StringN])	
String 1	Required string parameter, evaluated by the function for comparison to all other parameters to the function.

@EQSTR(String1 [, ..., StringN])	
String N	Optional string parameter(s), each evaluated by the function for comparison to `String 1`.

Supported Return Types
Boolean

@GT

The GT function takes two or more integral number parameters, comparing the second through the last parameters to the first, returning true if the first parameter is greater than all subsequent parameters. If any parameter is found to be greater than or equal to the first, the function will not evaluate any subsequent parameters and will return false.

Parameters

@GT (Integer 1, [, ..., Integer N])	
Integer 1	Required integral number parameter, contains the value to which all other parameters will be compared.
Integer N	Optional integral number parameter(s), each containing a value to be compared against `Integer1`. If `Integer1` is less than or equal to this parameter, the function returns false and will not evaluate any subsequent parameters. If `Integer 1` is greater than this parameter, the function will continue to evaluate any additional parameters, or return true of no other parameters are provided.

Supported Return Types
Boolean

@GTDEC

The GTDEC function takes two or more decimal number parameters, comparing the second through the last parameters to the first, returning true if the first parameter is greater than subsequent parameters. If any parameter is found to be greater than or equal to the first, the function will return false. It will not evaluate any subsequent parameters.

Parameters

@GTDEC (Decimal 1 [, ..., Decimal N])	
Decimal 1	Required decimal number parameter, contains the value to which all other parameters will be compared.
Decimal N	Optional decimal number parameter(s), each containing a value to be compared against `Decimal 1`. If `Decimal 1` is less than or equal to this parameter, the function returns false and will not evaluate any subsequent parameters. If `Decimal 1` is greater than this parameter, the function will continue to evaluate any additional parameters, or return true if no other parameters are provided.

Supported Return Types
Boolean

@GTEQ

The GTEQ function takes two or more integral number parameters, comparing the second through the last parameter to the first parameter, and returning true if the first parameter is greater than or equal to all subsequent parameters. If any other parameter is found to be greater than the first, the function will return false. It will not evaluate any subsequent parameters. If only a single parameter is provided the function will always return true.

Parameters

@GTEQ (Integer 1 [, ..., Integer N])	
Integer 1	Required integral number parameter, contains the value to which all other parameters will be compared.
Integer N	Optional integral number parameter(s), each containing a value to be compared against `Integer 1`. If `Integer 1` is less than this parameter, the function returns false and will not evaluate any subsequent parameters. If `Integer 1` is greater than or equal to this parameter, the function will continue to evaluate any additional parameters, or return true of no other parameters are provided.

Supported Return Types
Boolean

@GTEQDEC

The GTEQDEC function takes two or more decimal number parameters, comparing the second through the last parameters to the first, returning true if the first parameter is greater than or equal to all subsequent parameters. If any other parameter is found to be greater than the first, the function will return false. It will not evaluate any subsequent parameters. If only a single parameter is provided, this function will always return true.

Parameters

@GTEQDEC (Decimal 1 [, ..., Decimal N])	
Decimal 1	Required decimal number parameter, contains the value to which all other parameters will be compared.
Decimal N	Optional decimal number parameter(s), each containing a value to be compared against `Decimal 1`. If `Decimal 1` is less than this parameter, the function returns false and will not evaluate any subsequent parameters. If `Decimal 1` is greater than or equal to this parameter, the function will continue to evaluate any additional parameters, or return true if no other parameters are provided.

Supported Return Types
Boolean

@IF

The IF function provides the `if-then-else/else if` logic to rules. The function can take a variable number of parameters, and will behave differently based on the parameters. First, it can take a single parameter whose value is evaluated as a Boolean. If this parameter is true, the IF function will return true; otherwise it will return false. Note that this form of the function has limited use, as the condition being checked can be passed directly to what would otherwise be the caller of the IF function, without actually making the IF function call.

Second, the function can take a multiple number of parameters to provide the `if-then-else/else if` logic. The first parameter is a `Condition` parameter, and is evaluated as a Boolean. When true is returned, its corresponding `Then` parameter is evaluated and the resulting value is returned.

If a `Condition` parameter returns false, its corresponding `Then` parameter is not evaluated. The next `Condition` parameter is evaluated. A final optional parameter can be provided as the `Else` parameter. This parameter is evaluated when all `Condition` parameters have evaluated to false. The value resulting from evaluation of the `Else` parameter is then returned by the IF function.

All `Then` parameters and the `Else` parameter are evaluated in the context of the IF function call. The expected data type for these parameters is then the data type of that context.

Parameters

@IF (Condition 1 [,Then 1] [, ... Else If Condition N, Then N] [, Else])	
Condition 1	Required Boolean parameter, contains the value evaluated by the function to determine a return. When true, `Then 1` is returned if specified. When false, either the next `Else If Condition N` parameter is evaluated if specified. `Else` is evaluated if no `Else If Condition` exists. The context null-equivalent is returned if neither an `Else If Condition` or `Else` parameter is provided. If `Condition 1` is the only parameter, its Boolean value will be returned.
Then 1	Optional parameter, evaluated in the context of the function call. This parameter is evaluated when `Condition 1` is true. The value returned by the evaluation of `Then 1` is then returned by the IF function.
Else / Else If Condition N	Optional parameter(s), evaluated when `Condition 1` or the preceding `Else If Condition` parameter returns false. The data type for this parameter is dependent on whether it is the last parameter to the function or is followed by another parameter. If it is the last parameter, it is evaluated in the context of the function call. Its is the `Else` parameter to the function and will be evaluated when all preceding `Condition` parameters have returned false. If this parameter is not the last for the function, it is evaluated as a Boolean and is treated as an `Else IF Condition N` parameter. If it evaluates to true the corresponding `Then N` parameter will be evaluated by the function.
Then N	Optional parameter, evaluated in the context of the function call. This parameter is evaluated when its corresponding `Else If Condition N` parameter returns true. The value returned by the evaluation of `Then N` is then returned by the IF function.

Supported Return Types

- Boolean
- Integral Number
- Decimal Number
- String

- Property

@LT

The LT function takes two or more integral number parameters, comparing the second through last parameters with the first and returning true if the first parameter is less than all subsequent parameters. If any subsequent parameter is found to be equal to or less than the first parameter the function will return false. It will not evaluate any subsequent parameters. If only a single parameter is provided this function will return true.

Parameters

@LT (Integer 1 [, ..., Integer N])	
Integer 1	Required integral number parameter, contains the value to which all other parameters will be compared.
Integer N	Optional integral number parameter(s), each containing a value to be compared against `Integer 1`. If `Integer 1` is greater than or equal to this parameter, the function returns false and will not evaluate any subsequent parameters. If `Integer 1` is less than this parameter, the function will continue to evaluate any additional parameters, or return true of no other parameters are provided.

Supported Return Types
Boolean

@LTDEC

The LTDEC function takes two or more decimal number parameters, comparing the second through last parameters with the first and returning true if the first parameter is less than all subsequent parameters. If any subsequent parameter is found to be equal to or less than the first parameter the function will return false. It will not evaluate any subsequent parameters. If only one parameter is provided to this function it will return true.

Parameters

@LTDEC (Decimal 1 [, ..., Decimal N])	
Decimal 1	Required decimal number parameter, contains the value to which all other parameters will be compared.

@LTDEC (Decimal 1 [, ..., Decimal N])	
Decimal N	Optional decimal number parameter(s), each containing a value to be compared against `Decimal 1`. If `Decimal 1` is greater than or equal to this parameter, the function returns false and will not evaluate any subsequent parameters. If `Decimal 1` is less than this parameter, the function will continue to evaluate any additional parameters, or return true if no other parameters are provided.

Supported Return Types
Boolean

@LTEQ

The LTEQ function takes two or more integral number parameters, comparing the second through last parameter with the first parameter and returning true if the first parameter is less than or equal to all subsequent parameters. If any subsequent parameter is found to be less than the first parameter, the function will return false. It will not evaluate any subsequent parameters. If only a single parameter is provided, the function will return true.

Parameters

@LT (Integer 1 [, ..., Integer N])	
Integer 1	Required integral number parameter; contains the value to which all other parameters will be compared.
Integer N	Optional integral number parameter(s), each containing a value to be compared against `Integer 1`. If `Integer 1` is greater than this parameter, the function returns false and will not evaluate any subsequent parameters. If `Integer 1` is less than or equal to this parameter, the function will continue to evaluate any additional parameters, or return true if no other parameters are provided.

Supported Return Types
Boolean

@LTEQDEC

The LTEQDEC function takes two or more decimal number parameters, comparing the second through last parameter with the first parameter and returning true if the first parameter is less than or equal to all subsequent parameters. If any subsequent parameter is found to be less than the first parameter, the function will return false and it will not evaluate any subsequent parameters. If only one parameter is provided, the function will return true.

Parameters

@LTEQDEC (Decimal 1 [, ..., Decimal N])	
Decimal 1	Required decimal number parameter; contains the value to which all other parameters will be compared.
Decimal N	Optional decimal number parameter(s), each containing a value to be compared against `Decimal 1`. If `Decimal 1` is greater than this parameter, the function returns false and will not evaluate any subsequent parameters. If `Decimal 1` is less than or equal to this parameter, the function will continue to evaluate any additional parameters, or return true if no other parameters are provided.

Supported Return Types
Boolean

@NAND

The NAND function takes one or more Boolean parameters and evaluates each until the first true value is found or until all parameters have been evaluated. If a true parameter value is found, the function returns false. It will return true if all parameters evaluate to false. The function will end evaluation when the first true value is found.

Parameters

@NAND (Expression 1 [, ..., Expression N])	
Expression 1	Required Boolean parameter, evaluated by the function for its Boolean value. If false, the function will return true. If true, the function will continue to evaluate the next parameter.
Expression N	Optional Boolean parameter, evaluated by the function for its Boolean value. If false, the function will return true. If true, the function will continue to evaluate the next parameter, or if no more parameters are provided, the function will return false.

Supported Return Types
Boolean

@NOR

The NOR function takes one or more parameters and returns true if all parameters are false, or false if any parameter is found to be true. This function supports the Boolean return type. Each parameter is evaluated as a Boolean.

Parameters

@NOR(Expression1 [, ..., ExpressionN])	
Expression 1	Required Boolean parameter; if true the function will return false. If false, the function will continue evaluating the next parameter, or return true if no additional parameters are provided.
Expression N	Optional Boolean parameter(s), each evaluated for their Boolean value. The function will return false and end evaluation with the first true parameter found. If all parameters are true, the function will return false.

Supported Return Types
Boolean

@NOT

The NOT function will evaluate all of the parameters as Booleans and return false if all parameters are true, or true if one or more parameters are false. This function supports the Boolean return type.

Most calls to this function provide only a single Boolean parameter, the value for which is inverted and returned.

Parameters

@NOT (Expression 1 [, ..., Expression N])	
Expression 1	Required Boolean parameter; if the value is true, the function will return false and not evaluate any subsequent parameters. If the value is false and if there are no subsequent parameters, the function will return true. If additional parameters are present, they will be evaluated only when `Expression 1` is false.

@NOT (Expression 1 [, ..., Expression N])	
Expression N	Optional Boolean parameter(s), evaluated by the function in the order provided. The function will return false for the first parameter found to be true. If all parameters evaluate to true, the function will return false.

Supported Return Types
Boolean

@OR

The OR function takes two or more Boolean parameters and will return true if any one or more of its parameters evaluates to true; otherwise it returns false. This function can be called in a Boolean context.

Parameters

@OR (Expression 1 [, ..., Expression N])	
Expression 1	Required Boolean parameter; contains the first value evaluated by the function for true or false. If this parameter evaluates to true, the function will end evaluation and return true to the caller.
Expression N	Optional Boolean parameter(s); contains the next value evaluated by the function. Additional parameters are evaluated until a true parameter is found. The function will end evaluation and return true for the first true parameter found.

Supported Return Types
Boolean

@XOR

The XOR function provides the exclusive or logic and can take one or more parameters, each of which is evaluated as a Boolean, and returning true when one and only one of its parameters is true. If all of the parameters evaluate to false, or if two or more parameters evaluate to true, this function will return false. This function will end evaluation of all parameters and return false when the second true value is found.

Parameters

@XOR (Expression 1 [, ..., Expression N])	
Expression 1	Required Boolean parameter; evaluated by the function in relation to all other parameters looking for an exclusive true value among the parameter list.
Expression N	Optional Boolean parameter(s); evaluated by the function in relation to all other parameters looking for an exclusive true value among the parameter list. If both this and any preceding parameter are true, the function will end evaluation and return false. No additional parameters will be evaluated.

Supported Return Types
Boolean

Mathematical Functions for Rules

The Mathematical category of rule function terms available within the rule definition provide the mathematical operations. These include addition, subtraction, multiplication, division, and modulus, as well as several other math-related operations. Additional operations include returning the minimum and maximum values from a set of values, limiting a value to a given range, square root operations, rounding, and other similar functions.

@ABS

The ABS function returns the absolute value of the given numerical parameter. The function takes a single parameter, which is evaluated as either an integral or decimal value matching the context of the function call.

Parameters

@ABS (Number)	
Number	Required number parameter, evaluated as either an integral or decimal number depending on the context of the function call. The absolute value of this parameter is returned.

Supported Return Types

- Decimal Number
- Integral Number

@DIFF

The DIFF function takes two or more numeric parameters evaluated in the context of the function call. The second parameter through the last are subtracted from the first parameter. The function then returns the result. This function supports an integral or decimal number return type.

Parameters

@DIFF(Number 1 [, ..., Number N])	
Number 1	Required parameter, in a numeric context, contains the value from which all subsequent parameters will be subtracted. Evaluated as an integral or decimal number, matching the context of the function call.
Number N	Optional parameter(s), in a numeric context, contains the value(s) from which all subsequent parameters will be subtracted. Evaluated as integral or decimal number(s), matching the context of the function call.

Supported Return Types

- Integral Number
- Decimal Number

@DISTANCE

The DISTANCE function takes four decimal parameters assumed to be latitude and longitude values for two map positions, and returns the resulting distance in meters as a decimal value.

When working with GPS location values, this function should not be used. See the System functions GPS_LOCATION, LATITUDE, LONGITUDE, DISTANCE_MILES, DISTANCE_KILOMETERS, LOCATION, and IS_VALID_LOCATION.

Parameters

@DISTANCE (x1, y1, x2, y2)	
x1	Required decimal number parameter; contains the x coordinate of the first position.
y1	Required decimal number parameter; contains the y coordinate of the first position.
x2	Required decimal number parameter; contains the x coordinate of the second position.

@DISTANCE (x1, y1, x2, y2)	
y2	Required decimal number parameter; contains the y coordinate of the second position.

Supported Return Types
Decimal Number

@DIV

The DIV function takes two parameters, for which the data type is dependent on the context of the function. It divides the first parameter by the second and returns the quotient as either an integral number or decimal number, depending on the function's context.

Parameters

@DIV (Dividend, Divisor)	
Dividend	Required parameter containing the dividend value or the value to be divided. Evaluated as either an integral or decimal number, depending on the function's context.
Divisor	Required parameter containing the divisor value or the value to divide into `Dividend`. Evaluated as either an integral or decimal number, depending on the function's context.

Supported Return Types

- Integral Number
- Decimal Number

@FORMAT_DECIMAL

The FORMAT_DECIMAL function converts the given decimal number parameter into a string. It takes up to five additional optional parameters that are used in formatting the converted string value. The first parameter is the value to be converted and is required. This parameter is evaluated as a decimal, though the value itself may be either an integral or decimal number data type. This function should be used for any read-only detail screen field displaying a decimal value.

Parameters

@FORMAT_DECIMAL (Decimal [, Precision, Use Thousands Separator, Use Lead Zero, Decimal Point, Thousands Separator])	
Decimal	Required decimal number parameter; contains the value to be formatted to a string by the function. If this parameter is a decimal number property, the definition of that property's rounding attributes will affect the final value, specifically when rounding to a specified precision.
Precision	Optional integral number parameter; contains the number of digits after the decimal to keep when converting `Decimal`. The last kept digit will be rounded. If `Decimal` is a decimal number property, the property's rounding attributes will determine the behavior of rounding for the value returned. If `Decimal` is a decimal property, the precision defined for the property will take effect before the function applies any additional precision to the resulting string returned. If this value is not specified, the precision will be determined automatically by the function.
Use Thousands Separator	Optional Boolean parameter with a default value of false. When true, the final string value returned will contain a comma to denote thousands, millions, etc. When false, no comma will be present in the resulting string returned by the function.
Use Lead Zero	Optional Boolean parameter with a default value of false. When true, the final string value returned will contain a leading 0 in the ones position for decimals that contain only fractional values; e.g. when false or not specified `.23`; when true `0.23`.
Decimal Point	Optional string parameter with a default value of a decimal point (.). This value may be set to any single character to be used in place of a decimal point. Many locales use a comma to denote the fractional portion of a decimal value.

@FORMAT_DECIMAL (Decimal [, Precision, Use Thousands Separator, Use Lead Zero, Decimal Point, Thousands Separator])	
Thousands Separator	Optional string parameter with a default value of comma (,). This value can be set to any single character to be used in place of a comma to separate thousands and hundreds, millions and hundred thousands, etc. This parameter is only evaluated by the function when `Use Thousands Separator` is true. Many locales use a period (.) as the separator character.

Supported Return Types
String

@MAX

The MAX function takes one or more parameters containing numerical values and compares each to the other, returning the value of the parameter with the greatest value. This function can be called in a decimal or integral number context and will evaluate its parameters according to that context.

Parameters

@MAX (Number 1 [, ..., Number N])	
Number 1	Required parameter evaluated as the data type of the context in which the function is called. Contains the first value to be compared against all other parameters by the function.
Number N	Optional parameter(s) evaluated as the data type of the context in which the function is called. Each contains the value(s) to be compared against all other parameters to the function.

Supported Return Types
- Integral Number
- Decimal Number

@MIN

The MIN function takes two or more parameters containing numerical values and compares each to the other, returning the value of the parameter with the least value. This function can be called in a decimal or integral number context and will evaluate its parameters according to that context.

Parameters

@MIN(Number 1 [, ..., Number N])	
Number 1	Required parameter evaluated as the data type of the context in which the function is called. Contains the first value to be compared against all other parameters to the function.
Number N	Optional parameter(s) evaluated as the data type of the context in which the function is called. Each contains the value(s) to be compared against all other parameters to the function.

Supported Return Types

- Integral Number
- Decimal Number

@MOD

The MOD function performs a modulus operation on its two parameters, dividing the first by the second and returning the remainder. The parameters are evaluated as either decimal or integral numbers, matching the context of the function.

Parameters

@MOD (Dividend, Divisor)	
Dividend	Required parameter evaluated as either an integral or decimal number matching the context of the function call. Contains the dividend value, or the value to be divided by `Divisor`.
Divisor	Required parameter evaluated as either an integral or decimal number matching the context of the function call. Contains the divisor value, or the value to be divided into `Dividend`.

Supported Return Types

- Integral Number
- Decimal Number

@PARSE_FORMATTED_DECIMAL

The PARSE_FORMATTED_DECIMAL function converts the given string into a decimal. It takes up to two optional parameters that are used for deciphering the format of the number stored in the string.The first parameter is the value to be converted and is required. This parameter is evaluated as a string, and a decimal is created from it using the optional parameters or the client's locale. This string must be a valid representation of a decimal.

Parameters

@PARSE_FORMATTED_DECIMAL (String [, Decimal Point, Thousands Separator])	
String	Required string parameter, contains the value to be parsed interpreted as a string and parsed into a decimal by the function.
Decimal Point	Optional string parameter with a default value of the client's locale decimal separator. This value may be set to any single character to be considered as a decimal separator.
Thousands Separator	Optional string parameter with a default value of the client's locale thousands separator. This value may be set to any single character to be considered as separating thousand and hundreds, millions and hundred thousands, etc.

Supported Return Types
Decimal

@PERCENT

The PERCENT function takes one decimal number parameter, divides that parameter by 100, and returns the result. This function ignores any significant digit rules during the division process.

Parameters

@PERCENT (Dividend)	
Dividend	Required decimal number parameter; contains the value to be divided by 100.

Supported Return Types

Decimal Number

@PRECISION

The PRECISION function takes a single decimal number parameter and returns its precision. This is the number of digits after the decimal. The function supports string, integral number and decimal number return types for this precision value. The parameter to it is always evaluated as a decimal number.

Parameters

@PRECISION (Decimal)	
Decimal	Required decimal number parameter; contains the value whose precision is to be returned by the function, i.e. the number of digits after the decimal.

Supported Return Types

- Integral Number
- Decimal Number
- String

@PROD

The PROD function takes two or more numeric parameters, multiplies the values, and returns the product. The parameters to the function are evaluated as either decimal or integral numbers, matching the context of the function call.

Parameters

@PROD (Number 1 [, ..., Number N])	
Number 1	Required parameter; evaluated as either an integral or decimal number matching the context of the function call. Contains the value to be multiplied with all other function parameters.
Number N	Optional parameter(s); evaluated as either integral or decimal number(s) matching the context of the function call. Contains the value(s) to be multiplied with all other function parameters.

Supported Return Types

- Integral Number
- Decimal Number

@RANGE_LIMIT

The RANGE_LIMIT function constrains a given numeric parameter to within a range of values, returning a value that is no greater than or less than a given set of upper and lower limits. The function takes three parameters: the value to be constrained; a lower limit; and an upper limit. If the value to be constrained is greater than the lower limit and less than the upper limit, the function will return the value. If this value is less than the lower limit, the value of the lower limit will be returned. If the constrained value is greater then the upper limit, the upper limit value will be returned. Each of the three parameters are evaluated as either decimal or integral numbers, matching the context of the function call.

Parameters

@RANGE_LIMIT (Constrain, Limit 1, Limit 2)	
Constrain	Required parameter; evaluated as either an integral or decimal number matching the context of the function call. Contains the value to be constrained by the function. This value will be returned if it is between the values of the `Limit 1` and `Limit 2` parameters.
Limit 1	Required parameter; evaluated as either an integral or decimal number matching the context of the function call. Contains the first limiting value in the range to which `Constrain` will be limited. This may be either the minimum or maximum value to which the return value is to be constrained.
Limit 2	Required parameter; evaluated as either an integral or decimal number matching the context of the function call. Contains the second limiting value in the range to which `Constrain` will be limited. This may be either the minimum or maximum value to which the return value is to be constrained.

Supported Return Types

- Integral Number
- Decimal Number

@ROUND

The ROUND function will round a numeric value to the specified precision. The first parameter to the function is the numeric value to be rounded. The second parameter is always evaluated as an integral number, and specifies the number of digits before or after the decimal place to which the number should be rounded. This function may be called in an integral or decimal context.

Parameters

@ROUND (Number To Round, Precision)	
Number To Round	Required parameter; evaluated as either an integral or decimal number matching the context of the function call. Contains the value to be rounded by the function.
Precision	Required integral number parameter; contains the value specifying the precision to which `Number To Round` should be rounded. Positive numbers indicate a precision after the decimal place. Negative numbers indicate position before the decimal. This parameter can be only negative numbers when the function is called in an integral number context.

Supported Return Types

- Integral Number
- Decimal Number

@SIGN

The SIGN function takes a single decimal number parameter and returns -1, 0, or 1 if the parameter is negative, zero, or positive in value, respectively. The function may be called in an integral or decimal number context. The parameter is always evaluated as a decimal number.

Parameters

@SIGN (Number)	
Number	Required number parameter; evaluated as either a decimal or integral number based on context. Contains the value whose sign is to be determined by the function.

Supported Return Types

- Integral Number
- Decimal Number

@SIGNIFICANT_DIGITS

The SIGNIFICANT_DIGITS function takes a single parameter evaluated as a decimal number, and returns the number of significant digits it contains.

Parameters

@SIGNIFICANT_DIGITS (Decimal)	
Decimal	Required decimal number parameter; contains the value whose number of significant digits is to be calculated by the function.

Supported Return Types

- Integral Number
- Decimal Number
- String

@SQRT

The SQRT function returns the square root of a decimal number to the specified precision. The number for which the square root is found is evaluated as a decimal number, though whole integral values can be provided. The precision of the result is specified by an integral number.

Parameters

@SQRT (Decimal, Precision)	
Decimal	Required decimal number parameter; contains the value for which the square root will be found.
Precision	Required integral number parameter; contains the value specifying the precision to which the square root value will be calculated. The function will round the result to this precision. If a property is specified for the `Decimal` argument, the definition of that property's rounding behavior will be applied. A `Precision` value of 0 indicates no digits after the decimal, and a negative value indicates digits before the decimal.

Supported Return Types
Decimal Number

@SUM

The SUM function takes one or more numeric parameters, adds the values of each, and returns the result. This function may be called in an integral number, decimal number, or string context. The data type of each parameter in the numeric contexts will match that context. In a string context, the function will treat its parameters as decimal numbers, though integral numbers may be provided.

Parameters

@SUM (Number 1 [, ..., Number N])	
Number 1	Required parameter; evaluated as either an integral or decimal number, depending on the context of the function call. Contains the first value to be summed with all other parameters to the function.
Number N	Optional parameter(s); evaluated as either an integral or decimal number, depending on the context of the function call. Contains the additional value(s) to be summed with all other parameters to the function.

Supported Return Types

- Integral Number
- Decimal Number
- String

@TOTAL

The TOTAL function will add the values of a given object property together for all instances of the object in a specified collection property. Optional criteria may be specified to include only specific objects within the collection being processed.

This function takes three parameters. The first is the object collection property, and the second is the object property whose value is to be summed in each object instance. An optional third parameter can be provided to include only certain object instances within the collection in this operation. This function supports the integral number and decimal number return types.

The object property values are totalled by the function based on the function's context, not the data type of the object property definition. This is important to note when the context is in an integral number, and the properties to be totaled are decimals. In this situation, the fractional portion of the properties will be truncated from the values prior to being added together.

Parameters

@TOTAL (Object Collection, [Child Property [, Include Criteria]])	
Object Collection	Required object collection property parameter; contains the object instances whose property values will be totaled by the function.
Child Property	Optional property parameter; specifies the object property to be totaled in each object instance in `Object Collection`. These property values will be evaluated as the data type of the function's context, which may be either integral number or decimal number.
Include Criteria	Optional Boolean parameter; this term is evaluated once for, and in the context of each object instance in `Object Collection`. The function will process each object instance for which this term returns true, and exclude those for which it returns false. If this parameter is not provided, all object instances will be processed by the function.

Supported Return Types

- Integral Number
- Decimal Number

@TRUNC

The TRUNC function will truncate the given numeric value to the specified position either before or after the decimal. The first parameter to the function is the value to be truncated and will be evaluated as either a decimal or integral number, matching the context of the function call. The second parameter is evaluated as an integral number and specifies the precision, or number of digits to which the value should be truncated. A positive precision value counts the number of digits to the right of the decimal, while a negative precision counts them to the left of the decimal. Note that this function differs from the ROUND function in that no rounding occurs. The number is truncated to the specified precision with no rounding behavior.

Parameters

@TRUNC (Number [, Precision])	
Number	Required parameter; evaluated as either an integral or decimal number matching the context of the function call. Contains the value to be truncated by the function.
Precision	Optional integral number parameter; contains the value specifying the precision of the resulting truncation. Positive values indicate the number of places to the right of the decimal, while negative values indicate the number of places to the left.

Supported Return Types

- Integral Number
- Decimal Number

Property Functions for Rules

The Property category of rule function terms available within the rule definition provide functions that are specific to working with properties, and in most cases are intended for use with properties of a given type. While they may take any property as a parameter, the resulting behavior of the function may not be desired if the selected property is not of the type for which the function is intended.

Many of the functions within this category are intended for use with collection properties. Most of these are named to reflect this, beginning with the word COLLECTION, e.g. @COLLECTION_FIND. Exceptions to this are COUNT and SIZE, which both return the total number of objects in a collection property, optionally counting only those that meet some criteria.

Another sub-set of the functions within this category are intended for use with properties of type external data. External data properties are defined to reference files stored on the client device from within the mobile application. The functions within this category that are intended for this property type each begin with the value FILE in their names, e.g. @FILE_PATH.

Other functions within this category include those that work with the user interface, returning values or names from the screens and fields of the currently displayed screen or screen set on the client. Each of these functions begins with the value SCREEN or SCREENSET in their names, e.g. @SCREENFIELDVALUE.

@COLLECTION_FIND

The COLLECTION_FIND function searches a given collection for the first member for which the given second parameter returns true. This second parameter is evaluated once for each member of the collection in a Boolean context. When a member of the search collection is found, that member is returned to the caller of the function. If no member of the collection results in the search criteria returning true, an empty property of the type within the search collection is returned.

Parameters

@COLLECTION_FIND (Collection Property, Search Criteria)	
Collection Property	Required property parameter; the property referenced for this parameter is assumed to be a collection and contains the members to be searched by the function using the specified `Search Criteria` parameter.
Search Criteria	Required Boolean parameter; this term is evaluated once for, and in the context of each member of `Collection Property`. The member of `Collection Property` returned by the function will be the first one for which `Search Criteria` returns true.

Supported Return Types
Property

@COLLECTION_FIND_BY_DEC

The COLLECTION_FIND_BY_DEC function searches a collection property for the first member to match a specified decimal value. For object collection properties, the property within each object of the collection to compare to the search value is also specified. This function will return the first member within the specified collection property found to match the provided search decimal value. If no member of the collection matches the search value, an empty instance of the member type is returned to the function caller.

Parameters

@COLLECTION_FIND_BY_DEC (Collection Property, Search Decimal [, Search Property])	
Collection Property	Required property parameter; this parameter is assumed to be a collection property. References the collection the function will search.
Search Decimal	Required decimal number parameter; contains the value to search for within `Collection Property`.
Search Property	Optional property parameter; when `Collection Property` contains object instances `SearchProperty,` specifies the property within that object type to compare against `Search Decimal`. The value of the property specified for this parameter is converted from the data type of the property to a decimal number for comparison to `Search Decimal`.

Supported Return Types
Property

@COLLECTION_FIND_BY_NUM

The COLLECTION_FIND_BY_NUM function searches a collection property for the first member to match a specified integral value. For object collection properties, the property within each object of the collection to compare to the search value is also specified. This function will return the first member within the specified collection property found to match the provided search integral value. If no member of the collection matches the search value, an empty instance of the member type is returned to the function caller.

Parameters

@COLLECTION_FIND_BY_NUM (Collection Property, Search Integral [, Search Property])	
Collection Property	Required property parameter; this parameter is assumed to be a collection property. References the collection the function will search.

@COLLECTION_FIND_BY_NUM (Collection Property, Search Integral [, Search Property])	
Search Integral	Required integral number parameter; contains the value to search for within `Collection Property`.
Search Property	Optional property parameter; when `CollectionProperty` contains object instances `Search Property`, specifies the property within that object type to compare against `Search Integral`. The value of the property specified for this parameter is converted from the data type of the property to an integral number for comparison to `Search Integral`.

Supported Return Types
Property

@COLLECTION_FIND_BY_STR

The COLLECTION_FIND_BY_STR function searches a collection property for the first member to match a specified string value. For object collection properties, the property within each object of the collection to compare to the search value is also specified. This function will return the first member within the specified collection property found to match the provided search string value. If no member of the collection matches the search value, an empty instance of the member type is returned to the function caller.

Parameters

@COLLECTION_FIND_BY_STR (Collection Property, Search String [, Search Property])	
Collection Property	Required property parameter; this parameter is assumed to be a collection property. References the collection the function will search.
Search String	Required string parameter; contains the value to search for within `Collection Property`.

@COLLECTION_FIND_BY_STR (Collection Property, Search String [, Search Property])	
Search Property	Optional property parameter; when `Collection Property` contains object instances `Search Property,` specifies the property within that object type to compare against `Search String`. The value of the property specified for this parameter is converted from the data type of the property to a string for comparison to `Search String`.

Supported Return Types
Property

@COLLECTION_MAX

The COLLECTION_MAX function searches an object collection for the object instance whose designated property contains the largest value of all members of the collection and then returns that maximum value. The function takes parameters for the object collection to be searched, the property within the object type of the collection whose value is to be compared between object instances, and optionally a rule term containing the criteria specifying which objects to search and which to exclude.

The optional Boolean parameter to the function is evaluated once for each object instance contained in the specified collection. This term is evaluated in the context of each object instance. The function will then compare only those objects for which this rule term returns true, with those for which it returns false being excluded from the comparison.

The data type of the property to be compared should be one for which a value comparison makes sense. While a minimum or maximum value is readily apparent in a primitive data type such as an integer, such a comparison makes little sense for a signature or external data property type. For property data types like the latter, the return value is undefined. The data type of the property to be compared in each function should be considered in relation to the data type of the function’s context. Though the function supports the integral number, decimal number, and string return types, the conversion from the property’s data type to the return type should be “type safe.” Specifically, if the designated property to compare in each object is a string, the function should not be called in an integral or decimal number context.

Parameters

@COLLECTION_MAX (Object Collection, Child Property [, Include Criteria])	
Object Collection	Required object collection property parameter; specifies the collection to be processed by the function.
Child Property	Required property parameter; specifies the property whose value will be compared in each object instance in `Object Collection`. The data type of this property specifies the type of comparison made between the values for each object instance.
Include Criteria	Optional Boolean parameter; this term is evaluated once for, and in the context of each object instance in `Object Collection`. The function will compare the `Child Property` value of each object instance for which `Include Criteria` returns true, and excludes from this processing each object for which false is returned. If this parameter is omitted, all object instances in the collection will be processed.

Supported Return Types

- Integral Number
- Decimal Number
- String

@COLLECTION_MIN

The COLLECTION_MIN function searches an object collection for the object instance whose designated property contains the smallest value of all members of the collection and then returns that minimum value. The function takes parameters for the object collection to be searched, the property within the object type of the collection whose value is to be compared between object instances, and optionally a rule term containing the criteria specifying which objects to search and which to exclude.

The optional Boolean parameter to the function is evaluated once for each object instance contained in the specified collection. This term is evaluated in the context of each object instance. The function will then compare only those objects for which this rule term returns true, with those for which it returns false being excluded from the comparison.

The data type of the property to be compared should be one for which a value comparison makes sense. While a minimum or maximum value is readily apparent in a primitive data type such as an integer, such a comparison makes little sense for a signature or external data property type. For such property data types as the latter, the return value is undefined.

The data type of the property to be compared in each function should be considered in relation to the data type of the function's context. Though the function supports the integral number, decimal number, and string return types, the conversion from the property's data type to the return type should be "type safe." Specifically, if the designated property to compare in each object is a string, the function should not be called in an integral or decimal number context.

Parameters

@COLLECTION_MIN (Object Collection, Child Property [, Include Criteria])	
Object Collection	Required object collection property parameter; specifies the collection to be processed by the function.
Child Property	Required property parameter; specifies the property whose value will be compared in each object instance in `Object Collection`. The data type of this property specifies the type of comparison made between the values of each object.
Include Criteria	Optional Boolean parameter; this term is evaluated once for, and in the context of each object instance in `Object Collection`. The function will compare the `Child Property` value of each object instance for which `Include Criteria` returns true and exclude from this processing each object for which false is returned. If this parameter is omitted, all object instances in the collection will be processed.

Supported Return Types

- Integral Number
- Decimal Number
- String

@COUNT

The COUNT function returns the number of object instances in a given collection property and, optionally, can count only those where some condition is true, returning this count value or a Boolean indication of whether any objects within the collection were counted. The function takes two parameters. The first parameter is required and is the object collection property to be counted. The second parameter is an optional Boolean parameter that is evaluated once for each object in the collection. It is used to include only certain members of the collection in those counted. Specifically, each object for which the second parameter evaluates to true will be counted, and those for which it evaluates to false will not be counted.

Parameters

@COUNT (Object Collection [, Include Criteria])	
Object Collection	Required property parameter; contains the object collection to be counted by the function.
Include Criteria	Optional Boolean parameter; this term is evaluated once for, and in the context of each object instance in `Object Collection`. The function will count each object instance for which `Include Criteria` returns true and exclude from the count each object for which false is returned.

Supported Return Types

- Boolean
- Integral Number
- String

@CURRENTVALUE

The CURRENTVALUE function takes a variable number of property parameters. Each is evaluated in the context of the parameter that precedes it until either an invalid property is found or the last parameter is evaluated. The value of this property, or of a field on the current detail screen that targets the property, is returned.

The function can take a single property parameter, evaluated in the context of the function call. Either this property value is returned by the function, or the value of a detail screen field targeting that property on the current screen is returned. When multiple parameters are specified, it is assumed these parameters are either objects or object collections. Each subsequent parameter should be a child member of the parameter that precedes it in the function call. Otherwise, the parameter will be treated as an invalid property.

Parameters

@CURRENTVALUE (Property 1 [, ..., Property N])	
Property 1	Required property parameter; references the first property evaluated in the context of the function call. This parameter should reference a property that is a child member of the definition setting the context of the function call.
Property N	Optional property parameter(s); references the next property evaluated by the function in the context of the property that immediately precedes it in the function's parameter list. This should be a child member of that parameter.

Supported Return Types

- Boolean
- Integral Number
- Decimal Number
- String
- Property

@FILE_CHANGED

The FILE_CHANGED function is provided to work with the external data property type. It evaluates the file referenced by such a property and returns true if the file has been modified since it was downloaded to the client device or attached locally. Though the data type of this function's single parameter is string, the function expects an external data property and will return false for any other value provided.

Parameters

@FILE_CHANGED (File Name)	
File Name	Required string parameter; specifies the file to be checked by the function for modifications. Though the data type of this parameter is string, the intended usage of this function is that this parameter is an external data property. Specifying any other string value for this parameter will result in the function always returning false.

Supported Return Types
Boolean

@FILE_EXTENSION

The FILE_EXTENSION function takes a single parameter, normally an external data property, and returns the file extension referenced by that property as a string. This function is intended for use with external data properties as a part of the attached document's functionality. If the external data property provided as a parameter does not currently reference a file, the function will return an empty string. The return value from this function is undefined when the parameter provided is not an external data property.

Parameters

@FILE_EXTENSION (File Name)	
File Name	Required string parameter; specifies the file whose file extension is to be returned. While the data type of this parameter is a string, the intended parameter is a property of data type external data. The file extension of the file referenced by such a property is returned by this function. If the property does not reference a file, the function returns an empty string.

Supported Return Types
String

@FILE_NAME

The FILE_NAME function takes a single parameter, normally an external data property, and returns the name and extension of the file referenced by that property. This function is intended for use with external data properties as a part of the attached document's functionality. If the external data property provided as a parameter does not currently reference a file, the function will return an empty string. The return value from this function is undefined when the parameter provided is not an external data property

Parameters

@FILE_NAME (File Name)	
File Name	Required string parameter; specifies the file whose name and extension is to be returned. While the data type of this parameter is a string, the intended parameter is a property of data type external data. The name and file extension of the file referenced by such a property is returned by this function. If the property does not reference a file, the function returns an empty string.

Supported Return Types
String

@FILE_PATH

The FILE_PATH function takes a single parameter, normally an external data property, and returns the full path of the location where the file referenced by that property is stored. This function is intended for use with external data properties as a part of the attached document's functionality. If the external data property provided as a parameter does not currently reference a file, the function will return an empty string. The return value from this function is undefined when the parameter provided is not an external data property.

Parameters

@FILE_PATH (File Name)	
File Name	Required string parameter; specifies the file whose full path is to be returned. While the data type of this parameter is a string, the intended parameter is a property of data type external data. The full path to the location of the file referenced by such a property is returned by this function. If the property does not reference a file, the function returns an empty string.

Supported Return Types
String

@FILE_PATH_AND_NAME

The FILE_PATH_AND_NAME function takes a single parameter, normally an external data property, and returns the full path and name of the file referenced by that property. This function is intended for use with external data properties as a part of the attached document's functionality. If the external data property provided as a parameter does not currently reference a file, the function will return an empty string. The return value from this function is undefined when the parameter provided is not an external data property.

Parameters

@FILE_PATH_AND_NAME (File Name)	
File Name	Required string parameter; specifies the file whose full path and name is to be returned. While the data type of this parameter is a string, the intended parameter is a property of data type external data. The full path and name of the file referenced by such a property is returned by this function. If the property does not reference a file, the function returns an empty string.

Supported Return Types
String

@FILE_SIZE

The FILE_SIZE function takes a single parameter, normally an external data property, and returns the size in bytes of the file referenced by that property. This function is intended for use with external data properties as a part of the attached document's functionality. If the external data property provided as a parameter does not currently reference a file, the function will return an empty string. The return value from this function is undefined when the parameter provided is not an external data property.

Parameters

@FILE_SIZE (File Name)	
File Name	Required string parameter; specifies the file whose size is to be returned. While the data type of this parameter is a string, the intended parameter is a property of data type external data. The size of the file referenced by such a property is returned by this function. If the property does not reference a file, the function returns 0.

Supported Return Types
Integral Number

@IS_SPECIAL_VALUE

The IS_SPECIAL_VALUE function returns a Boolean value indicating whether or not a specified property is currently set to its special value. The function returns true when the property value is equal to its defined special value. It returns false when it is not equal to the defined special value, if no special value is defined for the property, or if the property type does not include special value attributes.

The function can take one or more parameters to allow for the navigation through the object data structure of a module, beginning with a child of the object setting the context for the function call, and drilling down into this structure to the descendent object and, finally, a property of that object.

Parameters

@IS_SPECIAL_VALUE (Property 1 [, ..., Property N])	
Property 1	Required property parameter; if the only parameter to the function, this property's value will be compared to its special value attribute settings. If additional parameters are provided, the function will evaluate the next parameter in the context of this one, assuming it is a child member of this property.
Property N	Optional property parameter(s); with each specifying an object further down in the data hierarchy of the module. Each subsequent `Property N` provided must be a child to the parameter that immediately precedes it in the function call. The last parameter provided is compared to its special value attribute settings.

Supported Return Types
Boolean

@IS_VALID_DECIMAL_NUMBER

The IS_VALID_DECIMAL_NUMBER function takes a single parameter and returns true if the value is a valid decimal. If the value returned when evaluating this parameter as a decimal number is NaN (Not a Number), the function will return false.

Parameters

@IS_VALID_DECIMAL_NUMBER (Decimal)	
Decimal	Required decimal number parameter; contains the value to be evaluated as a valid decimal number.

Supported Return Types
Boolean

@LASTSCANVALUE

The LASTSCANVALUE returns the last value scanned by the client device and processed by the mobile application. If called prior to a value being scanned in or on a device without scanning capabilities, the function will return an empty string. The last scanned value will be returned regardless of where or when the function is called. Exiting and restarting the mobile application will remove the scanned value, and the function will return null until a new value is scanned.

Parameters

@LASTSCANVALUE()	
	This function takes no parameters.

Supported Return Types
String

@MEMBER

The MEMBER function is used to search an object collection and to return the value of a property within that collection that matches the given search value. The first match found within the collection will be returned. The function takes two parameters; the first is an object collection, and the second is a property with a value that is to be located. The object instance with the same property name, data type, and value within the object collection is then found and the value of the object's key property is returned. If no match is found within the collection, the function returns the null equivalent for the context's data type.

Parameters

@MEMBER (Collection Property, Search Property)	
Collection Property	Required property parameter; references the object collection to be searched by the function.
Search Property	Required property parameter; references the property definition for which an exact match within the `Collection Property` object instances is to be searched.

Supported Return Types

- Boolean
- Integral Number
- Decimal Number
- String
- Property

@NEEDS_XMIT

The NEEDS_XMIT function takes one or more property parameters, expects each to be an object, and returns true if the last parameter is an object and that object or one of its descendent objects in the module data structure has a pending transaction. If multiple parameters are provided to the function, the first parameter is evaluated in the context of the function call.

Each subsequent parameter is evaluated in the context of the parameter that precedes it. If any parameter evaluates to any definition instance other than an object, the evaluation ends and the function returns false.

In most current implementations of this function, a single parameter is provided that is a target path to an object instance selected using the target browser. If a pending transaction targets this object or any of its descendent objects, this function will return true. Otherwise it returns false.

Parameters

@NEEDS_XMIT (Property 1 [, ..., Property N])	
Property 1	Required property parameter; evaluated by the function to determine if it is an object instance first, and if a pending transaction exists that targets this object or any of its descendent object instances second. In current implementations and uses for this function, this is the only parameter provided in most cases.
Property N	Optional property parameters; each is evaluated in the context of the parameter before it in the function's parameter list. These parameters are expected to evaluate to an object instance. The last parameter in this list is evaluated by the function for pending transactions targeting it or any of its descendent object instances.

Supported Return Types
Boolean

@SCREENFIELDVALUE

The SCREENFIELDVALUE function returns the current value of a field on the current detail screen. The name of the field whose value is desired is the only parameter to this function. This name value is the internal name that uniquely identifies the field definition within the parent detail screen.

Parameters

@SCREENFIELDVALUE (Field Name)	
Field Name	Required string parameter; contains the name of the detail screen field definition whose current value is to be returned. This is the internal name of the field definition.

Supported Return Types

- Boolean
- Integral Number
- Decimal Number
- String

@SCREENFIELDNAME

The SCREENFIELDNAME function is supported only in update rules and returns the name of the current detail screen field being updated by the rule within which the function is called. If this function is called in a rule not used in an update rule, its return value is undefined. The function takes no parameters.

Parameters

@SCREENFIELDNAME()	
	This function takes no parameters.

Supported Return Types
String

@SCREENNAME

The SCREENNAME function returns the name of the screen definition with the focus at the time of its evaluation. This function takes no parameters.

Parameters

@SCREENNAME ( )	
	This function takes no parameters.

Supported Return Types
String

@SCREENSETNAME

The SCREENSETNAME function returns the name of the current screen set displayed to the user. The value returned is the internal definition name of the screen set that uniquely identifies the screen set definition within the module. This function takes no parameters and supports the string return type. The current screen set is the parent definition to the screen with the current focus.

Parameters

@SCREENSETNAME ( )	
	This function takes no parameters.

Supported Return Types
String

@SIZE

The SIZE function returns the number of object instances in a given collection property and, optionally, can count only those where some condition is true, returning this count value or a Boolean indication of whether any objects within the collection were counted. The function takes two parameters. The first parameter is required and is the object collection property to be counted. The second parameter is an optional Boolean parameter that is evaluated once for each object in the collection. It is used to include only certain members of the collection in those counted. Specifically, each object for which the second parameter evaluates to true will be counted, and those for which it evaluates to false will not be.

Parameters

@SIZE (Object Collection [, Include Criteria])	
Object Collection	Required property parameter; contains the object collection to be counted by the function.
Include Criteria	Optional Boolean parameter; this term is evaluated once for and in the context of each object instance in `Object Collection`. The function will count each object instance for which `Include Criteria` returns true and exclude from the count each object for which false is returned.

Supported Return Types

- Boolean
- Integral Number
- String

@TRANSACTIONPROPERTYNAME

The TRANSACTIONPROPERTYNAME function returns the name of the transaction property for which the rule is being evaluated. This function supports the string return type. The value returned is the internal definition name of the transaction property. This function is only supported when part of a rule being evaluated is an initial value rule for a transaction property. The return from this function in any other context is undefined.

Parameters

@TRANSACTIONPROPERTYNAME ( )	
	This function takes no parameters.

Supported Return Types
String

@TYPE

The TYPE function returns the definition type of the last parameter it evaluates. This function takes one or more parameters, each evaluated as a property and each evaluated in the context of the parameter before it in the function's parameter list. The function supports the integral number, Boolean, and string return types.

In a Boolean context, the function will return true if the last parameter evaluated exists in the current context, or false if it does not exist. In a string context, this function returns the name of the definition type, e.g. object, transaction, etc. In an integral number context, this function returns the internal identifier for that definition type.

In current implementations this function has limited usage and may be deprecated in a future release.

Parameters

@TYPE (Property 1 [, ..., Property N])	
Property 1	Required property parameter; this is the first parameter evaluated by the function in the context of the function call. This parameter should be a child member of the definition setting the function's context. If it is not, the function will return false or null depending on the data type of the context. If this is the only parameter, and it exists, the return will be either true or the identifier for the definition type, depending on context.

@TYPE (Property 1 [, ..., Property N])	
Property N	Optional property parameter(s); each evaluated in the context of the parameter before it and assumed to be a child member of that previous property. Each additional `Property N` parameter will set the context for the next in the list. The last `Property N` parameter will be evaluated for its definition type, and the function will then return true or the identifier for this type if it exists, or else false or null if it does not exist.

Supported Return Types

- Boolean
- Integral Number
- String

@UI

This function has been deprecated and will not be supported in future releases. It should be replaced in all existing rule definitions with SCREENFIELDVALUE. The UI function takes a single parameter that is the field index for the currently displayed screen and returns the value of that field. The field index is a 0-based index. The parameter must be between 0 and the total number of field definitions on the current screen minus 1.

The value of the specified field is evaluated in the context of the function call. The UI function supports the Boolean, integral number, decimal number, string, and property return types.

Parameters

@UI (Field Index)	
Field Index	Required integral number parameter; specifies the field on the current detail screen by its index whose current value is to be returned. This is a 0-based index, with the first field at index 0 and the last field at the index position equal to the total number of fields on the screen minus 1. Index numbering occurs from left to right and top to bottom.

Supported Return Types

- Boolean
- Integral Number
- Decimal Number
- String

- Property

String Functions for Rules

The String category of rule function terms available within the rule definition provide the string parsing, concatenation, conversion, and other related behaviors for manipulating string values. These functions include those that return or remove sub-strings from source strings, convert strings to all upper or lowercase, trim whitespace, or return formatting characters, including new lines and tabs.

@CONCATENATE

The CONCATENATE function's behavior has changed with the release of Agentry 5.1. The function now takes two or more string parameters and concatenates them together, returning the resulting string. Previously this function took two strings and an integral number, with this third (and now deprecated) parameter limiting the length of the string returned by the function. This same behavior can now be provided by returning the result from CONCATENATE to the LEFT string function.

Parameters

@CONCATENATE(String 1, [, ..., String N])	
String 1	Required string parameter; contains the string value to which the remaining parameter(s) will be appended.
String N	Optional string parameter(s); each containing a string value to be appended to the resulting string returned by the function.

Supported Return Types
String

@FIND

The FIND function searches a source string for a provided sub-string and, depending on the context of the function call, returns one of the sub-string, the position of its first character within the source string, or an indicator as to whether or not it was located. This search can optionally be case-insensitive and may begin at the beginning of the source string or at some character position within the source string.

Note that this function no longer works with object collection properties. This functionality is now provided by the new rule function COLLECTION_FIND. Upgrades of projects from previous platform versions will be modified with the replacement of FIND with COLLECTION_FIND in any rule definition. This will happen as a part of the standard upgrade process built into the Agentry Editor and should require no additional actions on the part of the developer.

When FIND is called in a string context, the function will search a source string for a sub-string, returning that sub-string when found or an empty string if not found.

When FIND is called in an integral number context, the function will search a source string for a sub-string, returning the position of the first character of the sub-string within the source string when found, or -1 if not found. The first character of the string is at position 0.

When FIND is called in a Boolean context, the function will search a source string for a sub-string, returning true when found and false when not found.

Parameters

@FIND (Source String, Search String [, Case Sensitive [, Start Position]])	
Source String	Required string parameter; contains the string value to be searched by the function.
Search String	Required string parameter; contains the string value to search for within `Source String`. If `Search String` contains more characters than `Source String`, no sub-string will be found.
Case Sensitive	Optional Boolean parameter; when specified, determines whether or not the search should consider or ignore case. When this parameter is true or not present, the search will be case-sensitive. When this parameter is false, case will be ignored.
Start Position	Optional integral number parameter; when specified, contains the character position within `Source String` to begin the search for `Search String`. The first character in `Source String` is at position 0. If this parameter is not provided, the search will always begin with the first character of `Source String`.

Supported Return Types

- Boolean
- Integral Number
- String

@LEFT

The LEFT function returns a sub-string from a specified source string, with the length of the sub-string specified and beginning with the left-most character in the string. This function takes two parameters: a source string and the number of characters to extract from the source. If this length is equal to or greater than the length of the source string, the entire string is returned. If the length is 0 or less, an empty string is returned.

Parameters

@LEFT (Source String, Length)	
Source String	Required string parameter; contains the source string from which a sub-string will be extracted.
Length	Optional integral number parameter; contains the maximum number of characters to extract from `Source String`. If this value is equal to or longer than `Source String`, the entire string is returned. If `Length` is 0 or less, an empty string will be returned.

Supported Return Types
String

@LENGTH

The LENGTH function determines the length of the given string and based on context, returns either the number of characters in the string, or an indicator of whether or not the string is empty. In an integral context, the number of characters is returned as an integer. In a string context, the number of characters is returned as a string. In a Boolean context, true is returned if the source contains at least one character, and false is returned when the string is empty.

Parameters

@LENGTH (Source String)	
Source String	Required string parameter; contains the source string whose length will be determined.

Supported Return Types

- Boolean
- Integral Number
- String

@LOWERCASE

The LOWERCASE function converts all alphabetical characters in the source string to lower case and returns the result. Any non-alphabetical characters are unchanged.

Parameters

@LOWERCASE (Source String)	
Source String	Required string parameter; contains the string value to be converted to lowercase.

Supported Return Types
String

@MID

The MID function parses a source string to return a sub-string that begins at a specified position and contains at most the specified number of characters. The first parameter to the function is the source string from which the sub-string is extracted. The second and third parameters are optional and specify the start and end position within the source string from which the sub-string is to be extracted. The first character in the source string is at position 0. The default starting position is 0 if not provided. If the number of characters to return is not provided, the default is the remaining length of the source string after the start position.

Parameters

@MID (Source String [, Start Position [, Max Length]])	
Source String	Required string parameter; contains the source string from which the sub-string will be extracted.
Start Position	Optional integral number parameter; contains the zero-based position of the first character within `Source String` for the sub-string to be extracted. If this parameter is not provided, the default start position is 0 and the entire `Source String` value will be returned.
Max Length	Optional integral number parameter; contains the maximum number of characters to return from `Source String` after `Start Position`. If this parameter is not provided, all characters after `Start Position` will be returned as the sub-string.

Supported Return Types
String

@NEWLINE

The NEWLINE function returns the command characters <CR> <LF>, (0x0D 0x0A), which result in a Windows new line. The return value of this function can be concatenated with other strings for formatting purposes.

Parameters

@NEWLINE()	
	This function takes no parameters.

Supported Return Types
String

@REMOVE

The REMOVE function searches a given source string, removes all instances of a provided search string, and returns the result. As optional behaviors, parameters can be provided to specify whether or not the search is case-sensitive, and to specify the starting position within the source string to begin the search.

Parameters

@REMOVE (Source String, Search String [, Case Sensitive [, Start Position]])	
Source String	Required string parameter; contains the source string to be searched by the function.
Search String	Required string parameter; contains the sub-string to be removed from `Source String`.
Case Sensitive	Optional Boolean parameter; when provided, indicates whether or not the search of `Source String` for `Search String` should be case-sensitive. If this value is true or not specified, the search is case-sensitive. If set to false, case is ignored.

@REMOVE (Source String, Search String [, Case Sensitive [, Start Position]])	
Start Position	Optional integral number parameter; when provided, specifies the zero-based position within `SourceString` to begin the search. The default is to begin at position 0.

Supported Return Types
String

@REPLACE

The REPLACE function searches a given source string for a provided search string and replaces each instance of the search string with a replacement string. By default, this search is case-sensitive and includes the entire source string. Both of these behaviors can be overridden based on optional parameters to the function.

Parameters

@REPLACE (Source String, Search String, Replace String [, Case Sensitive [, Start Position]])	
Source String	Required string parameter; contains the source string to be searched by the function.
Search String	Required string parameter; contains the string value to be searched for within `Source String`.
Replace String	Required string parameter; contains the string value to replace `Search String` within `Source String`.
Case Sensitive	Optional Boolean parameter; when specified, indicates whether the search should be case-sensitive. When true or if not provided, the search is case-sensitive. When false, case is ignored.

@REPLACE (Source String, Search String, Replace String [, Case Sensitive [, Start Position]])	
Start Position	Optional integral number parameter; when specified, indicates the zero-based position within `Source String` to begin the search. The default is to search the entire `Source String`. If `Start Position` is less than 0 or is greater than the number of characters in `Source String`, an empty string will be returned.

Supported Return Types

String

@RFIND

The RFIND function searches a string for a sub-string of characters beginning with the right-most character in a string and based on the function's context, returns the sub-string when found, the position of the first character of the sub-string within the source string, or an indictor of whether the sub-string was found. This search can optionally be case-insensitive and may begin at the right-most character of the source string, or somewhere within the source string by specifying the first position, counting from the left, to begin searching.

When RFIND is called in a string context, the function will search a source string for a sub-string, returning that sub-string when found, or an empty string if not found.

When RFIND is called in an integral number context, the function will search a source string for a sub-string, returning the position of the first character of the sub-string within the source string when found, or -1 if not found. The left-most character of the source string is at position 0.

When RFIND is called in a Boolean context, the function will search a source string for a sub-string, returning true when found, or false if not found.

Parameters

@RFIND (Source String, Search String [, Case Sensitive [, Start Position]])	
Source String	Required string parameter; contains the string value to be searched by the function.

@RFIND (Source String, Search String [, Case Sensitive [, Start Position]])	
Search String	Required string parameter; contains the string value to search for within `Source String`.
Case Sensitive	Optional Boolean parameter; when provided, specifies whether or not the search of `Source String` for the `Search String` value should be case-sensitive. When this value is true or not provided, the search is case-sensitive. If this value is false, case is ignored.
Start Position	Optional integral number parameter; when provided, specifies the zero-based character position, counting from the left, within `Source String` to begin the search. If this value is not provided, the search begins at the right-most character within `Source String`. If this value is 0 or negative, an empty string is returned. If this value is equal to or greater than the total number of characters within `Source String`, the entire string is searched.

Supported Return Types

- Boolean
- Integral Number
- String

@RIGHT

The RIGHT function returns a sub-string of specified length from a given source string, beginning at the right-most character of the source string and counting back towards the beginning. The function takes two parameters. The first is the source string from which the sub-string is extracted. The second is the number of characters in the sub-string. If the specified number of characters is greater than the length of the source string, the entire source string is returned. If the specified number of characters specified is 0 or less, an empty string is returned.

Parameters

@RIGHT (Source, MaxLength)	
Source String	Required string parameter; contains the string value from which the sub-string will be extracted.

@RIGHT (Source, MaxLength)	
Max Length	Required integral number parameter; contains the maximum number of characters to return from `Source String`. If this value is greater than the number of characters in `Source String`, `Source String` is returned in its entirety. If `Max Length` is 0 or negative, an empty string will be returned.

Supported Return Types
String

@TAB

The TAB function takes no parameters and returns a tab <HT> character (0x09). This function is most often used to insert a tab into a text value for the purposes of formatting.

Parameters

@TAB()	
	This function takes no parameters.

Supported Return Types
String

@TRIM

The TRIM function removes any leading or trailing whitespace characters from a given source string. The following are considered whitespace characters and will be removed from the beginning and end of the given source string:

- Horizontal tab
- Vertical tabs
- Space
- Newline
- Carriage return
- Formfeed

Parameters

@TRIM (Source String)	
Source String	Required string parameter; contains the value from which all leading and trailing whitespace will be removed.

Supported Return Types
String

@UPPERCASE

The UPPERCASE function converts all alphabetical characters in a given source string to upper case and returns the result. Any non-alphabetical characters are returned unchanged.

Parameters

@UPPERCASE (Source String)	
Source String	Required string parameter; contains the string value to be converted to upper case by the function.

Supported Return Types
String

System Functions for Rules

The System function category of rule functions available in the rule definition provide functionality related mostly to accessing values from the mobile application as a whole, those retrieved by interacting with the client device or specific hardware components of the device, or by interacting directly with the client device's operating system.

These functions include those that return date and time values, interact with a client device's GPS system, return on-line state information about the mobile application, and other similar items.

@DATE

The DATE function returns either the current system date of the client device, or the date value specified as a parameter to the function. The function can also take an optional format parameter when called in a string context. The function supports the integral number, decimal number, and string return types.

When called in an integral or decimal number context, the DATE function will return the number of days before (negative number) or after (positive number) the date January 1, 2001.

This is the Agentry epoch date. When called in a string context the function will return the date value in the default format of the client device's locale.

The function's first parameter, if provided, is a string and contains the date value to be returned by the function. The second parameter is also optional and is evaluated as a string. It can contain one or more of the following date format tokens, which will be used to then format the date value returned by the function. Note that this format parameter is ignored when the function is called in any context other than string:

Table 1. Rule Date Format Tokens - *All date format tokens are case sensitive*

Date Token	Description
d	Day of month as digits with no leading zero for single-digit days.
dd	Day of month with leading zero for single-digit days.
ddd	Day of week as three letter abbreviation. The function uses the LOCALE_SABBREVDAYNAME value associated with the device's specified locale.
dddd	Day of week as its full name. The function uses the LOCALE_SDAYNAME value associated with the device's specified locale
M	Month as digits with no leading zero for single-digit months.
MM	Month as digits with leading zero for single-digit months.
MMM	Month as three letter abbreviation. The function uses the LOCALE_SABBREVMONTHNAME value associated with the device's specified locale.
MMMM	Month as its full name. The function uses the LOCAL_SMONTHNAME value associated with device's specified locale.
y	Year as last two digits with no leading zero for years less than 10.
yy	Year as last two digits with leading zero for years less than 10.
yyyy	Full four digit year value.
non-token characters	Any non-token character within the format string is passed through as is; e.g in the following string: `d={MM/dd/yyyy}` the resulting string will contain the slash characters separating each date element: `11/17/1967`

Parameters

@DATE ([Date String [, Format Tokens]])	
Date String	Optional string parameter; contains the date value to be returned by the function. To format the current system date, this parameter may be set to a second call to the DATE function. A data definition such as a property or global of type Date or Date and Time may be used for this parameter. The time portion of the value will be truncated. A string may be used, provided the date is in the format `MM/dd/yyyy`. A numeric value may be passed for this parameter, in which case it will be treated as the number of days before or after the Agentry epoch date.
Format Tokens	Optional string parameter; contains the date format tokens that will format the function's return value when called in a string context. When the function is called in any other context, this parameter is ignored. A `DateString` must be specified before `FormatTokens` can be provided.

Supported Return Types

- Integral Number
- Decimal Number
- String

@DATE_AND_TIME

The DATE_AND_TIME function returns either the current system date and time of the client device, or the date and time value specified as an optional parameter to the function. The function can also take an optional format parameter when called in a string context. The function supports the integral number, decimal number, and string return types.

When called in an integral or decimal number context, the function will return the number of seconds before (negative value) or after (positive value) the date and time of January 1, 2001 - 12:00:01 AM. This is the Agentry epoch date and time. When called in a string context, the function will return a date and time value in the default format for the client device's locale.

The function's first parameter, if provided, is a string and contains the date and time value to be returned by the function.

The second parameter is also optional and is evaluated as a string. It can contain one or more of the date and time format tokens. The syntax for this parameter is as follows:

```
d={date format tokens} t={time format tokens}
```

The tokens within the curly braces will be used to format the date and time value returned by the function:

Table 2. Rule Date Format Tokens - ***All date format tokens are case sensitive***

Date Token	Description
d	Day of month as digits with no leading zero for single-digit days.
dd	Day of month with leading zero for single-digit days.
ddd	Day of week as three letter abbreviation. The function uses the LOCALE_SABBREVDAYNAME value associated with the device's specified locale.
dddd	Day of week as its full name. The function uses the LOCALE_SDAYNAME value associated with the device's specified locale
M	Month as digits with no leading zero for single-digit months.
MM	Month as digits with leading zero for single-digit months.
MMM	Month as three letter abbreviation. The function uses the LOCALE_SABBREVMONTHNAME value associated with the device's specified locale.
MMMM	Month as its full name. The function uses the LOCAL_SMONTHNAME value associated with device's specified locale.
y	Year as last two digits with no leading zero for years less than 10.
yy	Year as last two digits with leading zero for years less than 10.
yyyy	Full four digit year value.
non-token characters	Any non-token character within the format string is passed through as is; e.g in the following string: `d={MM/dd/yyyy}` the resulting string will contain the slash characters separating each date element: `11/17/1967`

Table 3. Rule Time Format Tokens - ***All time format tokens are case sensitive***

Time Token	Description
h	Hour of day in 12 hour clock format with no leading zero for single digit hours.
hh	Hour of day in 12 hour clock format with leading zero for single digit hours.
H	Hour of day in 24 hour clock format with no leading zero for single digit hours.
HH	Hour of day in 24 hour clock format with leading zero for single digit hours.
m	Minute of the hour with no leading zero for single digit minutes.
mm	Minute of the hour with leading zero for single digit minutes.

Time Token	Description
s	Seconds of the minute with no leading zero for single digit minutes.
ss	Seconds of the minute with leading zero for single digit minutes.
t	One character time marker string, such as A or P.
tt	Two character time marker string, such as AM or PM.
non-token characters	Any non-token character within the format string is passed through as is; e.g in the following string: `t={hh:mm:ss}` the resulting string will contain the colon characters separating each time element: `10:12:32`

Parameters

@DATE_AND_TIME ([Date Time String [, Format Tokens]])	
Date Time String	Optional string parameter; contains the date and time value to be returned by the function. To format the current system date and time, this parameter may be set to a second call to the DATE_AND_TIME function. A data definition such as a property or global of type Date, Date and Time, or Time may be used for this parameter. A string may be used, provided the date and time is in the format `hh:mm:ss MM/dd/yyyy`. A numeric value may be passed for this parameter, in which case it will be treated as the number of seconds before or after the Agentry epoch date and time.
Format Tokens	Optional string parameter; contains the date and time format tokens that will format the function's return value when called in a string context. When the function is called in any other context, this parameter is ignored. A `Date Time String` must be specified before `Format Tokens` can be provided.

Supported Return Types

- Integral Number
- Decimal Number
- String

@DISTANCE_MILES

The DISTANCE_MILES function takes two GPS location parameters and returns the total distance between them in miles as a decimal number. The distance returned is always 0 or a positive number. The function may return an invalid decimal value (NaN) if either of the two GPS location parameters to the function are invalid location values.

This function is intended for use on devices equipped with a GPS unit, though it will return a distance in miles for any two valid GPS location values.

Parameters

@DISTANCE_MILES (GPS Location 1, GPS Location 2)	
GPS Location 1	Required location parameter; contains one of the two GPS location values between which the distance will be calculated by the function.
GPS Location 2	Required location parameter; contains one of the two GPS location values between which the distance will be calculated by the function.

Supported Return Types
Decimal Number

@DISTANCE_KILOMETERS

The DISTANCE_KILOMETERS function takes two GPS location parameters and returns the total distance between them in kilometers as a decimal number. The distance returned is always 0 or a positive number. The function may return an invalid decimal value (NaN) if either of the two GPS location parameters to the function are invalid.

This function is intended for use on devices equipped with a GPS unit, though it will return a distance in kilometers for any two valid GPS location values.

Parameters

@DISTANCE_KILOMETERS (GPS Location1, GPS Location2)	
GPS Location 1	Required location parameter; contains one of the two GPS location values between which the distance will be calculated by the function.
GPS Location 2	Required location parameter; contains one of the two GPS location values between which the distance will be calculated by the function.

Supported Return Types
Decimal Number

@GPS_LOCATION

The GPS_LOCATION function returns the GPS location of the device's current position, optionally based on a maximum age for the GPS data. This function can take a single optional parameter of type integral number treated as the maximum number of seconds for the GPS data. This function will return an invalid location value if the client device is not equipped with a GPS unit or if the GPS unit is not accessible to the client application.

If the maximum age of the GPS data available to the function exceeds the maximum age parameter provided to the function, the function will query the GPS unit for a current location. The function will set all components of the location data type, including the location value itself, as well as the number of satellites and precision as reported by the GPS unit.

Parameters

@GPS_LOCATION ([Max Age])	
Max Age	Optional integral number parameter; specifies the maximum age in seconds of the GPS data for which a location should be returned. If this parameter is not provided, the function will return the most recent location.

Supported Return Types
Location

@IS_VALID_LOCATION

The IS_VALID_LOCATION function takes a single GPS location parameter and returns true or false based on whether or not the value of the parameter is a valid location value. If this parameter is a GPS location property type, the definition of that property's precision attributes will be used as a part of determining the parameters validity. Empty location values are always invalid.

Parameters

@IS_VALID_LOCATION (Location)	
Location	Required location parameter; contains the GPS location value to be checked for validity. If the value of this parameter is valid, the function will return true. If this parameter is a location property type, the property's precision attributes will be used to determine if the value is valid.

Supported Return Types
Boolean

@JAVASCRIPT

The JAVASCRIPT function is provided to allow JavaScript logic to be embedded within a rule definition. Included in the JAVASCRIPT function's behavior is the ability to pass values to the script being processed from the rule definition. The value returned by the JavaScript logic will be the value returned by the function. This function takes one required parameter, and as many additional parameters as needed to pass in additional values. The supported return types of the JAVASCRIPT function are Boolean, integral number, decimal number, or string.

The first parameter to the JAVASCRIPT function is the JavaScript logic to be processed. This parameter may be any string value from any source within the application. In most cases, it is recommended that the rule term JavaScript Text is used, which is an available item within the rule editor. The main purpose for this term is that it provides a large text box control to display multiple lines of text, making it easier to write and edit JavaScript logic.

Additional parameters to the JAVASCRIPT parameter are referenced within the JavaScript logic through the zero-based array `argv[]`. This array is available in all JavaScript logic processed by the JAVASCRIPT rule function. The second parameter to JAVASCRIPT is stored in the `argv[0]` element, the third is in `argv[1]`, and so on.

The data types of the additional parameters are strings. The parameters will be converted to this data type and will be passed to the JavaScript as such. These values can then be converted to a different data type where necessary within the JavaScript logic.

The JavaScript engine used to process the script logic is SpiderMonkey. Note that the Data Object Model (DOM) and the `XMLHttpRequest` object are not implemented as a part of this JavaScript support.

The usage of JavaScript within rule definitions is intended to be supplemental functionality. It is not recommended that all rules be written exclusively with JavaScript, as the processing of such script files is less efficient than the processing of rule definitions. The main intent is to allow a developer to implement certain pieces of logic using JavaScript wherever it is deemed appropriate to do so.

Parameters

@JAVASCRIPT (JavaScript [, ..., ArgV String N])	
JavaScript	Required string parameter; contains the JavaScript logic to be processed by the JavaScript engine. In most cases, this logic will be contained in the special rule term JavaScript text, though any data term that may be safely converted to a string and that contains valid JavaScript may be used.
ArgV String N	Optional string parameter(s); contains value(s) passed to the JavaScript and available in the `argv[]` array. The value of the second JAVASCRIPT parameter, i.e. `ArgV String 1`, is available within the array element `argv[0]`, the next optional JAVASCRIPT parameter's value is stored in `argv[1]`, and so on. The members of this array are strings and should be converted within the JavaScript logic where necessary.

Supported Return Types

- Boolean
- Integral Number
- Decimal Number
- String

@LATITUDE

The LATITUDE function returns the latitude of a provided GPS location in degrees. The function takes a single location parameter from which the latitude portion of the coordinates is returned. This function will return an invalid decimal value (NaN) if the GPS location parameter is invalid.

This function is primarily intended for use on client devices equipped with GPS units, though it will return a latitude for any valid GPS location value provided. It does not interact with the GPS unit directly.

Parameters

@LATITUDE (GPS Location)	
GPS Location	Required location parameter; specifies the GPS location from which the latitude will be calculated. If this is not a valid GPS location value, the function will return an invalid decimal value (NaN).

Supported Return Types
Decimal Number

@LOCATION

The LOCATION function takes two decimal number parameters, treated as degrees of latitude and longitude, and returns the GPS location for those two values. The function may return an invalid GPS location if either the latitude or longitude parameters are invalid values.

Longitude values must be in the range -179 and 180, inclusive. Latitude values must be in the range of -90 and 90, inclusive. The returned location value includes the GPS location, a satellite count of 50, and a precision of 1.0.

This function is intended for use on devices equipped with a GPS unit, though it will return a GPS location for a given valid set of longitude and latitude values.

Parameters

@LOCATION (Latitude, Longitude)	
Latitude	Required decimal number parameter; provides the latitude in degrees of the location to be returned as a GPS location by the function. Valid latitude values are in degrees with a range of -90 to 90, inclusive.
Longitude	Required decimal number parameter; provides the longitude in degrees of the location to be returned as a GPS location by the function. Valid longitude values are in degrees in the range -179 to 180, inclusive.

Supported Return Types
Location

@LONGITUDE

The LONGITUDE function returns the longitude for a given GPS location. The function takes a single location parameter from which the longitude portion of the coordinates is returned. This function will return an invalid decimal value (NaN) if the GPS location parameter is invalid.

This function is primarily intended for use with devices equipped with a GPS unit, though it will return a longitude value for any valid GPS location value provided. It does not interact with the GPS unit directly.

Parameters

@LONGITUDE (GPS Location)	
GPS Location	Required location parameter; contains the GPS location from which the longitude in degrees is calculated by the function.

Supported Return Types
Location

@MODULE_ENABLED

The MODULE_ENABLED function returns a Boolean value indicating whether or not the module specified by name is enabled or disabled. The function will return true if the module is enabled. It will return false if the module is disabled or is not present. The function takes a single string parameter containing the name of the module definition to be checked.

Parameters

@MODULE_ENABLED (Module Name)	
Module Name	Required string parameter; contains the definition name of the module to be checked by the function.

Supported Return Types
Boolean

@OFFLINE

The OFFLINE function returns a value indicating whether or not the client application is in an off-line state. The function supports the Boolean and integral number return types. In a Boolean context, this function will return true of the client application is in an off-line state, and false if it is in an on-line state. In an integral number context, the function will return a non-zero value of the client application is in an off-line state, and zero if in an on-line state.

Parameters

@OFFLINE ( )	
	This function takes no parameters.

Supported Return Types

- Boolean
- Integral Number

@TIME

The TIME function returns either the current system time of the client device, or the time value specified as an optional parameter to the function. The function can also take an optional format parameter when called in a string context. The function supports the integral number, decimal number, and string return types.

When called in an integral or decimal number context, the function will return the number of seconds before (negative value) or after (positive value) the time 12:00:00 AM. This is the Agentry epoch time. When called in a string context, the function will return a time value in the default format for the client device's locale.

The function's first parameter, if provided, is a string and contains the time value to be returned by the function.

The second parameter is also optional and is evaluated as a string. It can contain one or more of the time format tokens. The syntax for this parameter is as follows:

```
t={time format tokens}
```

The tokens within the curly braces will be used to format the time value returned by the function:

Table 4. Rule Time Format Tokens - *All time format tokens are case sensitive*

Time Token	Description
h	Hour of day in 12 hour clock format with no leading zero for single digit hours.
hh	Hour of day in 12 hour clock format with leading zero for single digit hours.
H	Hour of day in 24 hour clock format with no leading zero for single digit hours.
HH	Hour of day in 24 hour clock format with leading zero for single digit hours.
m	Minute of the hour with no leading zero for single digit minutes.
mm	Minute of the hour with leading zero for single digit minutes.
s	Seconds of the minute with no leading zero for single digit minutes.
ss	Seconds of the minute with leading zero for single digit minutes.
t	One character time marker string, such as A or P.

Time Token	Description
tt	Two character time marker string, such as AM or PM.
non-token characters	Any non-token character within the format string is passed through as is; e.g in the following string: `t={hh:mm:ss}` the resulting string will contain the colon characters separating each time element: `10:12:32`

Parameters

@TIME ([Time String [, Format Tokens]])	
Time String	Optional string parameter; contains the time value to be returned by the function. To format the current system time, this parameter may be set to a second call to the TIME function. A data definition such as a property or global of type Time or Date and Time may be used for this parameter. The date portion of a Date and Time value will be truncated. A string may be used provided the time is in the format `hh:mm:ss`. A numeric value may be passed for this parameter, in which case it will be treated as the number of seconds before or after the Agentry epoch time.
Format Tokens	Optional string parameter; contains the time format tokens that will format the function's return value when called in a string context. When the function is called in any other context, this parameter is ignored. The `Time String` parameter must be specified before `Format Tokens` can be provided.

Supported Return Types

- Integral Number
- Decimal Number
- String

@TIME_TICKS

The TIME_TICKS function returns the number of milliseconds since the client device booted, excluding any time the device was in sleep or hibernation modes, or any similar modes of operation. The function supports the integral number, decimal number, and string return types.

The difference between the return values of two separate calls to this function can be used to calculate duration values for various purposes.

Parameters

@TIME_TICKS ( )	
	This function takes no parameters.

Supported Return Types

- Integral Number
- Decimal Number
- String

@USERID

The USERID function returns the user ID value entered to log into the client application. This value is returned as a string.

Parameters

@USERID ( )	
	This function takes no parameters.

Supported Return Types

String

Table Functions for Rules

The Table functions category of rule functions available within the rule definition provide access to the complex tables and data tables of an application. This category consists of three functions. The first two return a record from a complex table or data table based on some search criteria. The third returns the total number of records within a complex table.

@COMPLEXTABLE

The COMPLEXTABLE function searches the specified complex table for a single record and returns a single field from that record. The parameters to this function include, at a minimum, the name of the complex table to be searched and the value to search for within the records. If only these two values are provided, the function will search the complex table using the table's primary index, returning the field upon which the primary index has been defined from the record found.

As optional parameters to the function, the index to search upon and the field to return from the matching record can be specified by passing in the definition names of each. While optional, these parameters are provided in most use cases.

An additional variation on the parameters passed to the function is in the situation when a search index is specified, and that index is a child to another index within the table. In this scenario, additional search values must also be provided to the function. The requirement is for each index, starting with the one specified up to the top-level index (one that has no parent index) in the structure, there must also be a corresponding search value provided to the function.

As an example, assume a complex table with three indexes defined: A, B, and C. Index C is a child index to B, and B in turn is a child index to A, which is a top-level index in the table. When searching this table with the COMPLEXTABLE function, if index C is specified as the search index, search values for indexes A, B, and C must be provided. During the search, the function will begin by finding records that match on index A, then within that set those records that match index B, and finally within that sub-set the first record that matches index C.

Parameters

@COMPLEXTABLE (Table Name, Search Value, [Parent Search Value N,] [Search Index, Return Field])	
Table Name	Required string parameter; contains the name of the complex table to be searched by the function.
Search Value	Required string parameter; contains the value used to search the complex table records.
Parent Search Value N	String parameter(s); required and specified only when `Search Index` is a child index. One `Parent Search Value` must then be specified for each index above the `Search Index` in the index hierarchy.
Search Index	Optional string parameter; provides the name of the index within the complex table used by the function to locate the desired record based on the `Search Value` and possibly `Parent Search Values`. If this parameter is not provided, the primary index of the complex table is used by the function.
Return Field	Optional string parameter; provides the name of the complex table field whose value is returned from the record found by the function. If this parameter is not specified, the field for which the primary index of the complex table is defined is the default field value returned.

Supported Return Types

- Boolean

- Integral Number
- Decimal Number
- String

@TABLE

The TABLE function searches a data table for a record with the specified key value and returns the value field for the matching record. The search performed by this function is a record by record search, attempting to match the provided search value with the key field of each record. Data tables contain no indexes or guaranteed record order, and therefore the search is performed in a first to last manner. Data tables with large numbers of records will take longer to search, both by the TABLE function as well as any other methods. The value field returned is converted to the data type of the function's context, which may be integral number, Boolean, or string.

Parameters

@TABLE (Table Name, Search String)	
Table Name	Required string parameter; contains the name of the data table the function will search.
Search String	Required string parameter; contains the key value the function will use to search the data table records.

Supported Return Types

- Boolean
- Integral Number
- String

@TABLE_COUNT

The TABLE_COUNT function takes a single string parameter that is assumed to be the name of a data table within the application. The function counts the number of records in the named table and returns the result. This function supports the integral number return type.

Parameters

@TABLE_COUNT (Table Name)	
Table Name	Required string parameter; contains the name of the data table whose total number of records is to be returned. If this parameter contains a name that does not match any defined data tables, the function will return zero.

Supported Return Types
Integral Number

Syclo Data Markup Language

When synchronizing data between the mobile application and the back end system, it is necessary to have access to the mobile application's data values. This access is provided in Agentry using the Syclo Data Markup Language, or SDML. The SDML is a markup language consisting of tags that provide access to the data values of the mobile application. Additionally, the SDML includes a full set of functions, or function tags, that can be used to perform logical operations in relation to this values or to drive the overall logic the Agentry Server will execute against the back end system.

The SDML tags used during synchronization are a part of the text within the scripts for step definitions defined for SQL Database, HTTP-XML, and File system connection types. Also, the synchronization components of data tables and complex tables for each of these system connection types can contain SDML. In addition to SQL Step definitions, other `.sql` script files run by the Server may also contain SDML tags. Steps defined for Java Virtual Machine system connections also include the ability to access SDML tags, but these tags may not be contained directly in the source code of the Java Steplet files used by these steps.

The Agentry Server will pre-process the script files of steps containing SDML markup. This processing is referred to as tag expansion. Each tag within the script is expanded, with the value it represents replacing the tag at the exact position of that tag within the file. Function tags are expanded with the results of their expansion being placed in the exact position of the function call within the file. Once the tag expansion has completed, the resulting text is submitted to the back end system for processing.

The two categories of tags within the SDML are data tags and function tags. Data tags represent data values available to the script file based on when it is executed. This information must be known when writing the script in which the SDML will be contained. For step definitions the values in scope are dictated by the step usage definition running them. For `.sql` scripts run by the server, but not a part of the step definition, the values in scope will vary depending on how that script is used. Certain values are globally available, such as the user ID as entered by the user to log into the Agentry Client.

Function tags are globally available, with certain exceptions. Function tags provide the logical, mathematical, string manipulation, and other similar functionality to the SDML. Function tags can take values passed in as arguments, parameters, or expressions. These values are processed by the function during tag expansion, with the resulting value of the function call being placed within the script.

Following is a basic example of a simple SQL statement containing SDML data tags:

```
SELECT
    A.FIELD1,
    B.FIELD2,
    C.FIELD3,
FROM
    TABLEA A,
    TABLEB B,
    TABLEC C
WHERE
    A.NAME = '<<user.agentryID>>'    AND
    A.ACCTNUM = '<<object.acctnum>>'    AND
    B.ACCTNUM = A.ACCTNUM     AND
    C.ACCTNUM = B.ACCTNUM
```

In this example, the value `<<user.agentryID>>` is replaced with the user ID as entered when the user logged into the Agentry Client. The data tag`<<object.acctnum>>` will be replaced with the value of the `acctnum` property of the object currently being processed.

SDML Data Tags Overview

Data tags provide the access to the production data of the application within the synchronization components of the mobile application. This includes access to property values, global definition values, query constants, and client and server information system information. Each of these items just listed are referred to as the data tag's data source.

In addition to the tag's data source, all data tags also have a certain data type. The data source and the data type of a tag combine to give the data tag its overall behavior. This behavior includes how the value is expanded during data tag expansion, as well as the parameters that the tag will support. By and large, the data tags created for properties are the tags that have the most complex behavior.

Data tags that provide access to data other than from properties or globals are strings. Those data tags that are based on a property definition are one of several data types, based on the property data type.

Data Tag Data Types

Within the SDML, all data tags have a data type. This data type affects how the data tag is processed during tag expansion. Do not confuse the data tag's type with the data type for that same value in the back end system. When the Agentry Server has completed tag expansion the resulting values within the script are plain text. At this point, the methodology for denoting the value’s data type will depend on the type of back end system in use.

As an example, for a database system connection, data tags with a data type of date and time will be expanded with the date and time conversion function for that database as a part of the text, as in:

```
<<object.statusDate>>
```

If this data tag is a date and time and used in a script for an Oracle database, it would expand to:

```
to_date(‘01/12/2004 14:23:45’, ‘mm/dd/yyyy hh24:mi:ss’)
```

As you can see, the date and time value has been wrapped in the `to_date` function call for Oracle, which converts string values into dates and times.

The data types for data tags are as follows. Note that all data types other than string are applicable only to tags for global and property values:

- String
- Integral Number
- Decimal Number
- Boolean
- Date
- Time
- Date and Time
- Signature

The Scope of Data Tags

The data tags within the SDML may or may not be valid in one area verses another. The scope of a data tag will vary from one to the next. Certain tags are only valid in steps used by a fetch. Others are only available in steps used by transactions. Still others are available globally. It is important to note that for scripts within step definitions, the scope for a data tag is determined by the type of step definition and also the step usage definition referencing the step to be processed at run time. For example, the data tags that are in scope for a step used by an object read step will be different from the tags that are in scope for a step used by a transaction’s server update step. The terms used to describe a data tag’s scope are:

- **Global**: A data tag with Global scope is valid in all scripts processed by the Agentry Server. NOTE: Do not confuse the term Global used here to denote a tag’s scope with the definition type global. Values for a global definition do have a global scope. However, there are other data tags that also are globally available to the application’s synchronization components.

- **Definition-*Type***: A data tag with the Definition-*Type* scope is one that is in scope only for a certain type of definition, such as a Transaction or Object. The Type portion of this scope specifies the definition type for which the data tag is applicable.
- **Definition**: A data tag with the Definition scope is one that is only in scope for instances of a specific definition. For example, the data tags in scope for an Object named Customer will be different than those for an Object named Order. Data tags that have a Definition scope are those that provide access to the properties of an object.

<<user>> Data Tag Container

The user data tag is a container tag for several user-realted values. Each of these values is represented by a member tag within the user containter. Of these members, two contain members of their own. All members of the user data tag container are available in all scripts processed by the Agentry Server.

Table 5. <<user>> Data Tag Members

Tag Name	Description
<<user.name>>	Returns the name of the client user. By default this will be the ID entered by the user to log into the client. This value may be overridden.
<<user.deviceID>>	Returns the device ID for the client device upon which the Agentry Client is running. This value is set by the original equipment manufacturer.
<<user.agentryID>>	Returns the user ID entered to log into the Agentry Client. This value cannot be overriden during synchronization.
<<user.client>>	A data tag container within <<user>>. Member data tags provide information about the client application and client device as provided by the Agentry Client during synchronization.
<<user.info>>	A data tag container within <<user>>. Member data tags are variable and set during synchronization based on the logic of the mobile application.

<<user.client>> Data Tag Container

The `<<user.client>>` data tag container is a member of the `<<user>>` container. Members of `<<user.client>>` provide information about the client device’s hardware and software, and information about the Agentry Client application.

Many of the member data tags of `<<user.client>>` are valid only on client devices running a Windows operating system. Such members names begin with the text `Win_` and will return empty strings for any other client device type. This members providing information about the Agentry Client software or the host system are invalid for web browser clients.

Table 6. <<user.client>> Member Data Tags

Tag Name	Description
`<<user.client.time>>`	Returns the date and time of the client device when the transmission between the Agentry Client and Server began.
`<<user.client.Language>>`	Returns the two character abbreviation for the client device's configured locale.
`<<user.client.Win_MajorVersion>>`	Returns the major version number of the Windows operating system installed on the client device.
`<<user.client.LocaleID>>`	Returns the local ID for the configured locale of the client device.
`<<user.client.Win_ServicePack>>`	Returns the service pack installed on the Windows operating system on the client device.
`<<user.client.timeDifference>>`	Returns the difference in time's between the client device and the Agentry Server's host system. This value is represented in number of seconds where negative values indicate the client is behind the server.
`<<user.client.Win_MinorVersion>>`	Returns the minor version number of the Windows operating system installed on the client device.
`<<user.client.FirstLogin>>`	Returns the text value true or false based on whether the current transmit is the result of the user's initial login to the Agentry Client. This value is representative of the fist time a user transmits from a given client device.
`<<user.client.Platform>>`	Returns the platform type of the client device.
`<<user.client.timeZone>>`	Returns the time zone to which the client device has been set.
`<<user.client.timeZoneBias>>`	Returns the difference in seconds between the client's time zone and Greenwich Mean Time (GMT).

Tag Name	Description
`<<user.client.screenHeight>>`	Returns the height of the client device's screen in pixels.
`<<user.client.screenWidth>>`	Returns the width of the client device's screen in pixels.
`<<user.client.PreviousUser>>`	Returns the text true or false based on whether the current synchronization processing is a part of a previous user transmit resulting from a user change on the Agentry Client.
`<<user.client.Win_ComputerName>>`	Returns the network name of the client device.
`<<user.client.Win_PlatformID>>`	Returns the platform ID of the client device. The specific value is dependent on the system and OEM settings.
`<<user.client.Country>>`	Returns the abbreviated country name for the client device, as indicated by the device's locale settings.
`<<user.client.Win_OS>>`	Returns the type of Windows operating system (e.g. Mobile, XP, etc.) of the client device.
`<<user.client.clientTime>>`	Returns the current time of the client device when the transmission began.
`<<user.client.ClientVersion>>`	Returns the Agentry Client executable's Agentry version number, such as 6.0.0.0.
`<<user.client.Win_UserName>>`	Returns the Windows account login under which the client device is currently running. This tag is only valid on Windows devices where an account name is entered. It will return an empty string for all other device types.
`<<user.client.Win_BuildNumber>>`	Returns the build number of the Windows operating system installed on the client device.
`<<user.client.clientTimeZone>>`	Returns the time zone configured on the client device.

Tag Name	Description
`<<user.client.WinCE_Platform>>`	Returns the platform type of the client device. This value is valid only for client devices running a mobile version of the Windows operating system.
`<<user.client.TestEnvironment-Version>>`	Returns the version number of the Agentry Test Environment. This value is valid only when the client is the ATE. Returns an empty string for all other clients.
`<<user.client.Win_ProcessorLe-velCode>>`	Returns the processor level code of the client device's processor. This value is dependent on the original equipment manufacturer.
`<<user.client.Win_ProcessorRevi-sion>>`	Returns the processor revision of the client devices processor. This value is dependent on the original equipment manufacturer.
`<<user.client.clientTimeZoneDif-ference>>`	Returns the difference in seconds between the client's time zone and Greenwich Mean Time (GMT).
`<<user.client.Win_ProcessorArch-titecture>>`	Returns the processor architecture of the client device's processor. This value is dependent on the original equipment manufacturer.
`<<user.client.Win_ProcessorArch-itectureID>>`	Returns the processor architecture ID of the client device's processor. This value is dependent on the original equipment manufacturer.
`<<user.client.Win_Processor-Count>>`	Returns the number of processors on the client device.
`<<user.client.isDaylightSav-ings>>`	Returns the text true or false based on whether or not the client device is currently in daylight savings time.
`<<user.client.xmitConfigGroup>>`	Returns the defined group of the transmit configuration definition selected by the user for the current transmission.

Tag Name	Description
`<<user.client.xmitConfigName>>`	Returns the name of the transmit configuration definition selected by the user for the current transmission. This is the internal definition name.

<<user.info>> Data Tag Container

The `<<user.info>>` data tag container is a special data tag provided to allow for user-specific data to be captured at the beginning of the synchronization process and made available globally to all other synchronization processing. The members of this container tag are determined based on values returned from the back end system. The methodology for this depends on the type of system connection for the back end system.

Members within the `<<user.info>>` container are named when retrieved. Those values are then referenced using the syntax:

```
<<user.info.tagName>>
```

The members within this container tag are retrieved using either SQL queries run from the `SqlBE.ini` section `[UserInfo]`; or they are set via an HTTP-XML system connection's response mappings. Specifically, the response mappings within the validate user requests for this system connection type.

When creating these tags via a SQL Database system connection, the column in the return set of the query retrieving these values will be the name for the data tag. When creating these tags using the HTTP-XML system connection, a part of the response mapping definition is the attribute containing the tag's name.

For both system connection types, the tags available in the <<user.info>> container are set immediately following user validation and are available to all synchronization processing that takes place after this point.

When a tag is added to this container in one system connection it will be available to synchronization components for all other system connections.

<<server>> Data Tag Container

The <<server>> data tag container includes members that return values and information about the Agentry Server instance for the current transmission. Each of these tags will return values specific to the current Server instance for the current transmission.

Table 7. <<server>. Member Data Tags

Tag Name	Description
`<<server.ad-min.name>>`	Returns the value configured in the `agentry.ini` server configuration file section `[Server]`. The configuration option `administratorName` contains the value returned by this tag.
`<<server.ad-min.phone>>`	Returns the value configured in the `agentry.ini` server configuration file section `[Server]`. The configuration option `administratorPhone` contains the value returned by this tag.
`<<server.ad-min.email>>`	Returns the value configured in the `agentry.ini` server configuration file section `[Server]`. The configuration option `administratorEmail` contains the value returned by this tag.
`<<server.system-Name>>`	Returns the value configured in the agentry.ini server configuration file section [Server]. The configuration option `systemName` contains the value returned by this tag. If this configuration option is not set or is not present, the default return from this tag is Agentry Server.
`<<server.serial-Number>>`	Returns the serial number entered when the Agentry Server was installed. This value will be unique for all Server instances in a multi-server production implementation.

Data Tags for Application Globals

The values of any application global definition can be returned using SDML data tags. The syntax for a global data tag is as follows:

```
<<groupName.globalName [length=n]>>
```

The `groupName` is the defined group for the global definition. The global name is internal definition name of the global. These tags will return the current value of the global definition. If the global value has been overridden the override value will be returned. References to global data tags in synchronization components processed by the Agentry Server prior to the global override processing will return the global's defined value.

All global data tags are strings, regardless of the data type of the global definition. Global data tags accept a single named parameter specifying the length of the string to return from the global. When a length is specified the data tag will return no more characters than specified in the length parameter, counting from the left. Any characters beyond the specified length will be truncated from the returned value.

Query Constants Files and Data Tags

Installed with the Agentry Server are two query constants files provided for use with SQL Database system connections: `Oracle_sd.ini` and `SqlServer_sd.ini`. Each is intended for use with the database type for which they are named. The contents of these files include a single configuration section, [Database], within which are a set of configuration options listed as key and value pairs. Within each file exist the same keys. The values for these items are different in each file.

The purpose of these values is to provide support for applications which may synchronize with the same back end system, but which may be driven by different database types. These files support query reuse between these systems by providing expressions matching a given vendors variation in support of the ANSI SQL and database-type specific functions. The contents of these files are listed next, with the value for each key listed for both files:

Table 8. Query Constant Files Keys and Values

key	Oracle_sd.ini Value	SqlServer_sd.ini Value
`name`	Oracle	SqlServer
`getSystemTime`	sysdate	getdate()
`timeStampFormat`	to_date('%m/%d/%Y %H:%M:%S', 'mm/dd/yyyy HH24:MI:SS')	'%m/%d/%Y %H:%M:%S'
`dateFormat`	to_date('%m/%d/%Y', 'mm/dd/yyyy')	%m/%d/%Y'
`timeFormat`	to_date('%H:%M:%S', 'HH24:MI:SS')	'%H:%M:%S'
`tempdate`	to_date('01/02/1901 12:00:00', 'mm/dd/yyyy HH24:MI:SS')	convert(DATETIME, '01/02/1901 12:00:00')
`substring`	substr	substring
`stringcat`	||	+
`charFunction`	chr	char
`nullFunction`	nvl	isNull
`singleRow`	from dual	*null*
`unicodePrefix`	N	N

key	Oracle_sd.ini Value	SqlServer_sd.ini Value
`terminalErrorCodes`	00028;01001;01012;03113;03114; 12203;12500;12505;12535;12571	0;1;2;4;5;11;53
`retryWithChangeErrorCodes`	*null*	*null*
`retryWithoutChangeErrorCodes`	00060	00060
`fatalWithMessageErrorCodes`	*null*	*null*
`fatalWithoutMessageErrorCodes`	*null*	*null*

Each of these values is specific to a database type. The key for each value is available using SDML data tags using the syntax:

```
<<database.keyName>>
```

These data tags will return the value as specified in the query constant file in use for the SQL Database system connection. The file used by a system connection is set in the `[SQL-n]` section of the `agentry.ini` file by setting the configuration option `queryConstantFiles`.

This file is processed by the Server at startup and data tags are created and made available to all SQL scripts processed by the Agentry Server.

Other values may be added to this file within other sections. The syntax for referencing these values is:

```
<<sectionName.keyName>>
```

Following is a description of each of these data tags intended purpose:

Tag Name	Description
`<<database.name>>`	Returns the name of the database type for which the file was created. This value should not be altered in the source file

Tag Name	Description
<<database.getSystemTime>>	Returns the database-specific system date and time function.
<<database.timeStampFormat>>	Returns the database-specific date and time tokens used to format date and time values. This setting is used by the Agentry Server when expanding date and time data tags and should not be altered in the source file. This may be passed to the format parameter for date and time values within data tags.
<<database.dateFormat>>	Returns the database-specific date tokens used to format date and time values. This setting is used by the Agentry Server when expanding date data tags and should not be altered in the source file. This may be passed to the format parameter for date and time values within data tags.
<<database.timeFormat>>	Returns the database-specific time tokens used to format date and time values. This setting is used by the Agentry Server when expanding time data tags and should not be altered in the source file. This may be passed to the format parameter for date and time values within data tags.
<<database.tempdate>>	Returns the date and time of January 2, 1901, 12:00:01 am in the database-specific format for dates and times. This value can be used in synchronization when last update or other date and time values for data definitions contain invalid date and times. If a different date and time is necessary it can be altered in the query constants file.
<<database.substring>>	Returns the database-specific function for extracting a substring from a source string.
<<database.stringcat>>	Returns the database-specific character for concatenating string values.
<<database.charFunction>>	Returns the database-specific function for converting values to the character or VARCHAR data type.

Tag Name	Description
<<database.`nullFunction`>>	Returns the database-specific function used to test values for null and optionally replace those values with a default.
<<database.`singleRow`>>	Returns the required FROM portion of a query to select values from nothing.
<<database.`unicodePrefix`>>	Returns the prefix to append to values to in-dicate they are encoded in unicode.

Password Data Tags

The following two data tags are available only in SQL Scripts run from the `[ChangePassword]` section of the `SqlBE.ini` configuration file of the Agentry Server. They contain the old and new passwords for a user when that user is performing a password change from the Agentry Client.

Tag Name	Description
`newPassword`	Returns the new password entered by the user when changing the password on the Agentry Client.
`oldPassword`	Returns the previous password being change by the user when changing the password on the Agentry Client.

Complex Table Data Tags

The synchronization components for complex tables have specific data tags available providing information about the definition and its current data. These include whether or not the table is in a rebuild state, the last update date and time indicating when the table was last synchronized on the client, and the name of the table. These tags are in addition to those that are globally available.

Tag	Description
`<<name>>`	Returns the internal name of the complex table definition as entered in the application project.

Tag	Description
`<<rebuild>>`	Returns true or false, indicating whether or not the table is in a rebuild state. This is intended for use in synchronization to determine if the synchronization should retrieve only modifications to the table's data, or if all records for the table should be retrieved. Returns true when a change to the complex table definition has been published to the Agentry Server; if the synchronization logic includes the data tag <<user.agentryID>> and the value of that tag changes from the previous transmit; or if the tables force reload logic indicates the table should be in a rebuild state.
`<<lastUp-date>>`	Returns the date and time provided by the client when the complex table was last synchronized. By default this value is provided by the Agentry Server based on a query of the back end system. However, synchronization of the complex table should include retrieving this value with the table's data. The latest date and time retrieved during that process will be used as the complex table's last update value. This tag supports the use of the named parameter `format` to format the time and date using date and time tokens.

Data Table Data Tags

The synchronization components for data tables have specific data tags available providing information about the definition and its current data. These include the last update date and time indicating when the table was last synchronized on the client, and the name of the table. These tags are in addition to those that are globally available.

Tag	Description
`<<name>>`	Returns the internal name of the data table definition as entered in the application project.
`<<lastUp-date>>`	Returns the date and time provided by the client when the data table was last synchronized. By default this value is provided by the Agentry Server based on a query of the back end system. However, synchronization of the data table should include retrieving this value with the table's data. The latest date and time re-trieved during that process will be used as the data table's last update value. This tag supports the use of the named parameter `format` to format the time and date using date and time tokens.

Property Data Tags Overview

The data tags for property values within the application have unique, additional behaviors to the other data tags within Agentry. There are common items to all property data tags, including access to raw values and indicator of the property value in relation to its defined special value. There are also behaviors for data tags specific to the data type of the property the tag represents. Another aspect unique to property data tags is their scope. The property values available within the SDML will vary depending on which synchronization definition is referencing the tag.

The first item to be aware of with property data tags is that they are only available in step definitions. The specific properties in scope for a step will depend on which step usage definition is running the step during synchronization. No property data tags are available to any other synchronization component beyond the step definitions of a module.

Property Data Tag Parameters

Data tags for properties, regardless of data type, support two parameters specific to properties. These are the `.isSpecial` and `.raw` parameters. The syntax for these parameters is as follows:

```
<<object.propertyName.isSpecial>>
<<object.propertyName.raw>>
```

The `.isSpecial` parameter returns a true or false value indicating whether or not the current property value is equal to the property's defined special value. True indicates it is equal to the special value. False will be returned when the property value is anything other than the special value or if the property does not have a special value.

The `.raw` parameter is available to all property data tags regardless of data type, though its exact behavior will be data type-specific. The purpose of the `.raw` parameter is to return the value of the property without any formatting of the data. By default many of the property data tags will return the value in a formatted manner befitting the data type of the property. As an example, string properties are automatically dequoted during expansion. If the `.raw` parameter is used, the value will not be dequoted during expansion. Other data types have different behaviors related to the formatting and therefore the value returned by `.raw` will be different for each data type.

Data Tags for Fetch Client Exchange and Server Exchange Steps

Steps run by fetch definitions will have access to all properties defined for the fetch. Object key properties for the object instances in the collection targeted by the fetch may also be available depending on how the step usage definition's **Run** attribute setting.

The following lists describe the data tags in scope for each of the fetch step usage definition **Run** attribute settings.

Table 9. Run Attribute: Run One Time

Tag	Description
`<<`*`collectionName`*`>>`	This tag returns the collection targeted by the fetch. This tag may be passed to the <<fore-ach...>> function tag to iterate over the object instances within the collection. For each object instance, the tags available include the key property of the object type, and the last update (<<lastUpdate>>) value of each object.
Either: • `<<fetch.`*`propertyName`*`>>` • `<<`*`fetchName`*`.`*`propertyName`*`>>`	Any properties defined within the fetch definition are available using the syntax shown.
`<<fetch.messageNumber>>`	This data tag returns the fetch's message number as recorded in the `messages.log` file generated by the Agentry Server. This is typically used for debugging and similar purposes.

Table 10. Run Attribute: Run Once per Object

Tag	Description
`<<object.`*`keyPropertyName`*`>>`	This tag returns the key property of the object instance currently being processed by the step.
`<<lastUpdate>>`	This tag returns the last update value of the object instance currently being processed by the step.
`<<fetch.messageNumber>>`	This data tag returns the fetch's message number as recorded in the `messages.log` file generated by the Agentry Server. This is typically used for debugging and similar purposes.

Data Tags for Transaction Step Usage Definitions

All step usage definitions within transactions include the same data tags within their scope. Following is a list of these data tags:

Tag	Description
`<<timestamp>>`	Returns the date and time when the transaction was applied on the Agentry Client. This value is obtained from the client device.

Tag	Description
`<<transaction.`*`propertyName`*`>>`	Returns the value of the transaction property, *`propertyName`*. All properties within a transaction are available via data tags.
`<<objectName.`*`keyPropertyName`*`>>`	Returns the key property of the object targeted by the transaction. The object name must be used in this syntax, as the generic `object` designation is not valid in this context.
`<<transaction.messageNumber>>`	Returns the value of the transaction's message number as recorded in the `messages.log` file generated by the Agentry Server. Typically used for debugging and similar purposes.

Property Data Tags for Push Step Usage Definitions

Steps run as push retrieval and push removal steps are either once per poll period or once per user per poll period. For either run setting, these steps do not have access to object properties and therefore have no available property data tags.

For a given poll, push read steps can be run once, once per user, once per object, or once per collection object. When run once per object, the steps will be able to use property data tags to access the key property of any object for the target collection created by the push retrieval steps or previous push read steps. When run once per collection object, the data tag for that child object type's key property will be available to the step. The child objects in the collection must have been defined before the step that needs to reference the key property.

Push response and error steps are both always run once per object. Therefore these steps have access to the key property of the object for which they are run.

Property Data Tags for Service Event Step Usage Definitions

The step usage definitions for service events include read steps, data state steps, update steps, and error handling steps.

Service event read steps can be defined to run once or to run once per object. When defined to run once, the read step will have access to the collection created by the service event's synchronization components.

When defined to run once per object, the read steps the property data tags for the object type will be available. These will expand to the property value of the object instance currently be processed by the error handling step. For all of these values the syntax of the SDML property data tag is `<<object.propertyName>>`.

Data state steps and update steps within service events are always run once per object and will have access to all property values of the object instance being processed.

Error handling steps can be defined to run one time or run once per object. When defined to run one time, the error handling steps will have access to the object collection being synchronized by the service event. When defined to run once per object, the property data tags for the object type will be available. These will expand to the property value of the object instance currently be processed by the error handling step. For all of these values, the syntax of the SDML property data tag is `<<object.propertyName>>`.

Property Data Tags for Object Read Steps

Object read steps are run as a part of downstream synchronization that may occur for various synchronization processes, including fetch, push, service event, and transaction processing. Also, the read step itself may be defined to run one time or run once per object. Both of these aspects of an object read step can impact the property data tags available to the step.

When the object read step is run after a fetch, any fetch properties will be in scope for the object read step. These must be referenced as `<<fetch.propertyName>>`.

For all four situations, the read step will be run either in the context of a target collection, for a specific object instance, or for an instance of an object in a collection property of the object definition that contains the read step, based on the run attribute.

When defined to run one time the object read step will have access to the collection targeted by the fetch, push, or service event run immediately before the object read steps. If run one time the object read steps will have access to the key property and last update values for the object currently being processed. If defined to run once per collection object, the properties of the object collection property available to the read step include the key property of the child object, as well as the key property of the step's parent object. In this case, both objects must be referenced by their definition names, as in: `<<customer.customerID>>`, `<<order.orderID>>`.

Data Tags and Property Data Types

The data tags to access property values are different from other data tags. The basics of there use are the same as all data tags. However, data tags for properties include additional parameters to access the property values and those parameters depend upon the data type of the property they are referencing.

Boolean

A Boolean property will result in a Boolean data tag. Like their property counterparts, Boolean data tags are either true or false. When passed as an argument to a function tag, there are special syntactical rules that apply to Boolean data tags.

When data tag expansion occurs, a Boolean data tag will result in either the text "true" or "false." Because of this fact, when a Boolean data tag is used as a parameter to a function tag, it should not be enclosed in markers (<< and >>). When a Boolean is not enclosed in these markers, the value of either true or false is passed to the function, rather than the text values of "true" or "false". This is important since, if the text values are passed to a function that is expecting a Boolean value, it will always consider the value passed in to be true. Remember that true is a value and false is the absence of a value. The text “false” is a value and, thus, will be treated as true in the context of a Boolean parameter.

When a Boolean data tag is passed as an argument to a function, it is likely that the Boolean value of true or false is desired, not the text. In this case, you omit the markers around the tag. This will result in the Boolean value of the tag being passed as an argument to the function. So, in the following examples:

```
<<if <<object.BooleanProp>> ... >>
```

```

```

```
<<if object.BooleanProp ... >>
```

The first will result in the Boolean data tag being expanded to result in:

```
<<if “false” ... >>
```

This will result in the text value of “false” being passed to the function.

The second line will result in the Boolean value of true or false being passed to the `<<if...>>`, rather then the text “true” or “false.” Note that referencing a Boolean without the markers is only valid when the Boolean tag is passed as an argument to a function.

Strings

There are four property data types that will result in a string data tag. These property data types include:

- String
- Complex Table Selection
- Data Table Selection
- External Data (provides access to the file name and location, not the file data)

The value of the item the data tag provides access to will be placed in the script at expansion time. There is, however, a minor modification to the value that will occur for scripts used in SQL system connections. Any single quotes within the string will be escaped for the database, that is, a second single quote will be placed before the existing quote. This is the standard escape character in most database systems and is necessary as the single quote is used to denote the beginning and end of a string in a SQL statement. So, when expanded, if a string data tag contains the value:

```
The customer’s car has front end damage.
```

the value in the script when data tag expansion occurs will be:

```
The customer’’s car has front end damage.
```

Note the two single quotes in place of the previously single quote (used as an apostrophe here) within the word “customer’s”. As explained in the chapter on function tags, the `<<dequote...>>` function also provides this ability. However, for string data tags with a property as its data source, this behavior is automatic.

Another optional behavior in string property data tags is the ability to truncate the value, if needed. This is accomplished through the optional named parameter, length=. The syntax for this is as follows:

```
<<parent.stringDataTag length=n>>
```

Denoting this data tag in this manner, the value of the string will be truncated to the length of *n*. This truncation occurs before any quotes are escaped, so that the extra quotes added in that process are not affected by the truncation. As stated, length is an optional parameter and, if not provided, the entire value of the string will be placed in the script file during data tag expansion.

Another optional parameter to a property string data tag is raw. This parameter will return the value of the string without escaping the quotes it may contain. The syntax for this is:

```
<<object.stringDataTag.raw>>
```

Integral and Decimal Numbers

Integral data tags result from properties of the types Integral Number and Identifier. Decimal data tags result from properties of type Decimal Number. During data tag expansion, the value of these tags are placed in the script without modification. In this respect these two data types are treated the same. It is when these values are passed as arguments to functions where the difference between the two types becomes important.

Integral Number data tags will contain whole number values. These data tags can be used with the math function tags that accept integral numbers. Many of the function tags that can accept the use of numerical values have a type parameter. When using this type of data tag, the value to the type parameter of the function is `Int`.

Decimal Number data tags will contain numerical values that have a fractional portion, such as 2.4 or 3.00. These data tags can be used with math function tags that accept decimal numbers. Many of the function tags that accept the use of numerical data, math tags as well as others, can accept a type parameter. When using this type of data tag, the value for the type parameter is `Float`.

Date

Date data tags contain a calendar date value. During data tag expansion of a SQL script, date tags are expanded in such a way that the resulting text is enclosed in the conversion function of the target database system that converts string values to date values. So, if a date data tag, `StatusDate`, contains the value 01/25/2006, then the data tag

```
<<transaction.StatusDate>>
```

in an Oracle database will be expanded to the value

```
to_date('01/25/2006', 'MM/DD/YYYY')
```

It is possible to get just the date value as a string by using the raw parameter, which is available to all date data tags. Continuing with the previous example, the data tag

```
<<transaction.StatusDate.raw>>
```

will be expanded to the value

```
01/25/2006
```

This can be useful if the date value is to be within some sort of string value within the database, such as a description. In this case, you do not want to convert the value to a database date format, but rather use it as a string.

Time

Time data tags contain a time of day value, in a 24 hour format. When data tag expansion occurs in a SQL script, the resulting value is enclosed in the conversion function for the target database system that is used to convert string values into times. So, if a data tag named `EndTime` contains the value 13:10:43, then the data tag

```
<<transaction.EndTime>>
```

in an Oracle database will be expanded to

```
to_date('13:10:43', 'HH24:MI:SS')
```

It is possible to access the value as a string, without the conversion function, by using the parameter raw, which is available to all Time data tags. Using the previous example data tag

```
<<transaction.EndTime.raw>>
```

will expand to the value

```
13:10:43
```

This is used whenever you wish to access just the time string, without converting it to the database time format.

Date And Time

Date and Time data tags are, in essence, a combination of the Time data type and the Date data type. This data tag type contains the calendar date and time of day in a single value. When data tag expansion occurs in a SQL script, the resulting value is enclosed in the conversion function for the target database system that is used to convert string values into dates and times. If a data tag named `InspectionDateTime` contains the value 02/13/2005 13:20:35, then the data tag:

```
<<transaction.InspectionDateTime>>
```

in an Oracle database will be expanded to

```
to_date('02/13/2005 13:20:35', 'MM/DD/YYYY HH24:MI:SS')
```

As with other data types, it is possible to access the string value without wrapping it in a conversion function, by using the parameter raw, which is available in all Date and Time data tags. Using the previous example data tag

```
<<transaction.InspectionDateTime.raw>>
```

will expand to

```
02/13/2005 13:20:35
```

This is used whenever just the date and time value is desired, without wishing to convert it before being processed by the enterprise system.

Formatting Dates and Times

These three data types that deal with dates and times support the use of the named parameter format=. This parameter accepts one or more of several date and time tokens. These tokens are combined to provide a picture of how the data should be placed in the file. When a date, time, or date and time data tag contains the format parameter, the default format is overridden, including the conversion function within which the values are normally contained.

Following is a list of the tokens supported by these data tag types. In each of the examples the date and time is 02/07/2001 10:09:03 AM. The **Example** column contains the value for the token listed. The **Short Form** contains the example of the value that results by preceding the token with a hyphen, as in: `%-m`

Token	Description	Example	Short Form
%a	The three letter abbreviation of the day.	Wed	We
%A	The name of the day.	Wednesday	Wed
%b	The three-letter abbreviation of the month.	Feb	n/a
%B	The name of the month.	February	Feb
%d	The date of the month.	07	7
%j	The Julian date, with a leading 0.	038	38
%m	The two digit month (01-12)	02	2

Token	Description	Example	Short Form
%w	The numerical day of the week (0-6 Sunday = 0)	3	n/a
%y	The two-digit year	01	1
%Y	The four-digit year.	2001	n/a
%R or %r	The raw format of the value	36543,37	n/a
%H	The hour of the day, 24 hour format.	10	n/a
%h or %I	The hour of the day, 12 hour format	10	n/a
%M	The minutes of the hour.	09	9
%p	AM or PM indicator.	A	a
%S	Seconds of the minute.	03	3
%Z	The time zone when the time was recorded.	Central Standard Time	n/a
non-token characters	Any non-format token character, or any character not preceded by the % sign passed to the named parameter format will be returned unchanged at the position at which it was placed in the parameter value.	n/a	n/a

Signature

Signature data tags result from Signature property types. This data tag type is used with the signature capture functionality available in Agentry. This functionality allows for an application to capture and store a signature the user enters on the screen. The image is stored as a bitmap, and is also available in the raw pixels.

This data tag type supports the following parameters. The value returned for all of these parameters is a string, with the exception of `bmp`, `row.`*n*, and `raw`.

- `type` - Will return either “image” or “none” during expansion. Image indicates that there is an image and the transaction was performed on a client device that supports this functionality. None indicates that the device does not support the signature capture functionality.
- `bmp` - This parameter returns the signature, if it exists, in a bitmap format. This returns a string of hexadecimal values that may be used as an argument to another utility program that processes the data, e.g. stores it in a database.
- `height` - This parameter returns the height, in pixels, of the signature image.
- `width` - This parameter returns the width, in pixels, of the signature image.

- `row.`*`n`* - This parameter returns the row of pixels, specified by n
- `signed` - Returns either true or false. True is returned when a signature has been captured on the client, or, for devices that do not support this functionality, if the check box control that replaces it has been checked. These values are returned as text values of “true” or “false.”
- `raw` - Returns the pixels that make up the image.

The syntax for these parameters is:

```
<<transaction.signatureProp.parameter>>
```

Of these parameters, only `type` and `signed` are always available. The others will return a data tag not found error if a signature was not captured on the client, i.e. `signed` returns false. Therefore, the return value of `signed` should be checked before attempting to access the other parameters.

Image

Image data tags result from Image property types. This data tag type is used with the image capture functionality available in Agentry. This functionality allows the application to interact with a device’s built in still camera, when present. A captured image is stored on the device as a file and referenced by the image property.

During synchronization this file data is sent to the server for processing as a part of the transaction data. To access this image data there are two options. The first is to use a file document management step. In this case, it is likely not necessary to reference the data tags for the image property, though they can be when necessary. For other step types access to the image data requires the use of SDML data tags. Data tags for the image property include two parameters in the format `<<transaction.imageProperty.`*`parameter`*`>>`.

The following list describes these parameters and their purpose:

- `data` - This parameter returns the image data in ASCII-encoded hexadecimal values.
- `type` - This parameter returns the image type as stored on the client device. The possible return values of this parameter are `jpeg`, `bitmap`, and `unknown` when the file type is not determined.

<<agent>> Data Tag Container

The `<<agent>>` data tag container includes only a single member, `.version`. The data tag `<<agent.version>>` returns the full version of the Agentry Server.

This is the only member of this data tag member in the `<<agentry>>` data tag container. Additional members may be added in a future release.

SDML Function Tags Overview

SDML Function tags provide value processing and evaluation to the SDML. Function tags are represented in print with the syntax `<<funcName...>>`. Within the SDML there are numerous functions that provide processing for logic operations, string operations, and mathematical operations.

Function tags within the SDML will often return a value. That return value is placed at the point where the function tag exists within the script file in which it is contained. The use of function tags can provide a significant source of operational power, allowing for different sets of logic to be processed by a script at runtime, depending on conditions.

The following sections list each of the function tags available, including descriptions of their purpose and behavior, as well as usage syntax and similar information.

<<if>>

The `<<if...>>` function provides the if-then-else logic to the SDML. This is the most common decision making mechanism within any language. The `<<if>>` function receives a single argument, which it evaluates as being either true or false. If the argument is true, its first expression, `trueExpression`, is returned. Otherwise, the `falseExpression` value is returned. This expression must be preceded by the keyword `else`. `falseExpression` is optional and if it is not present, the `else` keyword cannot appear either.

This function can be used to return something as basic as a single word, or as complex as an entire SQL statement. The contents of either expression can contain SDML text as well. In the case where there is no `else` portion, and the `boolArg` is false, the return value of the `<<if...>>` function is an empty string.

Arguments

<<if boolArg “trueExpression” [else “falseExpression”]>>	
boolArg	The value to be evaluated as either true or false. May be a Boolean data tag or the Boolean return value of function call. If this argument is a Boolean data tag, the tag should be entered by name, excluding the tag markers (<< and >>) to return the Boolean value of that property, rather than the text value.

Expressions

- **trueExpression –** Required expression containing the value to be returned by the function when `boolArg` is true. This expression must be enclosed in double quotes.
- **falseExpression –** Optional expression containing the value to be returned by the function when `boolArg` is false. This expression must be preceded by the keyword else and the expression itself must be enclosed in double quotes.

Parameters

- N/A

<<case>>

The `<<case...>>` function provides the `switch-case` logic to the SDML. It takes a required switch argument and at least one case-expression argument pair. It may take as many additional case-expression pairs as are needed, plus an optional default argument.

During expansion, this function tag compares the value of the switch argument to each provided case argument in turn. It will return the expression argument for the first case argument to which the switch argument matches.

As an optional argument, a default value may be provided that will be returned by the function when the switch argument does not match any provided case argument. The syntax for the default return value is `default=returnValue`, where default is a keyword. For this reason, neither the value of the switch argument, nor any of the case arguments may be the value default.

Arguments

<<case switch case1=exprssion1 [caseN=expressionN] [default=defaultExpression]>>	
switch	The value the function will switch on, comparing to the value of each `case` argument until a match is found.
case1	The first case argument to the function. This is a required argument and must be immediately followed by an equal sign (=) with no whitespace between the case argument and the sign. The expression1 argument immediately follows the equal sign, also with no whitespace allowed.
expression1	The first `expression` argument to the function. This is a required argument and contains the value the function will return when switch matches case1. case1 and expression1 are separated by an equal sign with no whitespace allowed between then, as in: `case1=expression1`. Any SDML text in `expression1` will be expanded after it has been returned by the function.

<<case switch case1=exprssion1 [caseN=expressionN] [default=defaultExpression]>>	
caseN	Additional optional `case` arguments to the function. If `case1` does not match `switch`, the function will compare each subsequent case argument in order until a match is found. Each `caseN` argument must be followed by and equal sign and then a corresponding `expressionN` value. No whitespace can exist between each case-expression pair.
expressionN	Additional optional `expression` arguments to the function. Each case argument must be followed by a corresponding expressionN value. Each case-expression argument pair must be separated by an equal sign with no white space allowed between then, as in: `caseN=expressionN`. Any SDML text in `expressionN` will be expanded after it has been returned by the function.
default=defaultExpression	This optional argument specifies the expression returned by the function should the `switch` not match any of the `case` values. The syntax for this argument requires the text `default=` followed by the default expression the function should return.

Parameters
None

<<skip>>

The `<<skip...>>` function will force the Agentry Server to skip the step definition in which the function call is contained. This function takes an optional comment argument, the contents of which will be the log message generated by the Server for the log file of the step type's system connection. This function is only valid when called within the script component of a module-level step definition.

This function can be used during testing to skip over a script that you do not wish to run, or in certain production situations where you may not wish a script to run under certain conditions. The primary intent of this function is the result of the requirements of the contents of a SQL step's script. This script cannot be empty, nor can it contain only SDML logic with no valid SQL statement to be processed. Depending on conditional processing, such as queries returned by the `<<if...>>` function, it is possible for a script to return a valid SQL statement in one condition, but not in another. In this situation, the `<<skip>>` function should be the expression returned when no SQL statement should be run. Note that this function is not limited to SQL step definitions, though this is its primary intended use.

Arguments

<<skip ["comment"]>>	
comment	This is an optional argument. It contains any text value that will be used as a log message generated by the Server for the log file of the step type's system connection.

Parameters
None

<<stop>>

The `<<stop...>>` function will stop the Agentry Server's interactions with the the back end system. Any subsequent steps within the same group of the same parent definition will not be processed. The function takes an optional comment argument, the contents of which will be written as a log message by the server to the log file for the parent step's back end log category.

As an example of the function's behavior, if the second of four client exchange steps within a fetch contains a `<<stop>>` function, that step and those that come after it will not be run. This function will result in a commit being performed, committing any changes made previously by the processing of the previous steps within the same parent.

Arguments

<<stop ["comment"]>>	
comment	This optional argument contains text which will be written as a log message by the server to the log file for the parent step's back end log category.

Parameters
None

<<abort>>

The `<<abort...>>` function within a step will result in that step's processing being halted. Any subsequent steps within the same parent definition will also not be processed. Any changes made by the previous steps in the group will be rolled back. This function takes an optional comment argument, the contents of which will be written as a log message by the Agentry Server to the log category of the step's system connection.

As an example of this function's behavior, if the third of five steps in a fetch's server exchange steps is aborted, steps four and five will not be run either. Changes made by the first two steps will be rolled back.

Note that the `<<abort>>` and `<<rollback>>` functions perform the exact same behavior.

Arguments

<<abort ["comment"]>>	
comment	This optional argument contains text which will be written as a log message by the Agentry Server to the log category of the step's system connection.

Parameters
None.

<<rollback>>

The `<<rollback...>>` function within a step will result in that step's processing being halted. Any subsequent steps within the same parent definition will also not be processed. Any changes made by the previous steps in the group will be rolled back. This function takes an optional comment argument, the contents of which will be written as a log message by the Agentry Server to the log category of the step's system connection.

As an example of this function's behavior, if the third of five steps in a fetch's server exchange steps is rolled back, steps four and five will not be run either. Changes made by the first two steps will be rolled back.

Note that the `<<abort>>` and `<<rollback>>` functions perform the exact same behavior.

Arguments

<<rollback ["comment"]>>	
comment	This optional argument contains text which will be written as a log message by the Agentry Server to the log category of the step's system connection.

Parameters
None.

<<and>>

Description

The `<<and...>>` function performs a logical conjunction of two or more Boolean values. If all arguments to the function are true, the function will return true. Otherwise, the `<<and...>>` function will return false.

This function is almost always used as an argument to another function, normally the `<<if...>>` function. The reason for this is that the value returned is a Boolean value within the SDML and will simply return the text values of "true" or "false," if not an argument to another function.

Arguments

<<and boolArg1 boolArg2 [boolArg3...boolArg	
boolArg1-N	The boolean values checked for true or false by the function. May be either a Boolean data tag, or a function that returns a Boolean value. The function must have at least two arguments, and up to as many as needed. Each is checked in the order listed, until a false value is found.

Parameters

None

<<or>>

Description

The `<<or...>>` function performs the logical disjunction of two or more Boolean values. Each argument is compared in the order listed until a true value is found, at which point the function returns true. If all arguments contain a false value, then the function will return false.

This function is almost always used as an argument to another function, normally the `<<if...>>` function. The reason for this is that the value returned is a Boolean value within the SDML and will simply return the text value of either "true" or "false" if not passed as an argument to a function.

Arguments

<<or boolArg1 boolArg2 [boolArg3...boolArgN]>>	
boolArg1-N	The boolean values checked for true or false by the function. May be either a Boolean data tag, or a function that returns a Boolean value. The function must have at least two arguments, and up to as many as needed. Each is checked in the order listed, until a true value is found.

Parameters
None

<<not>>

The `<<not...>>` function inverts the Boolean value of `boolArg` and returns this inverted value. If `boolArg` is true, the function will return false, and vice versa.

This function is almost always used as an argument to another function, normally the `<<if...>>` function. The reason for this is that the value returned is a Boolean value within the SDML and will simply return the text value of either “true” or “false” if not passed to another function.

Syntax
<<not boolArg>>

Return Value
Boolean

Arguments

<<not boolArg>>	
boolArg	A boolean value that is inverted by the function. May be a Boolean data tag or a function that returns a Boolean value.

Parameters
None

<<eq>>

The `<<eq...>>` function compares `arg1` and `arg2`, based on the value of the “type” parameter if present, and returns true if the two values are found to be equal. Otherwise, this function returns false. If the type parameter is not specified, the default comparison is String.

When a type is specified any values not of that type are converted prior to the comparison. So, if an integral data tag contains a value of 10 and a decimal number contains a value of 10.1, and the type is `Int`, these two values would be considered equal, as the decimal value would be converted to an integer, and the decimal portion is truncated.

Arguments

<<eq arg1 arg2 [type=Int|Float|String]>>	
arg1	The first value of the two compared by the function. May be a hard coded value, data tag, or a function.
arg2	The second value of the two compared by the function. May be a hard coded value, data tag, or a function.

Parameters

- `type` - This optional named parameter specifies how the function will compare the values of the two arguments. The acceptable values for this parameter are
  - **Int** - The arguments are Integral Numbers.
  - **Float** - The arguments are Decimal Numbers (short for "floating point number")
  - **String** - The arguments are String values.

<<ne>>

Description
The <<ne...>> function compares arg1 and arg2, based on the value of the "type" parameter if present, and returns false if the two values are found to be equal. Otherwise, this function returns true. If the type parameter is not specified, the default comparison is as Strings.

When a type is specified any values not of that type are converted prior to the comparison. So, if an integral data tag contains a value of 10 and a decimal number contains a value of 10.1, and the type is Int, these two values would be considered equal, as the decimal value would be converted to an integer, and the decimal portion is truncated.

Syntax
<<ne arg1 arg2 [type=Int|Float|String]>>

Arguments

<<ne arg1 arg2 [type=Int|Float|String]>>	
arg1	The first value of the two compared by the function. May be a hard coded value, data tag, or a function.
arg2	The second value of the two compared by the function. May be a hard coded value, data tag, or a function.

Parameters

- `type` - This optional named parameter specifies how the function will compare the values of the two arguments. The acceptable values for this parameter are
  - **Int** - The arguments are to be compared as Integral Numbers.
  - **Float** - The arguments are to be compared as Decimal Numbers (short for “floating point number”)
  - **String** - The arguments are to be compared as Strings

<<gt>>

The `<<gt...>>` function compares `arg1` and `arg2`, based on the value of the “type” parameter if present. It returns true if `arg1` is greater than `arg2`. If `arg1` is less than or equal to `arg2`, it returns false. If the type parameter is not specified, the default comparison is as Strings. When comparing values of different data types, it is strongly recommended that you do specify the type.

When a type is specified any values not of that type are converted prior to the comparison. So, if an integral data tag contains a value of 10 and a decimal number contains a value of 10.1, and the type is `Int`, these two values would be considered equal, as the decimal value would be converted to an integer, and the decimal portion is truncated. String comparisons are made based on alphabetical order.

Arguments

<<gt arg1 arg2 [type=Int|Float|String]>>	
arg1	The first value of the two compared by the function. May be a hard coded value, data tag, or a function.

<<gt arg1 arg2 [type=Int|Float|String]>>	
arg2	The second value of the two compared by the function. May be a hard coded value, data tag, or a function.

Parameters

- `type` - This optional named parameter specifies how the function will compare the values of the two arguments. The acceptable values for this parameter are
  - **Int** - The arguments are to be compared as Integral Numbers.
  - **Float** - The arguments are to be compared as Decimal Numbers (short for “floating point number”)
  - **String** - The arguments are to be compared as Strings

<<lt>>

Description

The `<<lt...>>` function compares `arg1` and `arg2`, based on the value of the “type” parameter if present. It returns true if `arg1` is less than `arg2`. If `arg1` is greater than or equal to `arg2`, it returns false. If the `type` parameter is not specified, the default comparison is as Strings.

When a `type` is specified any values not of that type are converted prior to the comparison. So, if an integral data tag contains a value of 10 and a decimal number contains a value of 10.1, and the type is Int, these two values would be considered equal, as the decimal value would be converted to an integer, and the decimal portion is truncated. String comparisons are made based on alphabetical order.

Arguments

<<lt arg1 arg2 [type=Int|Float|String]>>	
arg1	The first value of the two compared by the function. May be a hard coded value, data tag, or a function.
arg2	The second value of the two compared by the function. May be a hard coded value, data tag, or a function.

Parameters

- `type` - This optional named parameter specifies how the function will compare the values of the two arguments. The acceptable values for this parameter are

- **Int** - The arguments are to be compared as Integral Numbers.
- **Float** - The arguments are to be compared as Decimal Numbers (short for "floating point number")
- **String** - The arguments are to be compared as Strings

Expressions
None

<<ge>>

The `<<ge...>>` function compares `arg1` and `arg2`, based on the value of the `type` parameter if present. It returns true if `arg1` is greater than or equal to `arg2`. If `arg1` is less than `arg2`, it returns false. If the type parameter is not specified, the default comparison is as Strings.

When a `type` is specified any values not of that type are converted prior to the comparison. So, if an integral data tag contains a value of 10 and a decimal number contains a value of 10.1, and the type is Int, these two values would be considered equal, as the decimal value would be converted to an integer, and the decimal portion is truncated. String comparisons are made based on alphabetical order.

Arguments

<<ge arg1 arg2 [type=Int|Float|String]>>	
arg1	- The first value of the two compared by the function. May be a hard coded value, data tag, or a function.
arg2	- The second value of the two compared by the function. May be a hard coded value, data tag, or a function.

Parameters

- `type` - This optional named parameter specifies how the function will compare the values of the two arguments. The acceptable values for this parameter are
  - **Int** - The arguments are to be compared as Integral Numbers.
  - **Float** - The arguments are to be compared as Decimal Numbers (short for "floating point number")
  - **String** - The arguments are to be compared as Strings

<<le>>

Description
The `<<le...>>` function compares `arg1` and `arg2`, based on the value of the `type` parameter if present. It returns true if `arg1` is less than or equal to `arg2`. If `arg1` is greater

than `arg2`, it returns false. If the `type` parameter is not specified, the default comparison is as Strings.

When a `type` is specified any values not of that type are converted prior to the comparison. So, if an integral data tag contains a value of 10 and a decimal number contains a value of 10.1, and the type is `Int`, these two values would be considered equal, as the decimal value would be converted to an integer, and the decimal portion is truncated. String comparisons are made based on alphabetical order.

Arguments

<<le arg1 arg2 [type=Int|Float|String]>>	
arg1	The first value of the two compared by the function. May be a hard coded value, data tag, or a function. The data type of the value must be Integral Number, Decimal Number, or String.
arg2	The second value of the two compared by the function. May be a hard coded value, data tag, or a function. The data type of the value must be Integral Number, Decimal Number, or String.

Parameters

- `type` - This optional named parameter specifies how the function will compare the values of the two arguments. The acceptable values for this parameter are
  - **Int** - The arguments are to be compared as Integral Numbers.
  - **Float** - The arguments are to be compared as Decimal Numbers (short for “floating point number”)
  - **String** - The arguments are to be compared as Strings

<<empty>>

The `<<empty...>>` function allows you to check whether or not an object collection has any objects or if it is empty. If the collection contains no objects, this function will return true. If the collection contains at least one object this function will return false. This function is only valid when an object collection property is in scope for the step definition in which it is used.

Arguments

<<empty objectCollectionArg>>	
objectCollectionArg	This argument is a data tag representing an object collection. This collection must be defined. It is evaluated by the function for members.

Parameters
None

<<notEmpty>>

The <<notEmpty...>> function allows you to check whether or not an object collection has any objects or if it is empty. If the collection contains no objects, this function will return false. If the collection contains at least one object this function will return true. This function is only valid when an object collection property is in scope for the step definition in which it is used.

Arguments

<<notEmpty objectCollectionArg>>	
objectCollectionArg	This argument is a data tag representing an object collection. This collection must be defined. It is evaluated by the function for members.

Parameters
None

<<size>>

The <<size...>> function returns the number ofobject instances in an object collection. This value is always 0 or higher.

Arguments

<<size objectCollectionArg>>	
objectCollectionArg	This is the object collection whose members are to be counted.

Parameters
None

<<exists>>

The `<<exists...>>` function determines whether or not the specified object collection exists. If the collection specified by the `objectCollectionArg` argument exists, this function will return true. Otherwise it returns false.

A common use for this function outside of development testing is to verify the data tag returned by a call to the function `<<sql...>>` exists and contains data.

This function may also useful during testing or debugging of an application. It can be useful in applications that are deployed with various configurations from one installation to the next. The usage of this function can be helpful in determining if a specific configuration contains all of the definitions necessary.

Arguments

<<exists objectCollectionArg>>	
objectCollectionArg	This argument is the data tag representing the object collection that whose existence is to be checked. This argument must be a data tag that contains the name of an object collection whose existence is to be confirmed.

Parameters

None

<<foreach>>

The `<<foreach...>>` function allows for iteration over an object collection property. This function takes as its single argument the name of an object collection. The `expression` can contain text and SDML that will be returned once for each member of the named collection. If the collection is empty, nothing is returned.

This function allows you to iterate over a collection of objects. The expression specified will be returned once for each member and normally contains SQL and SDML that is to be processed for a single object. The `<<foreach...>>` function is commonly used for INSERT statements, which can only insert a single record at a time. It is often seen in SQL steps used to update a client exchange table during fetch processing.

The `expression` to the function can also contain an additional data tag, `<<my>>`. This tag represents the current member of the collection being processed by the function. To use the `<<my>>` data tag, the same syntax is used as in other areas where property data tags are used.

There is also the optional `<<key>>` tag available within a `<<foreach>>` expression, which provides the name of the current key being iterated over. This is normally only used in conjunction with SQL Flunkies, explained later in this chapter.

Arguments

<<foreach objectCollectionArg expression>>	
objectCollectionArg	The name of a collection that the function is to iterate over.
expression	The text to be returned once for each member of the collection. The expression can contain plain text and SDML. It will be submitted for expansion once for each collection member.

Parameters
None

<<upper>>

The `<<upper...>>` function converts a given string to all uppercase characters. The value returned is the string passed in with all characters converted to upper case. Any non-alphabetical characters, such as numbers, symbols, or punctuation (%, $, etc.) are returned unchanged.

Arguments

<<upper stringArg>>	
stringArg	This argument is a string value to be converted by the function. It can be either a hard coded value, a data tag, or a string return value from a function.

Parameters
None

<<lower>>

The `<<lower...>>` function converts a given string to all lower case characters. The value returned is the string passed in with all characters converted to lower case. Any non-alphabetical characters, such as numbers, symbols, or punctuation (“%”, “$”, etc.) are returned unchanged.

Arguments

<<lower stringArg>>	
stringArg	This argument is a string value to be converted by the function. May be a hard coded value, a data tag, or a string return value from a function.

Parameters
None

<<length>>

This function returns the length of a string value. All printable characters within the string are counted. This includes white space characters, where tabs are counted as a single character, and symbols, such as $ or %. Non-printable characters are also counted, such as newline and carriage returns. Remember in Windows systems that the end of a line in a multi-line string value contains two command characters, \n and \r, which will be counted by the <<length...>> function as one character each.

Arguments

<<length stringArg>>	
stringArg	The string value evaluated by the function. May be a hard coded value, a string data tag, or the return value of another function.

Parameters
None

<<join>>

Description
The `<<join...>>` function concatenates two or more string values together, with each separated by the value of the optional named parameter, join. If the join parameter is not specified, the values are concatenated together without any character separating them.

Arguments

<<join stringArg1 stringArg2 [stringArg3...stringArgN] join=joinString>>	
stringArg1/stringArg2	The required arguments to the function, which are the strings that will be joined. May be a hard coded value, an string data tag, or the return value of a function.
stringArg3-N	The optional additional strings to be joined. May be a hard coded value, string data tag, or the return value of another function.

Parameters

- **joinString –** The value of this optional named parameter contains the character or string used to join the arguments together.

<<dequote>>

The `<<dequote...>>` function, by default, removes any double-quote characters from a given string. It contains three optional named parameters, however, that significantly alter and enhance this functionality. First, by including the quote parameter, you can specify a different character to be removed form the given string or strings.

Second, the replace parameter can specify the character you wish to replace the quote character with. Finally, the join parameter allows you to specify the character used to join multiple string arguments to the function together.

Arguments

<<dequote stringArg1 [stringArgN] [quote=quoteChar] [replace=repChar] [join=joinString]>>	
stringArg1	This argument contains the string to be “dequoted.” May be a hard coded value, a data tag, or the return value of another function call.
stringArgN	Additional, optional argument(s) to be dequoted, and joined together with the previous arguments.

Parameters

- **quote –** The value to this named parameter is the character to be removed from the given string or strings. If not provided, the default value is double quotes. If the value to this parameter contains more than one character, the first will be used and the rest ignored.
- **replace –** The value to this named parameter is the single character to replace the “quote” character with in the string. If the value to this parameter contains more than one character, the first will be used and the rest ignored.
- **join –** The value to the named parameter is the character or string used to join together the arguments to the function. If not provided, each string is separated by a single white space.

<<trunc>>

The `<<trunc...>>` function will truncate the given stringArg to the number of characters of the value given to the required length parameter. By default, the characters are counted from the left most position up to and including the character at the position specified by the length parameter. This includes any white space characters, and also includes the end of line characters of line feed an carriage return, which each count as one.

If the from parameter is given and its value is “left”, the counting begins at the right-most character of the string, truncating the left characters beyond the given length.

If the stringArg is shorter or equal to the length specified, the entire string is returned.

Arguments

<<trunc stringArg length=lengthParam [from=left|right]>>	
stringArg	This argument contains the string to be truncated by the function. May be a hard coded value, a data tag, or another function.

Parameters

- **length –** This required named parameter specifies the length to which the stringArg value should be truncated.
- **from –** This optional named parameter specifies the portion of the stringArg value to be truncated. The values are left or right; any other value will be ignored and the default value of right will be used.

<<wordTrunc>>

The `<<wordTrunc...>>` function is similar to the `<<trunc...>>` function in that it will truncate a string to a given length. The length parameter specifies the maximum length of the string and must be provided to the function. The difference between this function and

`<<trunc...>>` is that `<<wordTrunc...>>` will always end its truncation on a white space. That is, the truncation of the string will be at the end of a whole word.

The string returned by the function will be, at most, the size of the length specified. However, if the specified length ends in the middle of the word, the last white space character before this point will be where the string returned ends.

The start parameter specifies the starting point of the function. If this parameter is given, the function will count from the beginning of the string up to the start character. Then, the function will count from this point up to the length value of characters.

If the string contains no white space, then length number of characters will be returned. If the string is shorter or equal to the length, the entire string will be returned.

Syntax

<<wordTrunc stringArg length=lengthParam [start=startParam]>>

Arguments

- **stringArg –** The string value to be truncated by the function. May be a hard coded value, a string data tag, or the return value of another function, provided it returns a string.

Parameters

- **length –** This required named parameter specifies the maximum length of the string returned by the function.
- **start –** This optional named parameter specifies the starting position from which the function will begin counting. Any characters before this position will be truncated, as will any characters beyond the value of the length parameter. If this parameter is not present, the starting position will always be the first character of the string, as specified by the from parameter.

<<cgi>>

The `<<cgi...>>` function can operate with either a single unnamed argument, or with many arguments in name-and-value pairs. If a single stringArg is given, it expands to a string that has all characters converted to CGI scoped values. If one or more name and value pair arguments are given, they are formatted to a string that is the named pairs, joined by ampersands (&), with the values CGI escaped.

The CGI function escapes strings following the CGI conventions certain characters are replaced with a % followed by two hexadecimal digits that are the ASCII value for the character.

The stringArg will have any characters it contains escaped according to CGI conventions. The name-and-value pairs will be formatted into named parameters and values, with the values also escaped according to CGI standards. The order of the named parameters is not preserved when this function is expanded. Also, the CGI function escapes spaces with %20's rather than with +'s. Both are allowed by the CGI convention.

Arguments

<<cgi stringArg>> --OR-- <<cgi named1=value [named2=value...namedN=value]>>	
stringArg	A text string that will be escaped according to CGI conventions. This may be a hard coded value, a data tag, or the return value of a function.
named1-n	A named parameter to be returned with a value that will be formatted according to CGI conventions. May be a hard coded value, a data tag, or the return value of a function.
value	A value to the corresponding named parameters that will be formatted according to CGI conventions. May be a hard coded value, a data tag, or the return value of a function.

Parameters
None

<<sum>>

The <<sum...>> function provides the operation of the plus sign (+) operator in other languages. This function will sum the arguments and return the result. There must be at least two arguments provided to the function, and there can be as many more arguments as needed.

The data type of these values can be Strings, provided the string contains only numbers, sign, and a single decimal. The values of string data types will be converted before being passed to the function.

Arguments

<<sum numArg1 numArg2 [numArg3...numArgN]>>	
numArg1-2	The required arguments, numerical, that are summed together. Maybe a hard coded value, data tag, or the return value of another function. If the value is hard coded and contains a negative or postivie sign, the entire value must be enclosed in double quotes, as in “-12.34”.

<<sum numArg1 numArg2 [numArg3...numArgN]>>	
numArg3-N	The optional numeric aguments to be summed together with all other arguments. May be a hard coded value, data tag, or the return value of antoher function. If the value is hard coded and contains a negative or postiive sign, the entire value must be enclosed in double quotes, as in “-10.23”.

Parameters
None

<<diff>>

The `<<diff...>>` function provides the operation of the minus sign (-) operator in other languages. This function subtracts the second argument from the first and returns the difference. This function takes two and only two arguments.

String data tags or string return values may be passed to the function, provided those values contain only numeric, sign, and a single decimal character.

Arguments

<<diff numArg1 numArg2>>	
numArg1	The first numerical value from which `numArg2` is subtracted. May be a hard coded value, data tag, or the return value of another function. If the value is hard coded and contains a negative or positive sign, the entire value must be enclosed in double quotes, as in “-12.34”.
numArg2	The second numerical value that will be subtracted from `numArg1`. May be a hard coded value, data tag, or the return value of another function. If the value is hard coded and contains a negative or positive sign, the entire value must be enclosed in double quotes, as in “-10.23”.

Parameters
None

<<prod>>

The `<<prod...>>` function provides the operation provided by the multiplication operator, either x, or more commonly *, in other languages. This function multiples the first argument by the second and returns the product.

A String data tag may be passed as an argument to the function, provided it contains only numerical characters, sign, and a single decimal character.

Arguments

<<prod numArg1 numArg2>>	
numArg1	This required argument contains the value that will be multiplied by `numArg2`. May be a hard coded value, data tag, or the return value of antoher function. If the value is hard coded and contains a negative or postivie sign, the entire value must be enclosed in double quotes, as in "-12.34".
numArg2	This required argument contains the value to multiplied by `numArg1`. May be a hard coded value, data tag, or the return value of antoher function. If the value is hard coded and contains a negative or postiive sign, the entire value must be enclosed in double quotes, as in "-10.23".

Parameters
None

<<div>>

The `<<div...>>` function provides the same operation as is provided by the division sign (/) operator in other languages. This function divides the second argument into the first and returns the results.

A string data tag may be passed as an argument to the function, provided it contains only numerical characters, sign, and a single decimal character.

Arguments

<<div dividendArg divisorArg>>	
dividendArg	This required argument contains the number to be divided by the `divisor-Arg`. May be a hard coded value, data tag, or the return value of a antoher function. If the value is hard coded and contains a negative or postivie sign, the entire value must be enclosed in double quotes, as in "-12.34".

<<div dividendArg divisorArg>>	
divisorArg	This required argument contains the number to be divided into the `divi-dendArg`. May be a hard coded value, data tag, or the return value of another function. This value must not be 0. If the value is hard coded and contains a negative or postiive sign, the entire value must be enclosed in double quotes, as in “-10.23”.

Parameters
None

<<remainder>>

The `<<remainder...>>` function provides the modulus operation of the modulus sign operator, usually %, in other languages. This function divides the first argument by the second and returns the remainder of the division.

String data tags can be passed as arguments to the function, provided the value contains only numeric characters, sign, and a single decimal character.

Arguments

<<remainder dividendArg divisorArg>>	
dividendArg	The value to be divided by the divisorArg. May be a hard coded value, data tag, or the return value of another function. If the value is hard coded, and it contains a positive or negative sign, the entire value must be enclosed in quotes, as in “-12.34”.
divisorArg	The value to be divided into the `dividendArg`. May be a hard coded value, data tag, or the return value of another function. If the value is hard coded, and it contains a positive or negative sign, the entire value must be enclosed in double quotes, as in “-10.23”. The value of this argument must not be 0.

Parameters
None

<<local>>

The `<<local...>>` function allows you to create data tags within the script. The data tags created are always string values. Their scope is limited to the step within which they are created, and the other steps within the same parent definition that follow it. So, if an object contains 4 read steps, and the second contains a `<<local...>>` function call, the data tag or tags created will be available in the second step as well as the third and fourth. It will not be available in the first.

If the value for a `tagName` argument is a hard coded value, or contains a mixture of text and SDML, the value must be enclosed in quotes.

To reference a local data tag, the syntax is `<<local.tagName>>`, where `tagName` is the name given in the function call. Local data tags support the named parameter `length=`, which will truncate the string to the given value. String values are not dequoted; time and date values are not wrapped in any type of conversion function.

Arguments

<<local tagName1=value [tagName2=value...tagNameN=value]>>	
tagName1	This required argument is the name that will be given to the data tag created. Its corresponding value will be the value the tag contains.
tagName2-N	These optional arguments are the same as tag-Name1 and allow for the creation of multiple data tags with the same function call.

Parameters

- **length –** This optional named parameter takes a non-negative whole number and specifies the maximum number of characters to assign to the local data tag created by the function.

<<sql>>

The `<<sql...>>` function allows you to create data tags based on the data returned by a SQL statement, specifically a SELECT statement. All records returned by the statement are stored in the newly created data tag, commonly referred to as a SQL flunky. Each field of each record returned by the SELECT statement can be accessed in the flunky.

The flunky created is named the same as the argument you provide. This SQL flunky has a scope limited to the script within which it is contained.

The `<<sql...>>` function is normally used to retrieve a small number of records, usually consisting of one or two selected fields, to retrieve data using a simple SELECT statement, where otherwise it may be necessary to create a more complex statement within the script. The statement used in the function call should never be used to perform the main processing of the script, nor to return large numbers of records. Rather, it should be used to aid in this main processing. Furthermore, the statement should never contain UPDATE or INSERT statements. Additionally, if the SELECT statement is returning more than 10 records at a time, the design of your script should be reevaluated and adjusted so that this is not the case. The main reason for this is performance.

While the `<<sql...>>` function will not cause any delays or hitches in processing if used correctly, using it to return large amounts of data will slow down the processing of the Agentry Server considerably. Each record returned by the function must be processed by the Agentry Server and stored in memory until the script has completed processing. This can tie up a significant amount of the system resources in the event of a large number of records being returned.

As stated, the SQL flunky created by the `<<sql...>>` function call is only in scope within the script in which it is called. If the value is needed in other Steps within the same parent definition, the desired values can be assigned to a local flunky, via use of the <<local...>> function described previously.

The syntax to reference the SQL flunky created by this function is as follows

```
<<sql.nameArg[.recordIndex][.fieldName]>>
```

All SQL flunkies are referenced beginning with `sql`. The `nameArg` is the name of the argument as you provided when calling the `<<sql...>>` function. The `recordIndex` is a numerical value indicating which record within the data set you wish to access. The records are referenced in the order in which they were returned by the database system, and are indexed starting with 0. The field name is the name of the column, or its alias, that contains the data you wish to retrieve. So, to access a field named COST in the first record of a SQL flunky named `prodCost`, the tag would be `<<sql.prodCost.0.COST>>`

Arguments

<<sql nameArg="SQLStatement">>	
nameArg	The name of the SQL flunky to be created as a result of processing the argument value, `SQLStatement`. The `SQLStatement` must always be enclosed in double quotes and should contain a `SELECT` statement.

Parameters
None

<<include>>

The `<<include...>>` function allows you to include the contents of another file within the file calling the function. This content will be included at the point where the function call is placed. The included file should always be a plain text file.

This function is only used in specific cases and there are certain caveats that accompany its usage. These caveats are related to the fact that the file referenced does not need to be associated with any definition within the Agentry Editor. Because of this fact, the included file may not be controlled or monitored by the Editor. This means that, during a publish, this file will not be copied or transferred in any way to the Agentry Server. Therefore, changes made to this file will not be updated to the Server during a publish, meaning the file must be moved separately if changes are made to it.

Related to this, if the included file does not exist in a location that is accessible to both the Editor and Server, it must be copied to two separate locations, one for each of these components.

Arguments

<<include fileName>>	
fileName	The name of the file whose contents are to be included in the file calling the function.

Parameters
None

Agentry Test Script Overview

The Agentry Test Script is an XML schema supported by the Agentry Test Environment that can be used to automate testing the Client behavior of a mobile application built on Agentry. The Agentry Test Environment includes a script recorder that allows for the recording of test scripts, and can then play back those test scripts.

The test script language includes the ability to interact with all controls present on the client application’s interface, including field selection, data entry, button clicks, and navigation. Additionally, this language also supports the ability to check the current values of labels, fields, and other items displayed on the client application’s interface for expected values.

In addition to direct client interaction, the test script also includes the ability to query database systems for expected values. This can be used after transmit to verify the proper functioning of transactions related to the back end processing that is defined within those transactions.

Elements within the test script XML schema are logically grouped into the following categories:

- **Script Elements:** Elements for the script itself, including the top-level <script> element and elements related to logging and script execution.
- **Button Elements:** Elements that allow for interaction with button definitions, including selection (or “clicking”), checking the state of the button, and label values.

- **Field Elements:** Elements that allow for interaction with detail screen fields. Note that certain field edit types are supported by elements in other groups.
- **List Elements:** Elements for working with list controls of various types. This includes list controls on list screens, as well as the various list types that can be defined for detail screen fields.
- **Tree Elements:** Elements for working with tree controls. This includes tree controls presented by detail screen fields.
- **Scanner Elements:** Elements for simulating scanner behaviors, including passing values in as barcode scanner values.
- **SQL Elements:** Elements for creating connections to and running queries against database back end systems. Values can be returned by these queries and checked against expected values.
- **Tab Elements:** Elements for working with the tab controls presented by screen sets for each child screen definition.
- **Window Elements:** Elements for closing windows on the client. Rarely used, as navigational actions defined to close screen sets should be used wherever present.
- **Client Elements:** Elements for affecting the client process, including restarting and other behaviors.
- **Client Host Elements:** Elements to interact directly with the client device, which may in turn affect the test client, such as entering key strokes or executing commands on the client device.

Common Test Script Element Attributes

The following attributes are common to the bulk of the elements within the Agentry Test Script XML schema. They relate primarily to time outs for the execution of a given element, and the amount of time to pause between the execution of one element and the next. Setting these attributes in the <script> element of the test script will set defaults for the entire script execution that can then be overridden by individual child elements if needed.

Name	Description	Data Type	Default Value	Re-quired
timeout	The amount of time to wait for the element to finish processing before returning an error. This value can be set in the <script> element for the entire script and/or at each processing element within the test script. Child elements with this attribute will override the value set in parent elements. The value is specified in milliseconds.	Positive Integer	N/A	No

Name	Description	Data Type	Default Value	Required
sleep	The amount of time to pause after the element is executed. This value can be set in the <script> element for the entire test script and/or at each processing element within the test script. Child elements with this attribute will override the value set in parent elements. The value is specified in milliseconds.	Positive Integer	N/A	No

Agentry Test Script: Script Elements Overview

The script elements within the Agentry Test Script language include the top-level `<script>` element that is the root to all test scripts, as well as elements for logging messages and pausing the execution of the script.

When a new test script is created by the Script Recorder in the ATE, it automatically creates the `<script>` element and required attributes. The other elements `<script-log>` and `<script-pause>` are manually added when needed.

Included in the `<script>` element is the attribute specifying the name space for the Agentry Test Script language, which is `xmlns:ags="urn:script.Agentry.Syclo"`. If a script is created manually this should be an attribute included in the `<script>` element.

<script>

The `<script>` element is the root element for any Agentry Test Script. All elements are contained within the `<script>` element, either directly or as descendents. Two of its attributes, timeout and sleep, will affect how each element it contains is processed. The `timeout` attribute sets the duration of time to wait for an element to be processed. Setting the timeout in the `<script>` element will set a timeout for all elements. Other elements may individually override this duration with their own `timeout` attributes. The `sleep` attribute will set the amount of time to wait for before processing an element within the test script. Setting this attribute for the `<script>` element will affect all elements, waiting to process each for the configured time. Other elements may individually override this duration with their own `sleep` attributes.

Structure

Contained By:

- None - Root element for Agentry Test Script files.

Table 11. Attributes

Name	Description	Data Type	Default Value	Required
xmlns	This attribute defines the base namespace.	String	`urn:script:Agentry:Syclo`	Yes
xmlns:ags	This attribute defines the `ags` namespace, making it the default namespace.	String	`urn:script:Agentry.Syclo`	Yes
xmlns:meta	This attribute defines the `meta` namespace used for comments.	String	`urn:meta:Editor.Agentry.Syclo`	Yes
show-execute	This attribute enables or disables displaying log messages from the test script on standard output. When set to true log messages are written to standard output.	Boolean	False	No
common script attributes	This element includes the following common script attributes: • timeout • sleep	N/A	N/A	N/A

<script-log>

The <script-log> element will write a log statement to the log file `AgentryScriptOutput.log`. The contents of this element are the message written to the log file. The log message level must also be specified in the level attribute to the element, which indicates the severity of the log message.

Structure
Contains:

- Text - The log message to be written to the `AgentryScriptOutput.log` log file.

Contained By:

- `<script>`

Table 12. Attributes

Name	Description	Data Type	Default Value	Required
level	This attribute can contain either a numeric or string value indicating the severity of the log message. The following list includes both the numeric and string values, only one or the other of which should be used for this attribute: • 1 - critical • 2 - high • 3 - mediumHigh • 4 - medium • 5 - mediumLow • 6 - low • 7 - veryLow	String	N/A	Yes
common script attributes	This element includes the following common script attributes: • timeout • sleep	N/A	N/A	N/A

<script-pause>

The `<script-pause>` element will pause playback of the test script, displaying a popup dialog within the Agentry Test Environment. The playback will not resume until this popup message is acknowledged.

Structure
Contains:

- None

Contained By:

- `<script>`

Attributes: None

Agentry Test Script: Button Elements Overview

The button-related elements available in the Agentry Test Script structure can be used to simulate a user pushing a button on a screen as a part of a sequence of interactions. Additionally, these elements can be used to test the state of the button, such as enabled or disabled, its label value, and so forth, and to wait for the button to be enabled for attempting to push it.

Common Button Element Attributes

The following attributes are available to all button elements within the Agentry Test Script language:

Name	Description	Data Type	Default Value	Re-quired
id	The identifier of the button, typically set by the script recorder in the ATE and not modified manually. If specified, the `name` and `label` cannot be present.	String	N/A	No
name	The resource name of the button the element affects or monitors. If specified, the `id` and `label` attributes cannot be present.	String	N/A	No
label	The label text of the button definition the element affects or monitors. If specified, the `id` and `name` attributes cannot be present.	String	N/A	No

<button-expect>

The `<button-expect>` element is used to verify the state of a button definition within a screen. This includes the button's label text, enabled state, checked state, whether or not it is a popup button, and whether or not it is visible. The type of button definition checked by this element will dictate the supported states and other expected values for the button definition.

If any of the configured expected state information is not matched by the button, a script error is thrown.

Structure

Contains:

- Text - The expected label for the button definition as displayed on the screen.

Contained By:

- `<script>`

Table 13. Attributes

Name	Description	Data Type	Default Value	Required
enabled	This attribute specifies the expected enabled state of the button. The value `t` is true, meaning the button is expected to be enabled. The value `f` is false, meaning the button is expected to be disabled.	string	`t`	No
checked	This attribute specifies the expected checked state of the button, which is either checked or unchecked. The value `t` is true, meaning the button is expected to checked. The value `f` is false, meaning the button is expected to not be checked.	string	`t`	No
popup	This attribute specifies whether the button is expected to be an Action Button with a defined action of popup menu. The value `t` is true, meaning the button is expected to be a popup menu. The value `f` is false, meaning the button is not expected to be a popup menu.	string	`f`	No
visible	This attribute specifies whether the button is expected to be visible or not. The value `t` is true, meaning the button is expected to be visible. The value `f` is false, meaning the button is not expected to be visible.	string	`t`	No
common button attributes	For `<button-expect>` these attributes specify the expected related items for each attribute.	N/A	N/A	N/A
common script attributes	A set of related attributes common to most elements within the Agentry Test Script.	N/A	N/A	N/A

<button-push>

The `<button-push>` element is used to push, or click, a button on the current screen. The button can be pushed in combination with the `Shift`, `Ctrl`, and/or `Alt` keys when in a <script> context as optional attributes to the element. If the button is disabled when this element is processed a script error is thrown.

In a `<field-popup>` context the element will push a button on the popup dialog displayed. The contents of the `<button-push>` element contain the label of the button to be pushed. In this context none of the attributes are supported.

Structure
Contains:

- In a `<script>` context any text contents ignored.
- In a `<field-popup>` context this element contains text that specifies the button to push in the popup screen. Valid contents include:
  - 0-9
  - +/-
  - . (decimal)
  - “Back,” “Clear,” or “Close”

Contained By:

- `<script>`
- `<field-popup>`

Table 14. Attributes - *`<script>`* *context only*

Name	Description	Data Type	Default Value	Required
shift	This attribute specifies whether the `Shift` key should be held down in combination with the button push. The value `t` is true and the `shift` key will be held down. The value `f` is false.	string	`f`	No
ctrl	This attribute specifies whether the `Ctrl` key should be held down in combination with the button push. The value `t` is true and the `Ctrl` key will be held down. The value `f` is false.	string	`f`	No
alt	This attribute specifies whether the `Alt` key should be held down in combination with the button push. The value `t` is true and the `Alt` key will be held down. The value `f` is false.	string	`f`	No
check	This attribute specifies whether the control should be checked or unchecked. The value t is true and will result in the control being checked. The value f is false. This attribute is valid only for check box controls on built-in client screens.	string	`t`	No

Name	Description	Data Type	Default Value	Required
common button attributes	For `<button-push>` these attributes specify the button to be pushed.	N/A	N/A	N/A
common script attributes	A set of related attributes common to most elements within the Agentry Test Script.	N/A	N/A	N/A

<button-wait>

The `<button-wait>` element will force the test script to wait for the specified button's state to change either from enabled to disabled, or from disabled to enabled, depending on the element's configuration. The script will wait until either the specified state change occurs or until the timeout value for the `<script>` or the `<button-wait>` element is reached. If the button's state does not changed before the timeout has elapsed a script error will be thrown.

Structure

Contains:

- None

Contained By:

- `<script>`

Table 15. Attributes

Name	Description	Data Type	Default Value	Required
enabled	This attribute specifies whether to wait for the button to be enabled or disabled. The value `t` is true and the element will wait until the button is enabled. The value `f` is false and the element will wait until the button is disabled.	string	`t`	No
common button attributes	These attributes when set for the `<button-wait>` element specify the button to be monitored for a state changed.			

Name	Description	Data Type	Default Value	Required
common script attributes	A set of related attributes common to most elements within the Agentry Test Script.	N/A	N/A	N/A

Agentry Test Script: Field Elements Overview

The elements related to detail screen fields within the Agentry Test Script are provided to allow for interaction with the detail screen fields of the application. This includes field selection, entering values, testing for expected values, pushing or selecting fields with an edit type of button, and interacting with date, time, and duration related field types.

Common Field Element Attributes

The following lit includes the common attributes for most field-related XML elements within the Agentry Test Script. These attributes are used by the elements in different ways depending on the purpose and behavior of the element.

Name	Description	Data Type	Default Value	Required
id	The identifier of the field definition, typically set by the script recorder in the ATE and not modified manually. If the `id` attribute is specified, the `name` and `label` attributes cannot be present.	String	N/A	No
name	The resource name of the field the element affects or monitors. If the `name` attribute is specified, the `id` and `label` attributes cannot be present.	String	N/A	No
label	The label text of the field definition the element affects or monitors. If the `label` attribute is specified, the `name` and `id` attributes cannot be present.	String	N/A	No

<field-button-push>

The `<field-button-push>` element will push the button control for a detail screen, provided that field's edit type includes a button control. Following is a list of the field edit types for which this element will push a button:

- Barcode Scan (when defined to include a scan button)

- Button
- List Tile View - Add, Edit, and Filter Buttons
- Complex Table Search
- Complex Table Drop Down
- Complex Table List
- Complex Table Tree
- Data Table Selection (displays drop down list or popup list view based on field's definition)
- External Data

Structure
Contains:

- None

Contained By:

- `<script>`

Table 16. Attributes

Name	Description	Data Type	Default Value	Required
buttonLabel	This attribute contains the label text of the field's button that is to be pushed. Many field's button controls do not contain text, or the text changes based on theme. It is recommended that the `label` attribute be used to specify the label of the field containing the button control to be pushed.	string	none	No
buttonControl	This attribute specifies the control ID of the button control that is to be pushed. This is normally set by the Script Recorder within the ATE under specific circumstances and is normally not set when manually editing a test script.	string	none	No
common button attributes	The following common button attributes are a part of this element: • name • label • id	N/A	N/A	N/A

Name	Description	Data Type	De-fault Value	Re-quired
common script attrib-utes	A set of related attributes common to most elements within the Agentry Test Script.	N/A	N/A	N/A

<field-expect>

The <field-expect> element allows for the validation of a detail screen field's value and state. State information includes hidden or visible, and enabled or disabled. The contents of the field that can be validated include its current value, for duration fields the current value of each portion of the duration value (hours, minutes, seconds), for list fields the total number of rows in a list, or the current value of a row or rows within a list. When validating the value of a field, the validation attribute should be set indicating the type of validation. To validate multiple facets of a field, such as current value and enabled or disabled, multiple `<field-expect>` elements are required to validate each facet. To validate the value of multiple rows within a list, the `<field-expect>` element must contain on `<row>` element for each row to be validated. Note that only drop down lists may be validated by this element. For list view or list tile view fields, the `<list-expect>` element must be used. If the field does not match the expected criteria as specified by the `<field-expect>` element, a script error is thrown.

Structure

Contains:

- Text - The value that the field is expected to contain, or the value of the state being checked by the element.
- `<row>`

Contained By:

- `<script>`

Table 17. Attributes

Name	Description	Data Type	Default Value	Required
type	This attribute specifies what is being checked for within the field by this element. Valid values include: • string - The field's string value • bool - The field's Boolean value. • long - The field's integral number value. • decimal - The field's decimal number value. • validateValue - The field's value. • enabled - The field's enabled or disabled state. • visible - The field's visible or hidden state. • list - The field is a drop down list. • format - The label for a button field.	string	string	No
special	This attribute specifies that the field's value should or should not be equal to its defined special value. The value `t` is true and the field is expected to be set to its special value. The value `f` is false.	string	f	No
part	This attribute is valid only when checking a duration field. It specifies the portion of the duration value to be checked. Valid values for this attribute include: • hours - The hours portion of the duration. • decimalHours - The hours portion of the duration as a decimal. • minutes - The minutes portion of the duration. • seconds - The seconds portion of the duration.	string	none	Required for duration field edit types. Otherwise ignored.

Name	Description	Data Type	Default Value	Required
row	The row index to be checked by the element when the field is a drop down list. This index is 0-based, meaning the first row in the list is at position 0. This attribute is only valid when the edit type of the field is Complex Table Drop Down, Data Table Selection, or List Selection. If the `<field-expect>` contains one or more `<row>` elements, the first row checked is indicated by this attribute. Additional `<row>` elements are expected to be contained in the list in the order in which their corresponding `<row>` elements are contained in the `<field-expect>`.	non-negative integer	none	Required for drop down list fields. Otherwise ignored.
count	This attribute specifies the expected number of rows in the list.	non-negative integer	none	No
common field attributes	This element contains the following common field attributes: • `name` • `id` • `label`	N/A	N/A	N/A
common script attributes	A set of related attributes common to most elements within the Agentry Test Script.	N/A	N/A	N/A

<field-label-select>

The `<field-label-select>` element allows for selecting, or “clicking”, the label of a field when that field label is defined as a hyperlink. The field’s label to be selected is specified using one of the `name` or `label` attributes. If the field’s label is not a hyperlink, or if it cannot be selected for some other reason (e.g. the action it executes is disabled) a script error will be thrown.

Structure

Contains:

- None

Contained By:

- `<script>`

Table 18. Attributes

Name	Description	Data Type	Default Value	Required
common field attributes	This element includes the following common field element attributes: • name • label • id			
common script attributes	A set of related attributes common to most elements within the Agentry Test Script.	N/A	N/A	N/A

<field-popup>

The `<field-popup>` element will open the numeric popup screen that allows for the entry of numeric values into a detail screen field. This element is valid for fields with an edit type of Decimal Number, Integral Number, or Duration. This element can contain the `<edit-select>` and `<button-push>` elements to select values in the popup screen and to push the buttons on the popup to enter values, respectively. If this element is used for field with an edit type other than those it supports a script error will be thrown.

Structure

Contains:

- `<edit-select>`
- `<button-push>`

Contained By:

- `<script>`

Table 19. Attributes

Name	Description	Data Type	Default Value	Required
part	This attribute specifies the portion of a duration field for which the popup screen will be displayed. This attribute is ignored for other field edit types. Valid values for this attribute include: • hours • minutes • seconds	string	none	Required for Duration fields. Otherwise ignored.
common field attributes	This element includes the following common field attributes: • name • label • id	N/A	N/A	N/A
common script attributes	A set of related attributes common to most elements within the Agentry Test Script.	N/A	N/A	N/A

<edit-select>

The `<edit-select>` element selects the numeric values in a field by position. This element is contained by the `<field-popup>`, which specifies the field and, for duration fields, the part of the field in which the selection is made. The `<edit-select>` element selects the characters based on position within the field as specified by its `start` and `end` attributes. The first character is at position 0. All characters from the specified `start` up to and including the `end` character are selected.

Structure

Contains:

- None

Contained By:

- `<field-popup>`

Table 20. Attributes

Name	Description	Data Type	Default Value	Required
start	The first character within the field to select. Characters are specified using a 0-based index, meaning the first character is at position 0 within the field.	non-negative integer	0	Yes
end	The last character within the field to select. Characters are specified using a 0-based index, meaning the first character is at position 0 within the field.	non-negative integer	0	Yes

<field-set>

The `<field-set>` element sets the value of a detail screen field. The contents of the field specify the value to be set. The attributes name or label are used to specify which field to set. Other attributes can be used to set the value of the field to its defined special value, to check or uncheck a check box field, to select or deselect a radio button field, or to set just a portion of a date, time, date and time, or duration field. This element is used to set the value of most fields, regardless of field type. Other elements exist to allow for setting field values but should only be employed in less common situations. For most testing purposes the `<field-set>` element is sufficient. It is the element inserted into a test script generated by the Test Script Recorder within the Agentry Test Environment.

Structure

Contains:

- Text - The value to which the field will be set.

Contained By:

- `<script>`

Table 21. Attributes

Name	Description	Data Type	Default Value	Required
special	This attribute specifies whether the field should be set to its defined special value. The value `t` is true and the field will be set to its defined special value. The value `f` is false.	Boolean	f	No

Name	Description	Data Type	Default Value	Required
part	This attribute specifies the portion of the field to set. This attribute is only valid for fields with an edit type of Date, Time, Date and Time, or Duration. Valid values for this attribute include: • year • month • day • hours • minutes • seconds	string	none	Required for date, time, and duration fields. Otherwise ignored.
checked	This attribute specifies whether or not to check or select a field with an edit type of Button that is either a check box or radio button. The value `t` is true and will check or select the box or radio button. The `f` is false and will uncheck or deselect the button field.	Boolean	f	Required for Button fields of type radio or check box. Otherwise ignored.
common field attributes	This element includes the following common field attributes: • name • label • id	N/A	N/A	N/A
common script attributes	A set of related attributes common to most elements within the Agentry Test Script.	N/A	N/A	N/A

Agentry Test Script: List Elements Overview

The list-related elements of the Agentry Test Script allow for interaction with the various list controls of the Agentry Client. These include list screens, and the various list field edit types that can be defined for detail screens. The list-related elements support the selection of items in a list, double-clicking items, checking for the expected values of list items and column headers, expected values of list headers and detail panes.

Common List Element Attributes

The following attributes are found in most of the list-related elements of the Agentry Test Script. The purpose and use of these attributes depends on the nature of the element for which they are set.

Name	Description	Data Type	Default Value	Required
id	The identifier of the list definition, typically set by the script recorder in the ATE and not modified manually. If the `id` attribute is specified, the `name` and `label` attributes cannot be present.	String	N/A	No
name	The resource name of the list the element affects or monitors. If the `name` attribute is specified, the `id` and `label` attributes cannot be present.	String	N/A	No
label	The label text of the list (if applicable) the element affects or monitors. If the `label` attribute is specified, the `name` and `id` attributes cannot be present.	String	N/A	No
row	The row within the list to be affected by the element. This may be the row number, with the first row in the list at position zero (0), or one of the values: • selected - currently select row • first - first row in the list • last - last row in the list • next - the next row after the currently selected one • previous - the previous row after the currently selected one	String	N/A	No

<list-double-click>

The `<list-double-click>` element allows for an item to be double-clicked within a list. The item to be double-clicked may be the currently selected item, or this element can specify the item. To specify an item the row number can be used or text can be specified to select the item. If a screen contains multiple lists the field can be found by specifying the name or label for the the field definition. This element can be used with list screens to double-click an item in the list control on the screen, or with detail screens containing a field with an edit type of List View.

Structure

Contains:

- None

Contained By:

- `<script>`

Table 22. Attributes

Name	Description	Data Type	Default Value	Required
common list attributes	This element includes the following common list attributes: • name • id • label • row	N/A	N/A	N/A
common script attributes	A set of related attributes common to most elements within the Agentry Test Script.	N/A	N/A	N/A

<list-expect>

The `<list-expect>` element verifies the contents of a list control on a list screen. It can also verify a detail screen field with an edit type of List View. This element can verify the contents of the list, including the number of rows display, the number of selected rows, the values displayed in each column for each row, the presence of a value in a column in any row, the contents of the list's header label and detail pane, and the contents of the column header on each column within the list. When used on a detail screen containing multiple lists, the specific list to verify can be found using the field's name or label. If the list does not meet the expected criteria a script error is thrown. The expected values and state of the list and its items can be specified using the elements attributes as well as the elements it can contain.

Structure

Contains:

- Text - The value to use to locate the desired row to verify within the list. Text content and element content are mutually exclusive for the `<list-expect>` element.

- `<row>`
- `<column>`
- `<header>`
- `<columnheader>`
- `<detail>`

Contained By:

- `<script>`

Table 23. Attributes

Name	Description	Data Type	Default Value	Required
column	This attribute specifies the column to be verified by the <list-expect> element. This may be either the column's header label or a numeric value indicating the columns position from left to right, with the left-most column at position 1.	string	None	No
count	This attribute specifies the total number of rows the list should contain. This may any numeric integer no less than zero.	non-negative integer	None	No
selected-count	This attribute specifies the total number of rows currently selected in the list. This may any numeric integer no less than zero.	non-negative integer	None	No
common list attributes	This element includes the following common list attributes: • name • id • label • row • column	N/A	N/A	N/A
common script attributes	A set of related attributes common to most elements within the Agentry Test Script.	N/A	N/A	N/A

<list-select>

The `<list-select>` element will select an item in a list. This element can be used to select items in the list control of a list screen, in detail screen fields with an edit type of List View, or when the edit type of a detail screen field is List Tile View. This element may also be used to deselect an already selected item in the list. List items can be selected based on row position or column value. The contents of the `<list-select>` element will be the value to search for in a specified column when select by column value. Of the row cannot be found or selected in the list a script error is thrown.

Structure

Contains:

- Text - The value of a column by which the item will be found and selected.

Contained By:

- `<script>`

Table 24. Attributes

Name	Description	Data Type	Default Value	Required
column	This attribute specifies the column by which the item will be found. This may be either the columns header label text or a numeric value specifying the column's position from left to right, with the left-most column at position 1. This attribute is set to "none" for List Tile View detail screen fields.	String	None	Required when the `row` attribute is set to a value of "text."
select	This attribute specifies whether to select or deselect the row. The value `t` is true and will result in the row being selected. The value `f` is false and the row will be deselected.	String	`t`	No
common list attributes	This element includes the following common list attributes: • name • id • label • row			
common script attributes	A set of related attributes common to most elements within the Agentry Test Script.	N/A	N/A	N/A

<list-sort-by>

The <list-sort-by> element will sort a list by a specified column. This element can sort the items of a list control on a list screen, or the items in detail screen field with an edit type of list view. The column to sort the list on is specified in the elements column attribute. If the specified column cannot be found or if the list cannot be sorted on the column a script error is thrown.

Structure
Contains:

- None

Contained By:

- `<script>`

Table 25. Attributes

Name	Description	Data Type	Default Value	Required
column	This attribute specifies the column upon which the list should be sorted. This may be either the column header label text or the position of the column from left to right, with the left-most column at position 1.	string	None	Yes
common list attributes	This element includes the following common list attributes: • name • id • label	N/A	N/A	N/A
common script attributes	A set of related attributes common to most elements within the Agentry Test Script.	N/A	N/A	N/A

<detail>

The `<detail>` element can be contained in a `<list-expect>` element. When present the `<detail>` element must then contain a `<text>` element. The contents of the `<text>` element is then the text expected to be found in the detail pane of a list screen or a detail screen field with an edit type of List View. If the contents of the detail pane do not match the contents of the `<text>` element a script error is thrown.

Structure
Contains:

- <text> - This element contains the text expected to be found in the detail pane of the list.

Contained By:

- `<list-expect>`

Attributes: None

<header>

The `<header>` element can be contained in a `<list-expect>` element and, when present must contain a `<text>` element. The contents of the `<text>` element is then the text expected to be found in the lists header label. This element is valid only when the `<list-expect>` element containing it is for a list screen or a detail screen field with an edit type of List View. If the header pane does not match the contents of the `<text>` element a script error is thrown.

Structure
Contains:

- `<text>` - This element contains the expected text in the lists header label.

Contained By:

- `<list-expect>`

Attributes: None

<columnheader>

The `<columnheader>` element verifies the label displayed on a column header in a list. This element is contained in a `<list-expect>` element and, when present must contain a `<text>` element. The contents of the `<text>` element is the text expected to be in the label of the column header. This element can be used to verify header labels for columns in the list control of a list screen or the columns in a detail screen field with an edit type of List View. If the header of the specified column does not match the contents of the `<text>` element the or if the specified column cannot be found a script error is thrown. The column to verify is specified using the `<columnheader>` attribute `column`.

Structure
Contains:

- `<text>`

Contained By:

- `<list-expect>`

Table 26. Attributes

Name	Description	Data Type	Default Value	Required
column	This attribute can specify the column header to be verified as a part of the parent <list-expect> element processing. The contents of this attribute is either the column header label or the column index, with the left-most column at position 1.	string	none	No

<row>

The `<row>` element verifies the contents and state of a row in a list control. This element can be contained by the `<field-expect>` and `<list-expect>` elements. Multiple `<row>` elements may be contained by the same parent element to verify multiple rows in the same list. The `<row>` element can specify the row to be verified, or this may be specified by the parent element's row attribute. The contents of the row can include the text value indicating the expected contents of the row.

The `<row>` element may also contain one or more `<column>` elements. Each `<column>` element will verify the expected contents and/or state of the column it identifies within the row identified by the `<row>` element.

Structure

Contains:

- `<column>`

Contained By:

- `<field-expect>` - Only contained by this element when verifying the contents of a field with an edit type that displays a list control on the detail screen.
- `<list-expect>`
- Text - The expected contents of the row.

Table 27. Attributes

Name	Description	Data Type	Default Value	Required
row	This attribute contains the row position within the list. The top-most row is at position 1. Note that the same list may have a different row at the same numeric position based on sorting of the list.	positive integer	None	No
selected	This attribute specifies the expected selected state of the row. This is a Boolean value. The value t is true and the row is expected to be selected. The value f is false and the row is expected to not be selected.	Boolean	none	No

<menu-expect>

The `<menu-expect>` element verifies the contents and state of a menu. This element must contain at least one `<menu>` element specifying the menu to be verified. The `<menu>` element itself will likely contain other elements regarding the items within the menu.

Structure
Contains:

- `<menu>`

Contained By:

- `<script>`

Table 28. Attributes

Name	Description	Data Type	Default Value	Required
common script attributes	This element includes the following common script attributes: • timeout • sleep • frame	N/A	N/A	N/A

<menu-select>

The `<menu-select>` element is used to select a menu item. This element must contain a single `<menu>` element, which in turn must contain a single `<item>` element. These elements specify the menu and item to be selected by the `<menu-select>` element.

Structure

Contains:

- `<menu>`

Contained By:

- `<script>`

Table 29. Attributes

Name	Description	Data Type	Default Value	Required
common script attributes	This element includes the following common script attributes: • timeout • sleep • frame	N/A	N/A	N/A

<menu>

The `<menu>` element is contained by `<menu-select>` and `<menu-expect>`. The `<menu>` element identifies the menu to be acted on by the containing element. When contained by a `<menu-expect>` this element specifies what type of verification is to be performed. The `<menu>` element will contain one or more `<item>` elements identifying the menu item to be selected or verified. The `<menu>` element may also contain one or more `<separator>` elements when contained by a `<menu-expect>` element. The `<separator>` element will indicate the expected position of a menu separator.

Structure

Contains:

- Text - The name of the menu to be acted on by this element.
- `<item>`
- `<separator>`

Contained By:

- `<menu-select>`
- `<menu-expect>`

Table 30. Attributes

Name	Description	Data Type	Default Value	Required
type	This attribute is only used when the `<menu>` element is contained by a `<menu-expect>` element. The type attribute specifies what is to be verified within the named menu. Valid options for this attribute include: • exact - The `<item>` and `<separator>` elements contained by the `<menu>` element must match exactly with the contents of the menu. • sub-set - The `<item>` and `<separator>` elements contained by the `<menu>` element must exist within the menu, but others may also be present. • no separators - The `<item>` elements contained by the `<menu>` element must exist within the menu. Any separators within the menu are ignored and the `<menu>` element should contain no `<separator>` elements.	string	exact	No - valid only when `<menu>` is contained by a `<menu-expect>`

<item>

The `<item>` element is contained by the `<menu>` element, which in turn can be contained by a `<menu-expect>` or `<menu-select>` element. When the ancestor element is a `<menu-expect>` the `<item>` element specifies the expected state and value of the menu item identified by the `<menu>` element. When the ancestor is a `<menu-select>` the `<item>` element specifies the menu item to be selected. For both use cases the contents of the `<item>` element is the name of the menu item.

When the ancestor of the `<item>` element is a `<menu-expect>` the `<item>` element includes attributes to specify the expected state and other information about the named menu item. These include the enabled/disabled state and whether or not the menu item is checked (selected). These attributes are ignored when the ancestor element is a `<menu-select>`.

In a `<menu-expect>` context, if the expected state of the menu item does not match the attributes of the `<item>` attribute a script error is thrown. In a `<menu-select>` context if the menu item cannot be selected a script error is thrown

Structure

Contains:

- Text - The name of the menu item to be acted on.

Contained By:

- `<menu-select>`
- `<menu-expect>`

Table 31. Attributes

Name	Description	Data Type	Default Value	Required
checked	This attribute is only valid when the ancestor element is a <menu-expect> This is a Boolean attribute. The value t is true and indicates the menu item is expected to be checked. The value f is false and indicates the menu item is not expected to be checked.	Boolean	None	No
enabled	This attribute is only valid when the ancestor element is a <menu-expect>. This is a Boolean attribute. The value t is true and indicates the menu item is expected to be enabled. The value f is false and indicates the menu item is expected to be disabled.			

Agentry Test Script: Tree Elements Overview

The tree-related elements of the Agentry Test Script are used to work with tree controls presented on the Agentry client. This includes any detail screen fields that present a tree control. The tree-related elements should be used in place of the field-related elements for fields that present a tree control.

The elements in this group can be used to select nodes within a tree control, expand and collapse nodes, double-click nodes, a check for expected values of a node within the tree control.

Common Tree Element Attributes

The following list contains attributes found in most of the tree-related elements of the Agentry Test Script. The elements use these attributes differently depending on the purpose and behavior of the element.

Name	Description	Data Type	Default Value	Re-quired
name	The resource name of the tree control. If `name` is specified, `label` cannot be present.	String	N/A	No
label	The label of the tree control. If the `label` is specified, `name` cannot be present.	String	N/A	No
node	The node within the tree control to be affected by the element. The value of the attribute is the display value of the node in the tree control.	String	N/A	No
sibling	The sibling node to the currently selected node in the control. This attribute is set to the expected value of the node in the tree control.	String	N/A	No
child	The child node to the currently selected node in tree control. The value of this attribute is set to the expected value of the child node in the tree control.	String	N/A	No

<tree-select>

The `<tree-select>` element will select the specified node in a tree control. The node selected can be specified by its relationship to the currently selected node in the tree control, optionally in combination with that node's currently displayed text value.

Structure
Contains:

- Text - The text displayed for the tree node.

Contained By:

- `<script>`

Table 32. Attributes

Name	Description	Data Type	Default Value	Required
common tree control attributes	This element includes the following common tree control attributes: • name • label • node • sibling • child	N/A	N/A	N/A
common script attributes	This element includes the following common script attributes: • timeout • sleep	N/A	N/A	N/A

<tree-expect>

The `<tree-expect>` is used to verify the expected state and values of a tree control node. This element will contain one or more `<node>` elements. The `<tree-expect>` node specifies the node to be verified. Information the tree-expect node will verify includes the existence of the specified node and the total count of child nodes it contains. The `<node>` elements contained in a `<tree-expect>` then specify the expected child nodes of that node. The `<tree-expect>` node specifies the type of verification to perform for the specified node in the tree control. This options are to verify the child nodes match those `<node>` elements within `<tree-expect>` exactly, or to verify the child nodes include those `<node>` elements within the `<tree-expect>`. In the case of the latter verification, other child nodes may exist in the tree control. The first verification is referred to as an "exact" verification type. The latter is a "sub-set" verification. If the selected tree control node does not match the parameters of the `<tree-expect>` element, a script error is thrown.

Structure

Contains:

- Text - The expected contents of the specified node in the tree control.
- `<node>`

Contained By:

- `<script>`

Table 33. Attributes

Name	Description	Data Type	Default Value	Required
count	This attribute can specify the expected number of child nodes for the specified node being verified. If this attribute is not provided, the count of child nodes is not performed or checked.	non-negative integer	N/A	No
type	This attribute specifies the type of verification to perform. Valid values for this attribute are: • exact - All child nodes for the tree control node must exactly match all `<node>` elements contained in the `<tree-expect>` element. • sub-set - For each `<node>` element contained in the `<tree-expect>` element there must be a matching child node in the tree control Additional child nodes may also exist.	string	sub-set	No
common tree control attributes	This element includes the following common tree control attribute: • label • node • sibling • child	N/A	N/A	N/A
common script attributes	This element includes the following common script attributes: • timeout • sleep	N/A	N/A	N/A

<node>

The `<node>` element is contained by the `<tree-expect>` element. Multiple `<node>` elements may exist within the same `<tree-expect>`, with each representing an expected child node in the tree control. The contents of the `<node>` element is the text expected to be displayed for the node. If a `<node>` element does not match a corresponding node in the tree control, the containing `<tree-expect>` node will throw a script error. The order of the `<node>` elements must match the order of the expected nodes within the tree control.

Structure
Contains:

- Text - The expected value displayed for the child node in the tree control.

Contained By:

- `<tree-expect>`

Attributes: None

<tree-expand>

The `<tree-expand>` element will expand the currently selected node in a tree control. Optionally, a node can be specified, in which case the specified node will be selected and expanded by the element. If the node is already expanded this element will have no affect. If the specified node cannot be selected a script error is thrown.

Structure
Contains:

- Text - The currently displayed text of the tree control node to expand.

Contained By:

- `<script>`

Table 34. Attributes

Name	Description	Data Type	Default Value	Required
common tree control attributes	This element includes the following common tree control attributes: • label • node • sibling • child	N/A	N/A	N/A
common script attributes	This element includes the following common script attributes: • timeout • sleep	N/A	N/A	N/A

<tree-collapse>

The `<tree-collapse>` element will collapse a currently expanded node in a tree control. Optionally, the element can specify the node to be collapsed, in which case the node will first be selected and then collapsed. If a node in the tree control is not specified, the currently selected node will be collapsed. If the node is currently collapsed, this element will have no affect. If the specified tree control node cannot be selected a script error will be thrown. Note that if a node in the tree control exists but is currently hidden because its parent or ancestor node is not expanded, the `<tree-collapse>` node will not be able to select the node.

Structure

Contains:

- Text - The currently displayed value of the node to be collapsed.

Contained By:

- `<script>`

Table 35. Attributes

Name	Description	Data Type	Default Value	Required
common tree control attributes	This element includes the following common tree control attributes: • label • node • sibling • child	N/A	N/A	N/A
common script attributes	This element includes the following common script attributes: • timeout • sleep	N/A	N/A	N/A

<tree-toggle>

The `<tree-toggle>` element will toggle the specified node in a tree control; i.e., the specified node will be expanded if currently collapsed, or collapsed if currently expanded. If the specified node cannot be found in the tree control a script error will be thrown.

Structure

Contains:

- Text - The displayed text of the tree control node to be toggled

Contained By:

- `<script>`

Table 36. Attributes

Name	Description	Data Type	Default Value	Required
common tree control attributes	This element includes the following common tree control attributes: • label • node • sibling • child			
common script attributes	This element includes the following common script attributes: • timeout • sleep	N/A	N/A	N/A

<tree-double-click>

The <tree-double-click> element will double-click a node in the tree control. Optionally the node to double-click can be specified. Otherwise the currently selected node will be double-clicked. The node to double-click must currently be visible in the tree control. If the specified node cannot be selected a script error will be thrown.

Structure

Contains:

- Text - The currently displayed text for the tree control node to be double-clicked.

Contained By:

- `<script>`

Table 37. Attributes

Name	Description	Data Type	Default Value	Required
common tree control attributes	This element includes the following common tree control attributes: • label • node • sibling • child	N/A	N/A	N/A
common script attributes	This element includes the following common script attributes: • timeout • sleep	N/A	N/A	N/A

<tree-count-visible>

The `<tree-count-visible>` element will verify the number of currently visible nodes in a tree control. This count is based on the nodes a user can currently see on the client's screen, not the total number of nodes in the tree control. If the specified number of nodes is not currently visible in the tree control a script error is thrown.

Structure

Contains:

- Text - A non-negative value specifying the number of nodes expected to be currently visible in the tree control.

Contained By:

- `<script>`

Table 38. Attributes

Name	Description	Data Type	Default Value	Required
common tree control attributes	This element includes the following common tree control attributes: • label • node • sibling • child	N/A	N/A	N/A
common script attributes	This element includes the following common script attributes: • timeout • sleep	N/A	N/A	N/A

Agentry Test Script: Scanner Elements Overview

The scanner-related elements within the Agentry Test Script are used to test barcode scanner functionality. Included in this group of elements are those to start a barcode scan, set the scan value, and enable and disable the scanner simulator within the ATE.

It is important to note that these elements work directly through the barcode Scanner simulator within the Agentry Test Environment. The `<scan-enable>` element should be called to enable this simulator prior to calling the `<scan-start>` element, which passes the barcode value into the simulator, which in turn passes the value to the test client.

Alternately, if barcode scanner functionality is to be use, the barcode scanner can be enabled within the ATE prior to executing the test script. This will negate the need to enable it from within the script, which also requires the test client to be restarted.

<scan-data>

The `<scan-data>` element contains the default scan data to be passed by the Agentry Test Environment’s scan simulator to the test client. This element does not perform the actual scan and does not directly pass the data to the test client. Rather, this element contains the data to be passed to the test client when a scan is called for. The element `<scan-start>` will perform the actual scan and can contain a value to pass to the test client. If `<scan-start>` does not contain a value, and if it is preceded by a `<scan-data>` element, the value in `<scan-data>` will be the one passed to the test client.

Structure

Contains:

- Text - The default value to pass to the test client from the ATE’s scan simulator.

Contained By:

- `<script>`

Table 39. Attributes

Name	Description	Data Type	Default Value	Required
common script attributes	This element includes the following common script attributes: • timeout • sleep • frame	N/A	N/A	N/A

<scan-enable>

The `<scan-enable>` element will enable or disable the scan simulator in the Agentry Test Environment. When this element is processed it should be followed by a `<restart-client>` element in the test script, as enabling or disabling the scan simulator requires the test client to be restarted. The contents of this element can be the text values `true` or `false`, where `true` will enable the scan simulator and `false` will disable it.

Structure

Contains:

- Text - The `true` or `false` value to enable or disable the scan simulator in the Agentry Test Environment.

Contained By:

- `<script>`

Table 40. Attributes

Name	Description	Data Type	Default Value	Required
common script attributes	This element includes the following common script attributes: • timeout • sleep • frame	N/A	N/A	N/A

<scan-start>

The `<scan-start>` element will pass the value it contains to the test client through the Agentry Test Environment's scan simulator. Alternately, the value passed to the test client can be contained in a separate `<scan-data>` element. To pass this default value the proper syntax for the `<scan-start>` element is:

```
</ags:scan-start> -- OR -- </scan-start>
```

The use of an open and close marker for this element, such as: `<scan-start></scan-start>` will result in an error during test script playback.

Structure

Contains:

- Text - Option text value to be passed to the test client through the Agentry Test Environment's scan simulator.

Contained By:

- `<script>`

Table 41. Attributes

Name	Description	Data Type	Default Value	Required
common script attributes	This element includes the following common script attributes: • timeout • sleep • frame	N/A	N/A	N/A

Agentry Test Script: SQL Elements Overview

The SQL elements within the Agentry Test Script are provided to allow for the interrogation of the back end system processing built into the mobile application from within the test script. These elements do not affect the test client in any way. Rather, they are added to a test script to connect to and run queries against a database. Values returned by these queries can then be checked for expected values to validate the proper back end processing of the mobile application.

The primary use case for these elements is to check the proper processing of the mobile application's transactions. Typically these statements would be added to the script after a transmit is performed within the script that includes sending transactions to the Agentry Server for processing.

However, the SQL elements can be added at any point within the script where it is desirable to check the data in the back end as it relates to the mobile applications synchronization processing. Updates or changes made to exchange information and similar data can certainly be verified using these elements if desired.

The SQL elements are not added to the test script by the script recorder, but rather should be manually added to the script once it has been created.

Note that these elements are intended only for connecting directly to a database server. However, they can be used in testing an application with other types of system connections, provided there is direct access available to the database that may be behind the Java or Web Service interface.

<dsn-create-sql>

The `<dsn-create-sql>` element will create a System Data Source Name in ODBC representing a connection to a SQL Server database. This is a permanent DSN added to the host system and is not needed if an existing System DSN is already present. This element includes attributes specifying the DSN name, the database instance, SQL Server host, and authentication method (SQL or Windows). Any other parameters for a DSN are set to their defaults as configured on the host system. When using this element within a script, it is recommended that a corresponding `<dsn-remove-sql>` element exists to remove the same DSN created to allow for the same test script to be played multiple times on the same host system. If the DSN cannot be created a script error is thrown.

Structure

Contains:

- None

Contained By:

- `<script>`

Table 42. Attributes

Name	Description	Data Type	Default Value	Required
dsn	This attribute specifies the name given to the DSN. This value can contain no whitespace and must adhere to all requirements of a DSN name as specified in ODBC.	string	None	Yes
server	This attribute specifies the host system upon which the SQL Server service is running.	string	None	Yes
database	This attribute specifies the database instance to which the DSN should connect.	string	None	Yes
authentication	This attribute specifies the authentication method to be used to create a connection the SQL Server service and database. This may be one of: • SQL - The SQL Server authentication method. Connections made using this DSN will require the user login and password of a user account configured in the SQL Server service with permissions to access the database instance. • WINDOWS - The integrated Windows authentication method. The test script that uses the DSN created must be run as a Windows user known to the SQL Server system and with permissions to access the target database.	string	None	Yes
description	This attribute can contain a text value that will be added to the System DSN as a description.	string	None	No
overwrite	This attribute specifies whether or not to overwrite an existing System DSN with the same name as the one this element will created. This is a Boolean value, with true indicating the existing DSN should be overwritten and false indicating it should not.	Boolean	False	No

Name	Description	Data Type	Default Value	Required
ignoreErrors	This attribute specifies whether or not any errors returned when creating the System DSN should be ignored. This is a Boolean value, with true indicating errors should be ignored and false indicating they should not be ignored. If this attribute is false or not specified, any error returned when attempting to create the System DSN will result in a script error being thrown.	Boolean	False	No
common script attributes	This element includes the following common script attributes: • timeout • sleep	N/A	N/A	N/A

<dsn-remove-sql>

The `<dsn-remove-sql>` element is used to remove an ODBC System Data Source Name for a SQL Server database. The name of the DSN to be removed is specified in the `dsn` attribute to the element. Note that this element will permanently remove the DSN and will not be possible to recover it once this element has been processed.

Structure

Contains:

- None

Contained By:

- `<script>`

Table 43. Attributes

Name	Description	Data Type	Default Value	Required
dsn	The name of the ODBC System Data Source Name to be removed. The DSN named here must be for a SQL Server connection.	string	None	Yes

Name	Description	Data Type	Default Value	Required
ignoreErrors	This attribute specifies whether or not any errors returned with the attempt to delete the DSN should be ignored. This is a Boolean value with true indicating errors should be ignored. If this attribute is false or not specified, errors will not be ignored and, if any are returned, a script error will be thrown.	Boolean	False	No
common script attributes	This element includes the following common script attributes: • timeout • sleep	N/A	N/A	N/A

<sql-command>

The `<sql-command>` element will execute a SQL command against a database system. This element must be contained by a `<sql-connect>` element that creates a connection to the database system against which the SQL command will be executed. The `<sql-command>` element can contain the SQL statement to be executed, or it may reference a text file containing the command or commands to be executed. Results from executing the commands against a database can be logged based on the settings of the containing `<sql-connect>` element. The `<sql-command>` element can contain a `<sql-expect>` element that can verify the return set of any SELECT statement run by the `<sql-command>` element. Based on the `<sql-command>` element's `ignoreErrors` attribute setting, errors in executing the SQL command can be ignored or not. If errors are not ignored (default behavior) a script error will be thrown in the event an error occurs in executing the SQL statement. The `<sql-command>` element must either reference a file in its `commandFile` attribute containing the SQL command(s) to execute, or the text contents of the element itself must include a single SQL command.

Structure

Contains:

- Text - The SQL command to be executed against the database. Should not be present if a file is referenced in the `commandFile` attribute.
- `<sql-expect>`

Contained By:

- `<sql-connect>`

Table 44. Attributes

Name	Description	Data Type	Default Value	Required
commandFile	This attribute can contain the full path and file name, including file extension, of a plain text file containing the SQL command(s) to be executed against the database. This attribute should not be specified if the contents of the `<sql-command>` element include the SQL command to be executed.	string	none	No
ignoreErrors	This attribute specifies whether or not errors returned when attempting to execute the SQL command should be ignored. This is a Boolean value. When true errors will be ignored. When false errors will not be ignored and a script error will be thrown when an error occurs.	Boolean	False	No
common script attributes	This element includes the following common script attributes: • timeout • sleep	N/A	N/A	N/A

<sql-connect>

The `<sql-connect>` element will open a connection to a database server using the specified connection protocols and user credentials. The contents of this element include the `<sql-command>` element, which contains the SQL command to execute against the database with which the `<sql-connect>` element opens a connection. This connection will be closed after the processing of the `<sql-connect>` element and its contents has completed. In order for this element open a connection the host system upon which the test script is being played must have the proper configuration in place for the connection. For example the ODBC System DSN for a SQL Server database, or the TNS Name for an Oracle database connection. The attributes of this element specify whether or not to save the results of execution of any SQL commands to a log file.

Structure

Contains:

- `<sql-command>`

Contained By:

- `<script>`

Table 45. Attributes

Name	Description	Data Type	Default Value	Required
dbConnection-Name	The name of the connection configuration component for the target database; e.g. DSN, TNS Name, etc.	String	None	Yes
dbConnectionType	The type of connection to make. This must match the connection configuration component type named in dbConnectionName. Valid values for this attribute include: • DB2 • Informix • Interbase • MsSQL • ODBC (recommended setting for SQL Server connections) • Oracle • Postgre • SQLBase • SQLServer • Sysbase	String	None	Yes
dbConnectionU-serID	The user ID to connect to the database system.	String	None	Yes
dbConnection-Password	The password for the user ID.	String	None	Yes
commandFile	The full path and file name, including extensions, of a plain text file containing the SQL command(s) to execute once connected. This attribute is optional and should if used when one or more `<sql-command>` elements are contained in the `<sql-connect>` element, will be executed before those commands.	String	None	No

Name	Description	Data Type	Default Value	Required
saveQuery	This attribute specifies whether or not to log the SQL command executed against the database. This is a Boolean value, with true indicating the query should be saved. This attribute is ignored if `saveFile` is not present.	Boolean	False	No
saveResult	This attribute specifies whether or not to log the result of executing the SQL command(s) contained within the `<sql-connect>` element or any `<sql-command>` elements it contains. This is a Boolean value, with true indicating the results should be saved. Note that results are the responses of the database system, not the data returned by a query. This attribute is ignored if `saveFile` is not present.	Boolean	False	No
saveFile	This attribute can contain the full path and file name to which log messages from any SQL commands will be written. If this attribute is not set `saveQuery` and `saveResult` are ignored.	String	False	No
ignoreErrors	This attribute specifies whether or not any errors generated by the `<sql-connection>` element's operations should be ignored. This is a Boolean value, with true indicating errors should be ignored. If not set or set to false (default) any errors resulting from attempting to connect to the database, or in executing a SQL command, will result in a script error being thrown. Any `<sql-command>` elements contained in the `<sql-connect>` element can override this setting.	Boolean	False	No
common script attributes	This element includes the following common script attributes: • timeout • sleep	N/A	N/A	N/A

<sql-expect>

The `<sql-expect>` element is used to verify the return set from a SELECT statement executed from within a `<sql-command>` element. The `<sql-expect>` is contained within the `<sql-command>` that executed the command. It can then specify the expected number of records in the return set. The `<sql-expect>` in can also contain one or more `<sql-row>` elements, each of which will contain one or more `<sql-column>` elements. The `<sql-row>` and `<sql-column>` elements will verify the expected contents of the return set of the SELECT statement. The `<sql-expect>` element can also specify if the order of the return set should match the order of the `<sql-row>` elements it contains, or merely verify the existence of the same records, regardless of order. If the return set does not match the expected return a script error is thrown.

Structure

Contains:

- `<sql-row>`

Contained By:

- `<sql-command>`

Table 46. Attributes

Name	Description	Data Type	Default Value	Required
count	This attribute can specify the expected number of records, or row, in the return set. If this attribute is not specified, the number of rows is not factored into the validation of the return set.	non-negative integer	none	No
strictOrder	This attribute specifies whether or not the order of the rows in the return set is expected to match the order of `<sql-row>` elements contained in the `<sql-expect>` This is a Boolean attribute, with true specifying the order must match, and false specifying it does not. If false is specified, the rows in the return set can be in any order related to the order of `<sql-row>` elements. However, all `<sql-row>` elements must have corresponding `<records>`.			

Name	Description	Data Type	Default Value	Required
common script attributes	This element includes the following common script attributes: • timeout • sleep	N/A	N/A	N/A

<sql-row>

The `<sql-row>` element is contained in a `<sql-expect>` element. The `<sql-row>` element is generally a container for one or more `<sql-column>` elements, which specify the expected data in a given row. The `<sql-row>` element itself does not include any specific expected values. The order of the `<sql-row>` elements may impact the verification of the expected return set depending on the settings of the `<sql-expect>` element in which it is contained.

Structure

Contains:

- `<sql-column>`

Contained By:

- `<sql-expect>`

Attributes: None

<sql-column>

The `<sql-column>` element is contained in a `<sql-row>` element and specifies the expected data returned in that column. The `<sql-column>` element can specify the name of the column whose data is to be verified. Alternately, the `<sql-row>` element can contain `<sql-column>` elements with no name specification, in which case the data in the columns of the record must be in the same order as the `<sql-column>` elements within the `<sql-row>`. If the data specified in the `<sql-columnm>` elements contents does no match the data in the column within the row of the return set a script error is thrown.

Structure

Contains:

- Text - The expected value of the column within the row.

Contained By:

- `<sql-row>`

Table 47. Attributes

Name	Description	Data Type	Default Value	Required
name	This attribute can specify the name of the column within the record of the return set whose data is to be verified by the `<sql-column>` element. If this attribute is omitted, the `<sql-column>` elements must match the order of the columns in the rows of the return set or a script error is thrown. Note the either all `<sql-column>` elements within the same `<sql-row>` must specify the name attribute, or none of them can.	String	None	No

Agentry Test Script: Tab Elements Overview

The tab-related elements in the Agentry Test Script language are provided to allow for the selection of tabs within a screen set on the Client, and to test for expected values within the labels of those tabs.

In later versions of Agentry, screens within a screen set can be presented without tabs and instead using a popup menu to allow users to select a different screen within the screen set. The tab elements will work with this interface option as well.

<tab-expect>

The `<tab-expect>` element can be used to verify the label of a tab. The contents of the element specify the label expected to be displayed for the tab. The tab may be specified based on the screen definition's name, the screen's position index, a position relative to the currently selected tab's position, or by specifying the first or last tab from left to right. If the tab's label does not match the text contents of the `<tab-expect>` element a script error is thrown. If the tab, name, and label attributes are all omitted from this element, the currently selected tab's label is verified.

Structure

Contains:

- Text - The expected label for the tab.

Contained By:

- `<script>`

Table 48. Attributes

Name	Description	Data Type	Default Value	Required
tab	This attribute can be set to specify the tab to select based on its position relative to the currently selected tab, the first or last tab, or by specifying a positive integer matching the screen's position index within the screen set. Valid values for this attribute include: • first - The first (or left-most) tab displayed. • last - The last (or right-most) tab displayed. • next - The next tab to the right of the currently selected tab. • previous - The previous tab to the right of the currently selected tab. • A positive integer matching the screen definitions position within the screen set.	String	None	No
name	This attribute can be set to the screen definitions name to specify the tab whose label is to be verified.	String	None	No
label	This attribute can be set to specify the label for the tab whose label is to be verified. While supported for the `<tab-expect>` element, it makes little sense to use this attribute.	String	None	No
common script attributes	This element includes the following common script attributes: • timeout • sleep	N/A	N/A	N/A

<tab-select>

The `<tab-select>` element will select a tab on the client, in essence selecting the screen within a multi-screen screen set. The tab may be selected based on the tab's label (screen caption), the screen definition's name, the screen's position index, a position relative to the currently selected tab's position, or by specifying the first or last tab from left to right.

Structure

Contains:

- None

Contained By:

- `<script>`

Table 49. Attributes

Name	Description	Data Type	Default Value	Required
tab	This attribute can be set to specify the tab to select based on its position relative to the currently selected tab, the first or last tab, or by specifying a positive integer matching the screen's position index within the screen set. Valid values for this attribute include: • first - The first (or left-most) tab displayed. • last - The last (or right-most) tab displayed. • next - The next tab to the right of the currently selected tab. • previous - The previous tab to the right of the currently selected tab. • A positive integer matching the screen definitions position within the screen set.	String	None	No
name	This attribute can be set to the screen definition's name to specify the tab to be selected.	String	None	No
label	This attribute can be set to the label of the tab, which is the screen definitions caption attribute value, to be selected.	String	None	No
common script attributes	This element includes the following common script attributes: • timeout • sleep	N/A	N/A	N/A

Agentry Test Script: Window Elements Overview

The window-related elements within the Agentry Test Script are provided to allow for interaction with the windows presented by the mobile client application. This is, in essence, the same as interacting with the screen set itself.

The elements provided include those to close the window to either return to the previous window or to return to the window presenting the main screen set of the module, to check for expected values in the title bar of the currently displayed screen, and also those to support entering a signature value in a detail screen displaying a property of type Signature.

Note that the elements to close the current window should only be used if there is no navigational action available within the application to perform this operation. If such an action is present, it should be used just as if a user were executing that action.

<window-close>

The `<window-close>` element will close the current or specified window on the Client. Note that if an action button or some other control exists to close a window on the client that control should be used. If the specified window cannot be found a script error is thrown.

Structure
Contains:

- Text - The caption of the window to be closed.

Contained By:

- `<script>`

Table 50. Attributes

Name	Description	Data Type	Default Value	Required
common script attributes	This element includes the following common script attributes: • timeout • sleep	N/A	N/A	N/A

<window-close-main>

The `<window-close-main>` will close the test client running within the ATE. This will be the equivalent of selecting the **File \| Exit** menu item in the client’s menu bar.

Structure
Contains:

- None

Contained By:

- `<script>`

Table 51. Attributes

Name	Description	Data Type	Default Value	Required
common script attributes	This element includes the following common script attributes: • timeout • sleep	N/A	N/A	N/A

<window-expect>

The `<window-expect>` element will verify the caption of the currently displayed screen on the client. If the screen caption does not match the expected value a script error is thrown.

Structure

Contains:

- Text - The expected caption of the screen/window displayed on the client.

Contained By:

- `<script>`

Table 52. Attributes

Name	Description	Data Type	Default Value	Required
common script attributes	This element includes the following common script attributes: • window • timeout • sleep	N/A	N/A	N/A

<window-sign>

The `<window-sign>` element is provided to enter a signature in a signature capture field. This element will enter either a default signature, which is an X character, or may enter a more sophisticated image in the signature capture field. To specify the signature to enter other than the default signature, the `<window-sign>` element must contain two or more `<point>` elements, each of which specifies a point to draw a line in the signature capture field.

Structure
Contains:

- `<point>`

Contained By:

- `<script>`

Table 53. Attributes

Name	Description	Data Type	Default Value	Required
common script attributes	This element includes the following common script attributes: • window • control - for this element always set to `IDC_SIGNATURE_CAPTURE`. • timeout • sleep	N/A	N/A	N/A

<point>

The `<point>` element specifies a point within a signature capture control. The exact position is specified via two pixel coordinates, x and y. Two `<point>` elements contained within the same <window-signature> element will be connected. Subsequent `<point>` elements will continue to be connected, that is, the first and second `<point>` will be connected via a single line, then the second and third `<point>` elements will be connected, and so forth. If either the x or y coordinate of a point are outside the boundaries defined for the signature capture field a script error is thrown.

Structure
Contains:

- None

Contained By:

- `<window-sign>`

Table 54. Attributes

Name	Description	Data Type	Default Value	Required
x	The x, or horizontal coordinate for the point, specified in pixels, with the left-most pixel at position 0.	non-negative integer	none	Yes
y	The y, or vertical coordinate for the point, specified in pixels with the bottom-most pixel at position 0.	non-negative integer	none	Yes

Agentry Test Script: Client Elements Overview

The client-related elements of the Agentry Test Script are used to affect or interact with the client process itself. These elements can restart the client and retrieve and set registry key values.

Typically these elements are added manually to a test script, as the script recorded in the ATE does not support recording these values.

Those elements related to the registry should be used with caution and only by those with an understanding of the registry keys and values for the Agentry Test Environment and Client. If present, the script must be run by a user with permission to change Windows registry keys on the host system for the ATE.

<client-restart>

The `<client-restart>` element will restart the client process running within the Agentry Test Environment. As optional behavior for this element, using the `<registry>`, `<key>`, and `<value>` elements, one or more registry keys may be added, deleted, or modified during this restart. Note that this element will restart the test client running within the ATE. It will not restart the ATE itself.

Structure

Contains:

- `<registry>`

Contained By:

- `<script>`

Table 55. Attributes

Name	Description	Data Type	Default Value	Required
common script attributes	A set of related attributes common to most elements within the Agentry Test Script.	N/A	N/A	N/A

<registry>

The <registry> element can be used with the <restart-client> element to create or modify registry keys and values. This element requires the use of a <key> child element to specify the registry key to be affected.

Structure
Contains:

- `<key>`

Contained By:

- `<restart-client>`

Attributes:

- None

<key>

The `<key>` element specifies a registry key to be modified or created. Nested registry keys require nested `<key>` elements within the test script. If a specified key does not exist will be created. The reset attribute allows for the specified key to be deleted and recreated. If the `<key>` element contains a `<value>` element, the contents of the `<value>` will specify the value to which the named key should be set.

Structure
Contains:

- `<key>`
- `<value>`

Contained By:

- `<registry>`
- `<key>`

Table 56. Attributes

Name	Description	Data Type	Default Value	Required
name	This attribute specifies the name of the registry key to be affected by the `<key>` element. May also contain one of: • `HKLM` - HKEY_LOCAL_MACHINE • `HKCU` - HKEY_CURRENT_USER	string	none	Yes
reset	This attributes specifies whether the named key should be reset. The value `t` is treated as true and will result in the key being deleted and recreated. The value `f` is false.	string	`f`	No

<value>

The `<value>` element specifies the value to be set for the named registry key. The `<value>` element must be contained by a `<key>` element. The `<value>` element names the registry key that is to be modified within the one named by the containing `<key>` element. The `<value>` element can specify that the named key's value is to be added, modified, or deleted. When adding or modifying the value of a key, the `<value>` element also specifies the data type of the registry key value.

Structure

Contains:

- Text - contains the value to be set for the key. Ignored when the `reset` attribute is set to true.

Contained By:

- `<key>`

Table 57. Attributes

Name	Description	Data Type	Default Value	Required
name	The name of the registry key to be affected by the `<value>` element.	String	None	Yes

Name	Description	Data Type	Default Value	Required
type	Specifies the data type of the value expected for the registry key. Must be one of: • `binary` • `dword` • `string`	String	None	Yes - *ignored if reset is true.*
reset	Specifies whether or not the specified key should be deleted. The value `t` is true and will delete the value of named registry key. The value `f` is false.	String	`f`	No
common script attributes	A set of related attributes common to most elements within the Agentry Test Script.	N/A	N/A	N/A

Agentry Test Script: Client Host Elements overview

The client host-related elements provided in the Agentry Test Script are used to perform actions on the client host itself, outside the test client. This includes the ability to execute commands on the host system and to specify key's to be pressed on the client device.

While the test script runs only on the Agentry Test Environment, which can only be installed to a desktop, the interactions between the test client and host system can be scripted to mimic the behaviors of the target device type.

<command-line>

The `<command-line>` element allows for the execution of a command on the client device during test script playback. This element can specify how long to wait for the command to return and its expected exit code. The contents of the element are the commands to execute. Alternately a command may be specified using the `cmd` attribute of the element. These commands can be written to a temporary file that itself will then be executed by the `<command-line>` element. The default file extension for this file is `.bat`, though this can be overridden. If the specified command cannot be executed a script error is thrown.

Structure

Contains:

- Text - The command(s) to be executed by the element.

Contained By:

- `<script>`

Table 58. Attributes

Name	Description	Data Type	Default Value	Required
cmd	This attribute specifies the command to be executed by the element, or to pass parameters to a temporary file created by the element that will be executed. The first value in this parameter should be the token `%1` when used to pass parameters to a temporary script file. This should then be followed by the parameters, each separated by whitespace. The contents of the element should then be the command with the tokens `%1` through `%n` for each parameter specified in the `cmd` attribute.	string	None	No
ext	This attribute specifies the file extension for the temporary file to contain the commands executed by the element. The default file extension is `.bat`. The value of this attribute can optionally include a leading period (.).	string	`.bat`	No
waitFor	This attribute specifies the duration of time in milliseconds to wait for the command to complete execution. The value `-1` indicates to wait indefinitely. The value `0` indicates to execute the command and not wait for it to complete. The script will continue playback regardless of the result of executing the command. If the command does not exit before the specified duration elapses a script error is thrown.	integer	`0`	No
expectedExit	This attribute specifies the expected exit code to be returned by the processes executed by the command. This can be any non-negative integral number. This attribute is ignored if waitFor is set to `0`. If this attribute is not specified, the exit code of the command will not be checked. If the exit code differs from the value specified a script error is thrown.	non-negative integer.	None	No

Name	Description	Data Type	Default Value	Required
common script attributes	This element includes the following common script attributes: • timeout • sleep • frame	N/A	N/A	N/A

<key-press>

The `<key-press>` element will enter a single keystroke, mimicking that keystroke on a standard keyboard. This element includes the option of including key the combinations using the `Alt`, `Ctrl`, and/or `Shift` keys. Only a single character or keystroke can be set using the `<key-press>` element.

Structure

Contains:

- None

Contained By:

- `<script>`

Table 59. Attributes

Name	Description	Data Type	Default Value	Required
key	This attribute specifies the key stroke to enter. This may be one of the following values: • Any single printable character • “backspace” - The backspace key. • “tab” - The Tab key. • “enter” - The Enter/Return key. • “return” - The Enter/Return key. • “pause” - The Pause key. • “esc” - The Escape (Esc) key. • “space” - The spacebar. • “page up” - The Page Up key. • “page down” - The Page Down key. • “end” - The End key. • “home” - The Home key. • “left” - The left arrow key. • “right” - The right arrow key. • “up” - The up arrow key. • “down” - The down arrow key. • “print screen” - The Print Screen key. • “insert” - The Insert key. • “delete” The Delete key. • “F*n*” - *n* is any numeric value from 1 to 24. That function key is then pressed.	character	None	Yes
shift	This attribute specifies whether to depress the `Shift` key in combination with the key attribute. The value `t` is true and will depress the `Shift` key. The value `f` is false.	Boolean	`f`	No
ctrl	This attribute specifies whether to depress the `Ctrl` key in combination with the key attribute. The value `t` is true and will depress the `Ctrl` key. The value `f` is false.	Boolean	`f`	No

Name	Description	Data Type	Default Value	Required
alt	This attribute specifies whether to depress the `Alt` key in combination with the key attribute. The value `t` is true and will depress the `Alt` key. The value f is false.	Boolean	`f`	No
common script attributes	A set of related attributes common to most elements within the Agentry Test Script.	N/A	N/A	N/A

Agentry Java API

The Agentry Java API is provided for Agentry applications that make use of a Java system connection for data synchronization. This API exposes the mobile application data to the Java logic, system and user information, and other key data needed during synchronization.

com.syclo.agentry package

utility package

java_logging package

AgentryHandler class
This is an implementation of the Java Logging API's Handler class that will route log messages to the Agentry Server Java System Connection's log file.

Syntax
`public class AgentryHandler extends Handler`

Members
All members of AgentryHandler, including inherited members. **Variables**

Modifier and Type	Variable	Description
protected Formatter	*_defaultFormatter* on page 376	This is the default formatter used by this handler.

Constructors

Modifier and Type	Constructor	Description
public	*AgentryHandler()* on page 375	Constructs a new AgentryHandler object.

Methods

Modifier and Type	Method	Description
public void	*close()* on page 376	Agentry log files cannot be closed from Java; this method does nothing.
public void	*flush()* on page 376	The Agentry server always flushes its log files; this method does not do anything explicit.
protected LogLevel	*mapLogLevel(Level)* on page 376	Maps a Java log level to an Agentry log level, as described in the class documentation.
public void	*publish(LogRecord)* on page 376	Publishes a log record to the Agentry server.

Usage

It is also capable of routing messages to an Agentry user log file, if it receives objects of class UserLogRecord instead of objects of the base class `LogRecord`. `UserLogRecord` objects can be easily generated by using the UserLogger class.

Both the Java Logging API and the Agentry Server support the concept of log levels. The logging level of the Agentry Server is configured in the `AgentryLogging.ini` file, and the Agentry Server supports six levels, 0 through 5, with level 0 messages always being logged. For each system connection, there is a level setting in the configuration file, and all messages at or below that setting will be logged.

The levels in the Java Logging API map to the log levels in Agentry as follows:

Java Logging API	Agentry	Purpose
Level#SEVERE	Level 0	Severe errors, always logged
Level#WARNING	Level 1	Warnings or recoverable errors
Level#INFO	Level 2	"Now doing this" type messages

Java Logging API	Agentry	Purpose
Level#CONFIG	Level 2	Configuration settings and related messages. Note that this will map to the same Agentry level as `Level.INFO` messages.
Level#FINE	Level 3	Details of why a decision is being made. This is the level that the Server.debug and User.debug methods will log at.
Level#FINER	Level 4	Values being set from files, user input, etc; details of calculations.
Level#FINEST	Level 5	Really low level details. This is the level that the AJAPI classes and the Agentry Server will log at, so if you enable this you may get a lot of log messages. Because of this, using this level in end-user code is discouraged.

Configuration:

By default each `AgentryHandler` is initialized using the following LogManager configuration properties:

- `com.syclo.agentry.utility.java_logging.AgentryHandler.level`
- Specifies the default level (defaults to Level#ALL).
- `com.syclo.agentry.utility.java_logging.AgentryHandler.filter`
- Specifies the name of a Filter class (defaults to no filter).
- `com.syclo.agentry.utility.java_logging.AgentryHandler.formatter`
- Specifies the name of a Formatter class to use (defaults to an internal formatter that produces just the text of the message by itself; Agentry will add a timestamp to the message when it logs it).

AgentryHandler() constructor

Constructs a new AgentryHandler object.

Syntax

```
public   AgentryHandler()
```

close() method

Agentry log files cannot be closed from Java; this method does nothing.

Syntax

```
public void close()
```

flush() method

The Agentry server always flushes its log files; this method does not do anything explicit.

Syntax

```
public void flush()
```

mapLogLevel(Level) method

Maps a Java log level to an Agentry log level, as described in the class documentation.

Syntax

`protected LogLevel mapLogLevel(Level` *javaLevel* `)`

Parameters

- **javaLevel –** the Java log level

Returns

An LogLevel constant

Usage

If you have custom log levels, you can subclass this class and override this method to map your custom log levels to Agentry's levels. Otherwise, this method will make a best-guess attempt at mapping custom log levels, based on which of Java's levels the custom levels fall between (for example, if your custom level's integer value is greater than `Level.INFO` but less than `Level.WARNING`, it will be treated as equivalent to `Level.INFO`).

publish(LogRecord) method

Publishes a log record to the Agentry server.

Syntax

`public void publish(LogRecord` *record* `)`

_defaultFormatter variable

This is the default formatter used by this handler.

Syntax

```
protected Formatter _defaultFormatter
```

Usage

It just does localization and parameter substitution, and otherwise returns the message without any other adornment.

AgentryJavaLoggingConfigurator class

This class is used by the Java System Connection to set up a default configuration for the Java Logging API.

Syntax

```
public class  AgentryJavaLoggingConfigurator
```

Members

All members of AgentryJavaLoggingConfigurator, including inherited members. **Nested classes**

Modifier and Type	Class	Description
class	*ReallySimpleFormatter* on page 377	This is a basic formatter class that is used for the AgentryJavaCompiler log file handler (to mimic the format that com.syclo.agentry.utility.Logger was using).

Constructors

Modifier and Type	Constructor	Description
public	*AgentryJavaLoggingConfigurator()* on page 378	Configure the Java Logging API via the properties file at com/syclo/agentry/internal/logging.properties.

Usage

The default configuration will set up a single root logger using the AgentryHandler handler, which will result in all log messages being routed to the Java System Connection's log file on the Agentry Server.

AgentryJavaLoggingConfigurator.ReallySimpleFormatter class

This is a basic formatter class that is used for the AgentryJavaCompiler log file handler (to mimic the format that com.syclo.agentry.utility.Logger was using).

Syntax

```
class  ReallySimpleFormatter extends Formatter
```

Members

All members of ReallySimpleFormatter, including inherited members. **Methods**

Modifier and Type	Method	Description
public String	*format(LogRecord)* on page 378	

Usage

It just prefixes the log message with a timestamp.

format(LogRecord) method

Syntax

`public String format ( LogRecord` *record* `)`

AgentryJavaLoggingConfigurator() constructor

Configure the Java Logging API via the properties file at com/syclo/agentry/internal/logging.properties.

Syntax

`public AgentryJavaLoggingConfigurator ()` throws `IOException`

Exceptions

- **IOException –** if the logging properties file cannot be loaded.

Usage

In addition, add special file handlers for the various classes in `com.syclo.agentry.internal`, to mimic what those classes used to do with the deprecated `com.syclo.agentry.utility.Logger` class.

UserLogRecord class

This class is used to create a log record that, if received by an instance of AgentryHandler, will be routed to a user-specific log file on the Agentry server.

Syntax

`public class UserLogRecord extends LogRecord`

Members

All members of UserLogRecord, including inherited members. **Constructors**

Modifier and Type	Constructor	Description
public	*UserLogRecord(User, Level, String)* on page 381	Constructs a new UserLogRecord object.
public	*UserLogRecord(User, LogRecord)* on page 381	Constructs a new UserLogRecord object that copies all of its information from an existing LogRecord object.

Methods

Modifier and Type	Method	Description
public boolean	*equals(Object)* on page 381	
public Level	*getLevel()* on page 381	
public String	*getLoggerName()* on page 381	
public String	*getMessage()* on page 382	
public long	*getMillis()* on page 382	
public Object[]	*getParameters()* on page 382	
public ResourceBundle	*getResourceBundle()* on page 382	
public String	*getResourceBundleName()* on page 382	
public long	*getSequenceNumber()* on page 382	
public String	*getSourceClassName()* on page 382	
public String	*getSourceMethodName()* on page 382	
public int	*getThreadID()* on page 383	
public Throwable	*getThrown()* on page 383	
public User	*getUser()* on page 383	Returns the user that this log record pertains to.
public int	*hashCode()* on page 383	
public void	*setLevel(Level)* on page 383	

Modifier and Type	Method	Description
public void	*setLoggerName(String)* on page 383	
public void	*setMessage(String)* on page 383	
public void	*setMillis(long)* on page 384	
public void	*setParameters(Object[])* on page 384	
public void	*setResourceBundle(ResourceBundle)* on page 384	
public void	*setResourceBundleName(String)* on page 384	
public void	*setSequenceNumber(long)* on page 384	
public void	*setSourceClassName(String)* on page 384	
public void	*setSourceMethodName(String)* on page 384	
public void	*setThreadID(int)* on page 384	
public void	*setThrown(Throwable)* on page 385	
public String	*toString()* on page 385	

Usage

You can create instances of this class directly and hand them off to `Logger` objects to log user-specific messages, but it is far more convenient to use the UserLogger class, which is a subclass of `Logger` that handles converting `LogRecord` objects to `UserLogRecord` objects for you.

Note that it is perfectly acceptable for one of these objects to find its way to any other type of `Handler` object; other handlers will log these objects the same way they would log any other `LogRecord` object.

UserLogRecord(User, Level, String) constructor
Constructs a new UserLogRecord object.

Syntax
```
public  UserLogRecord ( User user ,  Level level ,  String msg )
```

Parameters

- **user –** The user to log the message for
- **level –** The message level
- **msg –** The log message

UserLogRecord(User, LogRecord) constructor
Constructs a new UserLogRecord object that copies all of its information from an existing LogRecord object.

Syntax
```
public  UserLogRecord ( User user ,  LogRecord record )
```

Parameters

- **user –** The user to log the message for
- **record –** The `LogRecord` object containing the original log message information.

Usage
This will copy all of the information from the existing log record, including message parameters.

equals(Object) method

Syntax
```
public boolean equals ( Object obj )
```

getLevel() method

Syntax
```
public Level getLevel ()
```

getLoggerName() method

Syntax
```
public String getLoggerName ()
```

getMessage() method

Syntax

```
public String getMessage()
```

getMillis() method

Syntax

```
public long getMillis()
```

getParameters() method

Syntax

```
public Object[] getParameters()
```

getResourceBundle() method

Syntax

```
public ResourceBundle getResourceBundle()
```

getResourceBundleName() method

Syntax

```
public String getResourceBundleName()
```

getSequenceNumber() method

Syntax

```
public long getSequenceNumber()
```

getSourceClassName() method

Syntax

```
public String getSourceClassName()
```

getSourceMethodName() method

Syntax

```
public String getSourceMethodName()
```

getThreadID() method

Syntax
```
public int getThreadID ()
```

getThrown() method

Syntax
```
public Throwable getThrown ()
```

getUser() method
Returns the user that this log record pertains to.

Syntax
```
public User getUser ()
```

Returns
the `User` object.

hashCode() method

Syntax
```
public int hashCode ()
```

setLevel(Level) method

Syntax
`public void setLevel ( Level` *level* `)`

setLoggerName(String) method

Syntax
`public void setLoggerName ( String` *name* `)`

setMessage(String) method

Syntax
`public void setMessage ( String` *message* `)`

setMillis(long) method

Syntax

`public void setMillis ( long` *millis* `)`

setParameters(Object[]) method

Syntax

`public void setParameters ( Object[]` *parameters* `)`

setResourceBundle(ResourceBundle) method

Syntax

`public void setResourceBundle ( ResourceBundle` *bundle* `)`

setResourceBundleName(String) method

Syntax

`public void setResourceBundleName ( String` *name* `)`

setSequenceNumber(long) method

Syntax

`public void setSequenceNumber ( long` *seq* `)`

setSourceClassName(String) method

Syntax

`public void setSourceClassName ( String` *sourceClassName* `)`

setSourceMethodName(String) method

Syntax

`public void setSourceMethodName ( String` *sourceMethodName* `)`

setThreadID(int) method

Syntax

`public void setThreadID ( int` *threadID* `)`

setThrown(Throwable) method

Syntax

`public void setThrown ( Throwable` *thrown* `)`

toString() method

Syntax

```
public String toString ()
```

UserLogger class

This class is used in combination with AgentryHandler to route messages to the user-specific log files on the Agentry Server.

Syntax

```
public class UserLogger extends Logger
```

Members

All members of UserLogger, including inherited members. **Methods**

Modifier and Type	Method	Description
public User	*getUser()* on page 386	Returns the user that this logger is logging for.
public static UserLogger	*getUserLogger(String, User)* on page 386	Returns an instance of UserLogger that uses the named Logger instance to log messages to the Agentry server for the given user.
public static UserLogger	*getUserLogger(String, String, User)* on page 386	Returns an instance of UserLogger that uses the named Logger instance to log messages to the Agentry server for the given user.
public void	*log(LogRecord)* on page 387	

Usage

It acts as a child logger of an existing `Logger` object. When its log methods are called, it creates log records of the class UserLogRecord instead of `LogRecord`, and then passes those log records to its parent logger. If these log records eventually find their way to an `AgentryHandler` instance, that instance will know to route the messages to user-specific logs on the Agentry server, if those logs are enabled via the `AgentryLogging.ini` file.

getUser() method

Returns the user that this logger is logging for.

Syntax

`public User getUser()`

Returns

the `User` object.

getUserLogger(String, User) method

Returns an instance of UserLogger that uses the named Logger instance to log messages to the Agentry server for the given user.

Syntax

`public static UserLogger getUserLogger(String` *name* `, User` *user* `)`

Parameters

- **name –** The logger name. Note that the same `Logger` object may be used by multiple `UserLogger` objects to log messages for multiple users.
- **user –** The user to log messages for.

Returns

A new `UserLogger` object, or `null` if `Logger.getLogger(String)` would have returned `null` for the same logger name.

getUserLogger(String, String, User) method

Returns an instance of UserLogger that uses the named Logger instance to log messages to the Agentry server for the given user.

Syntax

`public static UserLogger getUserLogger(String` *name* `, String` *resourceBundleName* `, User` *user* `)`

Parameters

- **name –** The logger name. Note that the same `Logger` object may be used by multiple `UserLogger` objects to log messages for multiple users.
- **resourceBundleName –** The name of a resource bundle to use for localizing messages for this logger. May be `null` if no localization is necessary.
- **user –** The user to log messages for.

Returns
A new `UserLogger` object, or `null` if `Logger.getLogger(String)` would have returned `null` for the same logger name.

log(LogRecord) method

Syntax
`public void log ( LogRecord` *record* `)`

log4j package

AgentryAppender class
AgentryAppender is a Log4j Appender that will route log messages to the Agentry Server Java System Connection's log file.

Syntax
`public class AgentryAppender extends AppenderSkeleton`

Members
All members of AgentryAppender, including inherited members. **Variables**

Modifier and Type	Variable	Description
public static final String	*AGENTRY_USER_MDC_KEY* on page 390	MDC key for adding an Agentry User object to the Log4j MDC.

Constructors

Modifier and Type	Constructor	Description
public	*AgentryAppender()* on page 389	Constructs a new AgentryAppender object.

Methods

Modifier and Type	Method	Description
protected void	*append(LoggingEvent)* on page 389	
public void	*close()* on page 389	
protected LogLevel	*mapLogLevel(Level)* on page 389	Maps a Log4j log level to an Agentry log level, as described in the class documentation.

Modifier and Type	Method	Description
public boolean	*requiresLayout()* on page 389	

Usage

It is also capable of routing messages to an Agentry user log file, if the key named by AGENTRY_USER_MDC_KEY is set to an Agentry `User` object in Log4j's `MDC`.

Both Log4j and the Agentry Server support the concept of log levels. The logging level of the Agentry Server is configured in the `AgentryLogging.ini` file, and the Agentry Server supports six levels, 0 through 5, with level 0 messages always being logged. For each system connection, there is a level setting in the configuration file, and all messages at or below that setting will be logged.

The levels in Log4j map to the log levels in Agentry as follows:

Log4j	Agentry	Purpose
`Level.FATAL, Level.ERROR`	Level 0	Severe errors, always logged
`Level.WARN`	Level 1	Warnings or recoverable errors
`Level.INFO`	Level 2	"Now doing this" type messages
`Level.DEBUG`	Level 3	Details of why a decision is being made.
`Level.TRACE`	Level 4	Values being set from files, user input, etc; details of calculations

Note that there is currently no Log4j level that corresponds to log level 5 in Agentry. That log level is generally used by very low-level logging in the Agentry Server itself, so its use in end-user applications would be discouraged anyways.

`AgentryAppender` supports the basic settings used by other Log4j appenders, including layouts. It is not required to have a layout configured for the appender; if no layout is configured, then messages will be sent directly to Agentry with no special formatting.

NOTE: If you want to use this class you must have Log4j on your Agentry server's class path, as configured in `Agentry.ini`. Log4j is not distributed with the Agentry Server. It can be obtained from the *Apache Log4j* web site. This appender is intended for use with Log4j 1.2.x; it may work with later versions, but there is no guarantee.

AgentryAppender() constructor
Constructs a new AgentryAppender object.

Syntax
```
public   AgentryAppender ()
```

append(LoggingEvent) method

Syntax
`protected  void  append ( LoggingEvent` *event* `)`

close() method

Syntax
```
public  void  close ()
```

mapLogLevel(Level) method
Maps a Log4j log level to an Agentry log level, as described in the class documentation.

Syntax
`protected  LogLevel  mapLogLevel ( Level` *log4jLevel* `)`

Parameters

- **log4jLevel –** the Java log level

Returns
An LogLevel constant

Usage
If you have custom log levels, you can subclass this class and override this method to map your custom log levels to Agentry's levels. Otherwise, this method will make a best-guess attempt at mapping custom log levels, based on which of Java's levels the custom levels fall between (for example, if your custom level's integer value is greater than `Level.INFO` but less than `Level.WARNING`, it will be treated as equivalent to `Level.INFO`).

requiresLayout() method

Syntax
```
public  boolean  requiresLayout ()
```

AGENTRY_USER_MDC_KEY variable

MDC key for adding an Agentry User object to the Log4j MDC.

Syntax

```
public static  final String  AGENTRY_USER_MDC_KEY
```

Usage

If a user is set in the `MDC` when a message is logged, then that message will be routed to the user's log file in the Agentry Server, instead of to the Java System Connection's log file.

DataTableMapIterator< K, V > class

This is a helper class that makes it easy to return Map objects from the iterator method of a DataTable.

Syntax

```
public class  DataTableMapIterator< K,  V > extends
java::util::Iterator< DataTableObject >
```

Members

All members of DataTableMapIterator< K, V >, including inherited members. **Methods**

Modifier and Type	Method	Description
public	*DataTableMapIterator(Map< K, V >)* on page 390	Constructs a new DataTableMapIterator object.
public boolean	*hasNext()* on page 391	
public DataTableObject	*next()* on page 391	Returns a new DataTableObject that contains the key and value of the next entry in the map.
public void	*remove()* on page 391	

Usage

It iterates over the contents of the map and returns each map entry as a DataTableObject.

DataTableMapIterator(Map< K, V >) method

Constructs a new DataTableMapIterator object.

Syntax

```
public   DataTableMapIterator ( Map< K,  V > map )
```

Parameters

- **map –** The Map to iterate over.

hasNext() method

Syntax

```
public boolean hasNext()
```

next() method

Returns a new DataTableObject that contains the key and value of the next entry in the map.

Syntax

```
public DataTableObject next()
```

remove() method

Syntax

```
public void remove()
```

Logger class

Deprecated. Use java.util.logging or log4j. This class implements basic log file functionality, if you wish to log to another file other than Agentry's server or user log files.

Syntax

```
public class Logger
```

Members

All members of Logger, including inherited members. **Constructors**

Modifier and Type	Constructor	Description
public	*Logger(String, boolean)* on page 392	Create a new log file.

Methods

Modifier and Type	Method	Description
public void	*appendDebug(String)* on page 392	Append a message to the active debug buffer.
public void	*beginDebug(String)* on page 393	Opens up a multiline debug buffer and adds the given message to it.

Modifier and Type	Method	Description
public synchronized void	*debug(String)* on page 393	Write a message to the log file.
public synchronized void	*debug(String, Map< String, String >, String)* on page 393	Write the contents of a Map object to the log file.
public void	*endDebug(String)* on page 393	Close the debug buffer and write it to the log file.
public boolean	*isDebugMode()* on page 394	Returns whether a debug buffer has been opened by beginDebug and is still open.

Usage

If you need more powerful logging than what this class provides, you should look at the logging functionality that was added in JDK 1.4, or use Apache log4j or something like it.

Logger(String, boolean) constructor

Create a new log file.

Syntax

`public  Logger ( String` *fileName* `,  boolean` *overwrite* `)` throws `IOException`

Parameters

- **fileName –** The log file name
- **overwrite –** `true` means to overwrite any existing file, `false` means append to any existing file.

Exceptions

- **IOException –** if an error occurs.

appendDebug(String) method

Append a message to the active debug buffer.

Syntax

`public  void  appendDebug ( String` *message* `)`

Parameters

- **message –** The message

beginDebug(String) method

Opens up a multiline debug buffer and adds the given message to it.

Syntax

`public void beginDebug ( String` *message* `)`

Parameters

- **message –** The debugging message

Usage

Additional messages can be added by calling appendDebug. The contents of the buffer will not be written to the log file until endDebug is called.

debug(String) method

Write a message to the log file.

Syntax

`public synchronized void debug ( String` *message* `)`

Parameters

- **message –** The message

debug(String, Map< String, String >, String) method

Write the contents of a Map object to the log file.

Syntax

`public synchronized void debug ( String` *header* `, Map< String, String >` *messages* `, String` *footer* `)`

Parameters

- **header –** The header message
- **messages –** The map to dump
- **footer –** The footer message

endDebug(String) method

Close the debug buffer and write it to the log file.

Syntax

`public void endDebug ( String` *message* `)`

Parameters

- **message –** The final debug message

isDebugMode() method
Returns whether a debug buffer has been opened by beginDebug and is still open.

Syntax
```
public boolean isDebugMode ()
```

Returns
`true` if a debug buffer is active.

AgentryException class
Base class for checked exceptions that can be thrown from various methods in the AJAPI.

Syntax
```
public class AgentryException extends Exception
```

Derived classes

- *BusinessLogicException* on page 397
- *FatalTransactionException* on page 420
- *FatalTransactionExceptionStop* on page 423
- *LoginException* on page 432
- *RetryTransactionException* on page 461
- *RetryTransactionWithChangeException* on page 463
- *StepletAbortException* on page 512
- *StepletStopException* on page 513

Members
All members of AgentryException, including inherited members. **Constructors**

Modifier and Type	Constructor	Description
public	*AgentryException(String)* on page 395	Constructs a new AgentryException object.
public	*AgentryException(String, Throwable)* on page 395	Constructs a new AgentryException object.

Modifier and Type	Constructor	Description
public	*AgentryException(String, String, String, Throwable)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.
public	*AgentryException(String, String, String)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.

Methods

Modifier and Type	Method	Description
public final String	*getNotificationText()* on page 397	Returns the notification window text.
public final String	*getNotificationTitle()* on page 397	Returns the notification window title.
public final String	*getOkButtonLabel()* on page 397	Returns the notification window button label.

Usage

Throwing this base class from a method will generally result in an error being logged in the server event log, and the client getting a "Server error - please contact your administrator (13)" message.

AgentryException(String) constructor

Constructs a new AgentryException object.

Syntax

`public   AgentryException ( String` *message* `)`

Parameters

- **message –** The error message.

AgentryException(String, Throwable) constructor

Constructs a new AgentryException object.

Syntax

`public   AgentryException ( String` *message* `,  Throwable` *cause* `)`

Parameters

- **message –** The error message.
- **cause –** The causing exception.

<u>*AgentryException(String, String, String, Throwable) constructor*</u>

Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.

Syntax

`public AgentryException(String` *title*`, String` *text*`, String` *okLabel*`, Throwable` *cause*`)`

Parameters

- **title –** The window title.
- **text –** The window text. This will also be used as the message text of the exception when it is logged (it will be returned when getMessage() is called).
- **okLabel –** The window button label.
- **cause –** The causing exception

Usage

For now this is pretty much supported only by steplets that are being used as part of Agentry transactions.

<u>*AgentryException(String, String, String) constructor*</u>

Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.

Syntax

`public AgentryException(String` *title*`, String` *text*`, String` *okLabel*`)`

Parameters

- **title –** The window title.
- **text –** The window text. This will also be used as the message text of the exception when it is logged (it will be returned when getMessage() is called).
- **okLabel –** The window button label.

Usage

For now this is pretty much supported only by steplets that are being used as part of Agentry transactions.

getNotificationText() method
Returns the notification window text.

Syntax
```
public  final String  getNotificationText ()
```

Returns
the text

getNotificationTitle() method
Returns the notification window title.

Syntax
```
public  final String  getNotificationTitle ()
```

Returns
the title

getOkButtonLabel() method
Returns the notification window button label.

Syntax
```
public  final String  getOkButtonLabel ()
```

Returns
the label

BusinessLogicException class
This exception represents an error condition that should be reported to the client.

Syntax
```
public class  BusinessLogicException extends AgentryException
```

Members
All members of BusinessLogicException, including inherited members. **Constructors**

Modifier and Type	Constructor	Description
public	*BusinessLogicException(String)* on page 398	Construct a new exception.
public	*BusinessLogicException(String, Throwable)* on page 399	Constructs a new BusinessLogicException object.

Inherited members from AgentryException

Modifier and Type	Member	Description
public	*AgentryException(String)* on page 395	Constructs a new AgentryException object.
public	*AgentryException(String, Throwable)* on page 395	Constructs a new AgentryException object.
public	*AgentryException(String, String, String, Throwable)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.
public	*AgentryException(String, String, String)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.
public final String	*getNotificationText()* on page 397	Returns the notification window text.
public final String	*getNotificationTitle()* on page 397	Returns the notification window title.
public final String	*getOkButtonLabel()* on page 397	Returns the notification window button label.

Usage

It can be thrown from any of the various client-related session methods, such as the methods on `Steplet`, `DataTable`, and `ComplexTable`. The error message in this exception will be displayed in the client's transmit window; in addition, if the `printBusinessLogicStackTrace` configuration option is enabled in the `agentry.ini` file for the Java system connection, then a full stack trace for the exception will also be displayed in the client's transmit window.

BusinessLogicException(String) constructor

Construct a new exception.

Syntax

```
public   BusinessLogicException ( String message )
```

Parameters

- **message –** The error message

BusinessLogicException(String, Throwable) constructor

Constructs a new BusinessLogicException object.

Syntax

```
public   BusinessLogicException ( String message ,  Throwable
cause )
```

Parameters

- **message –** The error message
- **cause –** The causing exception

ComplexTableSession class

The ComplexTableSession class encapsulates the processing involved in a complex table synchronization within an Agentry-based application.

Syntax

```
public class  ComplexTableSession extends Session
```

Members

All members of ComplexTableSession, including inherited members. **Constructors**

Modifier and Type	Constructor	Description
public	*ComplexTableSession(String, Server, SessionData, User)* on page 400	Construct a new session.

Inherited members from Session

Modifier and Type	Member	Description
public final void	*debug(String)* on page 503	Write the given message to a debug log, if debugging is enabled.
public String	*getName()* on page 503	Returns the name of the session, as configured in the Agentry application.
public Server	*getServer()* on page 503	Returns the Server singleton object that the Java system connection is currently using.
public SessionData	*getSessionData()* on page 504	Returns the session data for this session.

Modifier and Type	Member	Description
public User	*getUser()* on page 504	Returns the user for this session, if any.
protected	*Session(String, Server, SessionData, User)* on page 502	Construct a new session, and save each of the arguments so that they can be retrieved later via the getName(), getServer(), getSessionData(), and getUser() methods.
protected	*Session(String, Server, SessionData)* on page 502	Construct a new session, and save each of the arguments so that they can be retrieved later via the getName(), getServer(), and getSessionData() methods.
public void	*sessionAborted()* on page 504	This is called if the session is aborted (e.g., by an exception).

Usage

This class contains the session data for the complex table synchronization, as well the User object for the user performing the synchronization, and a reference to the Server singleton object.

ComplexTableSession(String, Server, SessionData, User) constructor

Construct a new session.

Syntax

```
public  ComplexTableSession ( String tableName ,  Server server ,
SessionData sessionData ,  User user )
```

Parameters

- **tableName –** The complex table name, as configured in the Agentry application.
- **server –** The `Server` object that the Java System connection was configured to use.
- **sessionData –** Session data for this transmit.
- **user –** The client user performing the transmit.

Usage

This constructor is called by the Server.createComplexTableSession method. Subclasses should implement a constructor with the same signature as this one.

ComplexTable< CTOBJ > class

The ComplexTable class encapsulates the data retrieval for a complex table within an Agentry-based application.

Syntax

```
public abstract class  ComplexTable< CTOBJ > extends
AgentryJavaBackEndManagedObject
```

Members

All members of ComplexTable< CTOBJ >, including inherited members. **Variables**

Modifier and Type	Variable	Description
protected SycloCalendar	*_clientLastDataUpdateTime* on page 411	Storage for the client's last update time that was passed into the constructor.
protected boolean	*_rebuilding* on page 412	This member will be set by the Agentry Server to whatever the willRebuildTable() method returns.
protected ComplexTableSession	*_session* on page 412	Storage for the session that was passed into the constructor.

Methods

Modifier and Type	Method	Description
public void	*build()* on page 404	This method is called by the Agentry Server after it calls willRebuildTable(), but before it calls dataIterator() or deleteIterator().
final boolean	*checkForReload()* on page 404	This method is called by the Agentry Server to indicate whether a reload should be done.
public	*ComplexTable(ComplexTableSession, GregorianCalendar)* on page 405	Constructs a new ComplexTable object.

Modifier and Type	Method	Description
public abstract Iterator< CTOBJ >	*dataIterator()* on page 405	This method should return either a partial or complete list of the records for the complex table, depending on the result of calling willRebuildTable().
public Iterator< CTOBJ >	*deleteIterator()* on page 406	This method should return a list of the records that have been deleted from the complex table since the client's last update time, which can be retrieved via getClientLastDataUpdateTime()).
public SycloCalendar	*getClientLastDataUpdateTime()* on page 407	Returns the time when the client last retrieved data for this complex table.
public GregorianCalendar	*getNewDataUpdateTime()* on page 407	Returns the new "last data update time", as set by a prior call to setNewDataUpdateTime.
public ComplexTableSession	*getSession()* on page 407	Return the session for this complex table transmission.
public final void	*initialize()* on page 407	Deprecated. Subclass overrides of this method should be renamed to build(). This method has been renamed to build().
public boolean	*isRebuilding()* on page 408	This method returns the previously cached result of the Agentry Server's call to the willRebuildTable() method.
final int	*lastUpdateDate()* on page 408	Returns the day-of-month component of the complex table data's last update time.
final int	*lastUpdateHours()* on page 408	Returns the hour component of the complex table data's last update time.
final int	*lastUpdateMinutes()* on page 408	Returns the minute component of the complex table data's last update time.

Modifier and Type	Method	Description
final int	*lastUpdateMonth()* on page 409	Returns the month component of the complex table data's last update time.
final int	*lastUpdateSeconds()* on page 409	Returns the seconds component of the complex table data's last update time.
final int	*lastUpdateYear()* on page 409	Returns the year component of the complex table data's last update time.
public final boolean	*reload()* on page 410	Deprecated. Subclass overrides of this method should be renamed to willRebuildTable(). This method has been renamed to willRebuildTable().
public void	*setNewDataUpdateTime(GregorianCalendar)* on page 410	This method sets the new "last data update time" for the complex table.
public boolean	*willRebuildTable()* on page 411	This method is called by the Agentry server to determine whether the server should rebuild the client's complex table from scratch.

Usage

When implementing a complex table using a Java Interface system connection, this class will always be extended.

Complex tables in Agentry support both full and incremental updates. It is up to an implementing subclass to determine which mechanism to use, based on whether the underlying data source supports time-stamped data. If it does, an implementation can return a "last update" timestamp along with the complex table data. The next time a client requests an update of the table, it will pass back that timestamp, and this class can then return only the data changes (additions, updates, and deletions) that have occurred since that time.

Data objects are returned from this class to the Agentry server via iterators. The data objects themselves are structured in the same manner as the data objects for a Steplet, in that their data is stored in public member fields that are accessed directly by Agentry, and are mapped to corresponding fields in the Complex Table when the table is defined in the Agentry Editor.

The synchronization process for a complex table involves determining first which records have been deleted from the complex table and retrieving those records' unique identifiers. Then, the records that have been added or updated since the last client update are retrieved. The Agentry Server will call the methods of this class in the following sequence:

1. Constructor - passes in the client's last update date
2. willRebuildTable() - tells Agentry whether to expect a full or incremental update
3. build() - can be used to gather records
4. dataIterator() - returns new or updated records
5. `lastUpdate*` methods - returns the new data timestamp (set via a call to setNewDataUpdateTime)
6. deleteIterator() - returns keys for deleted records; may not be called if `willRebuildTable()` returned true
7. `lastUpdate*` methods (again - should return the same value as the previous call); may not be called if `willRebuildTable()` returned true

Note that for each client transmit, a new instance of this class is created by the Agentry server.

build() method

This method is called by the Agentry Server after it calls willRebuildTable(), but before it calls dataIterator() or deleteIterator().

Syntax

`public void build()` throws `AgentryException`

Exceptions

- **AgentryException class –** if an error occurs.

Usage

It can be used to build up the data sets for the complex table, if it is more convenient to query both the updates and the deletes in a single place rather than querying them separately in the iterator methods.

checkForReload() method

This method is called by the Agentry Server to indicate whether a reload should be done.

Syntax

`final boolean checkForReload()` throws `AgentryException`

Returns

`true` if the table should be reloaded

Exceptions

- **AgentryException class –** if an error occurs

Usage
It calls willRebuildTable() and saves the result into the _rebuilding member variable.

ComplexTable(ComplexTableSession, GregorianCalendar) method
Constructs a new ComplexTable object.

Syntax
`public ComplexTable ( ComplexTableSession` *session* `,`
`GregorianCalendar` *clientLastDataUpdate* `)`

Parameters

- **session –** A ComplexTableSession object that provides the access to the session data pertinent to the complex table application component.
- **clientLastDataUpdate –** This argument contains the date and time of the client's last data update.

Usage
Subclasses need to implement the same constructor and pass the parameters back to this constructor via `super()`.

This constructor will set the return value of the default implementation of the willRebuildTable() method to `true` if the `clientLastDataUpdate` parameter contains the invalid date/time value, or to `false` if not.

dataIterator() method
This method should return either a partial or complete list of the records for the complex table, depending on the result of calling willRebuildTable().

Syntax
`public abstract Iterator< CTOBJ > dataIterator ()` throws
`AgentryException`

Returns
an Iterator object that will iterate over the records of the complex table.

Exceptions

- **AgentryException class –** if an error occurs.

Usage

It returns these records in the form of a Iterator object that iterates over a list of objects. Subclasses should override this method to return the desired data.

If willRebuildTable() returns `false` (indicating that the client will receive an incremental update), then this method should only return those records that have been added to the complex table since the client's last update time (which can be retrieved via getClientLastDataUpdateTime()). If willRebuildTable() returns `true` (indicating that the client should rebuild its complex table from scratch), then this method should return the complete set of complex table records.

deleteIterator() method

This method should return a list of the records that have been deleted from the complex table since the client's last update time, which can be retrieved via getClientLastDataUpdateTime()).

Syntax

```
public  Iterator< CTOBJ >  deleteIterator() throws
AgentryException
```

Returns

a Iterator object that will iterate over the records that have been deleted from the complex table since the client's last update.

Exceptions

- **AgentryException class –** if an error occurs.

Usage

It returns these records in the form of a Iterator object that iterates over a list of objects that identify the deleted records via their unique keys. Subclasses should override this method to return the desired data. Note that the objects returned do not need to be fully populated, they only need to contain their unique identifiers (which are configured in the Agentry application).

The actual behavior of this method is dependent on the result of the willRebuildTable() method. If that method returns `false`, then this method should return a list of deleted records. If that method returns `true`, then this method should return an empty iterator. The default behavior of this method is to return an empty iterator, so if your willRebuildTable() implementation will always return `true`, then you don't need to override this method.

Note that in some versions of Agentry, this method may not be called at all if `willRebuildTable()` returns true, since there should be no deleted objects in that case.

getClientLastDataUpdateTime() method

Returns the time when the client last retrieved data for this complex table.

Syntax

`public SycloCalendar getClientLastDataUpdateTime()`

Returns

the client's last data update time.

getNewDataUpdateTime() method

Returns the new "last data update time", as set by a prior call to setNewDataUpdateTime.

Syntax

`public GregorianCalendar getNewDataUpdateTime()`

Returns

The new update time, or `null` if `setNewDataUpdateTime()` hasn't been called yet.

getSession() method

Return the session for this complex table transmission.

Syntax

`public ComplexTableSession getSession()`

Returns

the session

initialize() method [deprecated]

Deprecated. Subclass overrides of this method should be renamed to build(). This method has been renamed to build().

Syntax

`public final void initialize()` throws `AgentryException`

Exceptions

- **AgentryException class –** not thrown
- **UnsupportedOperationException –** to report that the method is no longer used.

Usage

Override that method instead of this one. This method is only here to cause compilation errors in legacy code, and may be removed in a future release.

isRebuilding() method

This method returns the previously cached result of the Agentry Server's call to the willRebuildTable() method.

Syntax

```
public boolean isRebuilding()
```

Returns

the value of the _rebuilding member variable.

lastUpdateDate() method

Returns the day-of-month component of the complex table data's last update time.

Syntax

```
final int lastUpdateDate()
```

Returns

The day-of-month component of the complex table data's last update time.

Usage

This method is called by the Agentry Server to retrieve the last update time.

lastUpdateHours() method

Returns the hour component of the complex table data's last update time.

Syntax

```
final int lastUpdateHours()
```

Returns

The hour component of the complex table data's last update time.

Usage

This method is called by the Agentry Server to retrieve the last update time.

lastUpdateMinutes() method

Returns the minute component of the complex table data's last update time.

Syntax

```
final int lastUpdateMinutes()
```

Returns

The minute component of the complex table data's last update time.

Usage
This method is called by the Agentry Server to retrieve the last update time.

lastUpdateMonth() method
Returns the month component of the complex table data's last update time.

Syntax
```
final int  lastUpdateMonth ()
```

Returns
The month component of the complex table data's last update time.

Usage
This method is called by the Agentry Server to retrieve the last update time.

lastUpdateSeconds() method
Returns the seconds component of the complex table data's last update time.

Syntax
```
final int  lastUpdateSeconds ()
```

Returns
The seconds component of the complex table data's last update time.

Usage
This method is called by the Agentry Server to retrieve the last update time.

lastUpdateYear() method
Returns the year component of the complex table data's last update time.

Syntax
```
final int  lastUpdateYear ()
```

Returns
The year component of the complex table data's last update time.

Usage
This method is called by the Agentry Server to retrieve the last update time.

reload() method [deprecated]

Deprecated. Subclass overrides of this method should be renamed to willRebuildTable(). This method has been renamed to willRebuildTable().

Syntax

`public final boolean reload ()` throws `AgentryException`

Returns

Nothing. Always throws `UnsupportedOperationException`.

Exceptions

- **AgentryException class –** not thrown
- **UnsupportedOperationException –** to report that the method is no longer used.

Usage

Override that method instead of this one. This method is only here to cause compilation errors in legacy code, and may be removed in a future release.

setNewDataUpdateTime(GregorianCalendar) method

This method sets the new "last data update time" for the complex table.

Syntax

`public void setNewDataUpdateTime ( GregorianCalendar` *dataUpdateTime* `)`

Parameters

- **dataUpdateTime –** The last update time of the complex table's data.

Exceptions

- **IllegalArgumentException –** if `dataUpdateTime` is null, or is set to the Agentry "invalid" date/time value.

Usage

By default, the last update time will be "now"; however, if your underlying data source has a concept of a last update time, you can pass that on to the Agentry Server by calling this method from the constructor, build(), or dataIterator() to set it.

Note that calling this method does not affect the return value of getClientLastDataUpdateTime(); the two values are maintained separately.

willRebuildTable() method

This method is called by the Agentry server to determine whether the server should rebuild the client's complex table from scratch.

Syntax

`public boolean willRebuildTable ()` throws `AgentryException`

Returns

`true` if the client's table should be rebuilt from scratch, or `false` if the client's existing table should be updated.

Exceptions

- **AgentryException class –** if an error occurs.

Usage

If you return `true` from this method, then a subsequent call to dataIterator() should return the full set of complex table records and deleteIterator() should return nothing; if you return `false` from this method then dataIterator() and deleteIterator() should return incremental changes.

Note that if the client's last data update date was invalid, then the Agentry Server will assume that the table needs to be reloaded and will not call this method at all.

When the Agentry server calls this method, it will store the result of the call into the _rebuilding field. That field will also be initialized by the constructor to `true` if _clientLastDataUpdateTime is invalid or `false` if it is not, so that the field will contain the correct value even if the Agentry server does not call this method.

The default implementation of this method returns the value of _rebuilding as set by the constructor.

_clientLastDataUpdateTime variable

Storage for the client's last update time that was passed into the constructor.

Syntax

`protected SycloCalendar _clientLastDataUpdateTime`

Usage

This is a SycloCalendar instead of a GregorianCalendar because it is possible for it to contain Agentry's "invalid" timestamp.

_rebuilding variable

This member will be set by the Agentry Server to whatever the willRebuildTable() method returns.

Syntax

```
protected  boolean  _rebuilding
```

Usage

It can be read via the isRebuilding() method.

_session variable

Storage for the session that was passed into the constructor.

Syntax

```
protected  ComplexTableSession  _session
```

DataTableObject class

This class represents a single row in an Agentry data table, which consists of a key/value pair.

Syntax

```
public class  DataTableObject extends Object
```

Members

All members of DataTableObject, including inherited members. **Variables**

Modifier and Type	Variable	Description
public String	*code* on page 414	The code/key for the row.
public String	*value* on page 414	The value for the row.

Constructors

Modifier and Type	Constructor	Description
public	*DataTableObject(String, String)* on page 413	Constructs a new DataTableObject object.

Methods

Modifier and Type	Method	Description
public String	*code()* on page 413	Deprecated. Use getKey(). Return the code/key for this row.
public boolean	*equals(Object)* on page 413	

Modifier and Type	Method	Description
public String	*getKey()* on page 413	Return the code/key for this row.
public String	*getValue()* on page 414	Return the value for this row.
public int	*hashCode()* on page 414	
public String	*value()* on page 414	Deprecated. Use getValue(). Return the value for this row.

DataTableObject(String, String) constructor
Constructs a new DataTableObject object.

Syntax
```
public   DataTableObject ( String code1 ,  String value1 )
```

Parameters

- **code1 –** The code/key for the row
- **value1 –** The value for the row

code() method [deprecated]
Deprecated. Use getKey(). Return the code/key for this row.

Syntax
```
public  String  code ()
```

Returns
the code

equals(Object) method

Syntax
```
public  boolean  equals ( Object obj )
```

getKey() method
Return the code/key for this row.

Syntax
```
public  String  getKey ()
```

Returns
the code

getValue() method

Return the value for this row.

Syntax

```
public String getValue()
```

Returns

the value

hashCode() method

Syntax

```
public int hashCode()
```

value() method [deprecated]

Deprecated. Use getValue(). Return the value for this row.

Syntax

```
public String value()
```

Returns

the value

code variable

The code/key for the row.

Syntax

```
public String code
```

value variable

The value for the row.

Syntax

```
public String value
```

DataTableSession class

The DataTableSession class encapsulates the processing involved in a data table retrieval within an Agentry-based application.

Syntax

```
public class DataTableSession extends Session
```

Members

All members of DataTableSession, including inherited members. **Constructors**

Modifier and Type	Constructor	Description
public	*DataTableSession(String, Server, SessionData, User)* on page 416	Construct a new session.

Inherited members from Session

Modifier and Type	Member	Description
public final void	*debug(String)* on page 503	Write the given message to a debug log, if debugging is enabled.
public String	*getName()* on page 503	Returns the name of the session, as configured in the Agentry application.
public Server	*getServer()* on page 503	Returns the Server singleton object that the Java system connection is currently using.
public SessionData	*getSessionData()* on page 504	Returns the session data for this session.
public User	*getUser()* on page 504	Returns the user for this session, if any.
protected	*Session(String, Server, SessionData, User)* on page 502	Construct a new session, and save each of the arguments so that they can be retrieved later via the getName(), getServer(), getSessionData(), and getUser() methods.
protected	*Session(String, Server, SessionData)* on page 502	Construct a new session, and save each of the arguments so that they can be retrieved later via the getName(), getServer(), and getSessionData() methods.
public void	*sessionAborted()* on page 504	This is called if the session is aborted (e.g., by an exception).

Usage

This class contains the session data for the data table retrieval, as well the User object for the user performing the retrieval, and a reference to the Server singleton object.

DataTableSession(String, Server, SessionData, User) constructor

Construct a new session.

Syntax

```
public  DataTableSession ( String tableName ,  Server server ,
SessionData sessionData ,  User user )
```

Parameters

- **tableName –** The data table name, as configured in the Agentry application.
- **server –** The `Server` object that the Java System connection was configured to use.
- **sessionData –** Session data for this transmit.
- **user –** The client user performing the transmit.

Usage

This constructor is called by the Server.createDataTableSession method. Subclasses should implement a constructor with the same signature as this one.

DataTable< DTOBJ extends DataTableObject > class

The DataTable class in the AJAPI encapsulates the synchronization of a data table.

Syntax

```
public abstract class  DataTable< DTOBJ extends
DataTableObject > extends AgentryJavaBackEndManagedObject
```

Members

All members of DataTable< DTOBJ extends DataTableObject >, including inherited members. **Variables**

Modifier and Type	Variable	Description
protected SycloCalendar	*_clientLastDataUpdateTime* on page 420	Client's last data update date.
protected DataTableSession	*_session* on page 420	Storage for the session that was passed into the constructor.

Methods

Modifier and Type	Method	Description
public	*DataTable(DataTableSession, GregorianCalendar)* on page 418	Constructs a new DataTable.
public SycloCalendar	*getClientLastDataUpdate-Time()* on page 418	Returns the client's last data up-date date and time.
public DataTableSession	*getSession()* on page 418	Returns the session data for this data table transmission.
public void	*initialize()* on page 419	This method is called after this object is constructed.
public abstract boolean	*isOutOfDate()* on page 419	Returns whether the client's da-ta is out of date, based on the last update date passed in via the constructor.
public abstract Iterator< DTOBJ >	*iterator()* on page 419	Builds an Iterator object that will iterate through the DataTa-bleObject objects that contain the rows for the data table.

Usage

When a data table is created in the Agentry Editor, a subclass is automatically defined that extends this class. The designer must implement this subclass in order to retrieve the data for the data table. Such an implementation should retrieve the data table's data from some source, construct a list or set of DataTableObject objects, and then return an iterator object that iterates over that list.

The `DataTable` class provides two main pieces of functionality. First, it determines if the data table needs to be reloaded and, second, it provides the factory method to construct a `Iterator` object to process the returned data.

The Agentry server will call the methods of this class in the following sequence:

1. Constructor
2. initialize()
3. isOutOfDate()
4. iterator() (only if isOutOfDate() returned `true`)

Note that for each client transmit, a new instance of this class is created by the Agentry server.

DataTable(DataTableSession, GregorianCalendar) method

Constructs a new DataTable.

Syntax

`public DataTable ( DataTableSession` *session* `, GregorianCalendar` *clientLastDataUpdate* `)`

Parameters

- **session –** An object of type DataTableSession that provides access to pertinent session data for the data table.
- **clientLastDataUpdate –** This argument contains the date and time that the data table was last updated on the client application. This value can be accessed through the protected member _clientLastDataUpdateTime.

Usage

This method is called by the Agentry Server (indirectly via whatever subclasses you define) whenever a data table request is received from the client application. Extensions of this class should take the same arguments as this class, and pass those arguments on to this parent constructor.

getClientLastDataUpdateTime() method

Returns the client's last data update date and time.

Syntax

```
public SycloCalendar getClientLastDataUpdateTime ()
```

Returns

the date and time of the client's last update. This might be the Agentry "Invalid Date" time, which indicates that the client did not have any copy of the data table at all; note that it is generally safe to not check for the "invalid date" value, since it is a time far in the past (circa 1900) and therefore should fail any check against a recent date and time.

getSession() method

Returns the session data for this data table transmission.

Syntax

```
public DataTableSession getSession ()
```

Returns

the session data

initialize() method
This method is called after this object is constructed.

Syntax
`public void initialize ()` throws `AgentryException`

Exceptions

- **AgentryException class –** if an error occurs

Usage
Initialization can be performed here if it might result in an exception.

isOutOfDate() method
Returns whether the client's data is out of date, based on the last update date passed in via the constructor.

Syntax
`public abstract boolean isOutOfDate ()` throws `AgentryException`

Returns
`true` if the client's data is out of date, or `false` if not.

Exceptions

- **AgentryException class –** if an error occurs

Usage
If this returns `true`, then the Agentry Server will call iterator() to retrieve the latest data for the data table and send it to the client.

iterator() method
Builds an Iterator object that will iterate through the DataTableObject objects that contain the rows for the data table.

Syntax
`public abstract Iterator< DTOBJ > iterator ()` throws `AgentryException`

Returns
an `Iterator` object.

Exceptions

- **AgentryException class –** if an error occurs.

Usage

This is called by the Agentry Server to retrieve the actual data for the data table, when the isOutOfDate() method returns `true`. Note that once the data has been sent to the client, the server will change the last update date and time on the client to "now".

_clientLastDataUpdateTime variable

Client's last data update date.

Syntax

```
protected  SycloCalendar  _clientLastDataUpdateTime
```

_session variable

Storage for the session that was passed into the constructor.

Syntax

```
protected  DataTableSession  _session
```

FatalTransactionException class

This exception can be thrown from a transactional Steplet to indicate that the transaction failed in a way that cannot be corrected or recovered from.

Syntax

```
public class  FatalTransactionException extends
AgentryException
```

Members

All members of FatalTransactionException, including inherited members. **Constructors**

Modifier and Type	Constructor	Description
public	*FatalTransactionException(String)* on page 422	Constructs a new FatalTransactionException object.
public	*FatalTransactionException(String, Throwable)* on page 422	Constructs a new FatalTransactionException object.
public	*FatalTransactionException(String, String, String)* on page 422	Constructs a new FatalTransactionException object that will cause an error notification window to appear on the client.

Modifier and Type	Constructor	Description
public	*FatalTransactionException(String, String, String, Throwable)* on page 423	Constructs a new FatalTransactionException object that will cause an error notification window to appear on the client.

Inherited members from AgentryException

Modifier and Type	Member	Description
public	*AgentryException(String)* on page 395	Constructs a new AgentryException object.
public	*AgentryException(String, Throwable)* on page 395	Constructs a new AgentryException object.
public	*AgentryException(String, String, String, Throwable)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.
public	*AgentryException(String, String, String)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.
public final String	*getNotificationText()* on page 397	Returns the notification window text.
public final String	*getNotificationTitle()* on page 397	Returns the notification window title.
public final String	*getOkButtonLabel()* on page 397	Returns the notification window button label.

Usage

It will remove the transaction from the client and save it to the failed transactions queue on the Agentry Server. It will optionally display an error notification window to the client user.

This exception combines and replaces the functionality of the `SycloFatalExceptionWithMessage` and `SycloFatalExceptionWithoutMessage` exceptions from version 4 of the AJAPI.

FatalTransactionException(String) constructor

Constructs a new FatalTransactionException object.

Syntax

`public FatalTransactionException ( String` *message* `)`

Parameters

- **message –** The error message, which will be logged

Usage

No error notification window will be displayed on the client.

This constructor creates an exception that results in a "Fatal without message" transaction error.

FatalTransactionException(String, Throwable) constructor

Constructs a new FatalTransactionException object.

Syntax

`public FatalTransactionException ( String` *message* `, Throwable` *cause* `)`

Parameters

- **message –** The error message, which will be logged
- **cause –** The underlying exception

Usage

No error notification window will be displayed on the client.

This constructor creates an exception that results in a "Fatal without message" transaction error.

FatalTransactionException(String, String, String) constructor

Constructs a new FatalTransactionException object that will cause an error notification window to appear on the client.

Syntax

`public FatalTransactionException ( String` *title* `, String` *text* `, String` *okLabel* `)`

Parameters

- **title –** The window title for the error displayed on the client.

- **text –** The window text for the error displayed on the client.
- **okLabel –** The label for the acknowledgment button in the client's error window.

Usage
This constructor creates an exception that results in a "Fatal with message" transaction error.

FatalTransactionException(String, String, String, Throwable) constructor
Constructs a new FatalTransactionException object that will cause an error notification window to appear on the client.

Syntax
```
public  FatalTransactionException ( String title ,  String text ,
String okLabel ,  Throwable cause )
```

Parameters

- **title –** The window title for the error displayed on the client.
- **text –** The window text for the error displayed on the client.
- **okLabel –** The label for the acknowledgment button in the client's error window.
- **cause –** The causing exception

Usage
This constructor creates an exception that results in a "Fatal with message" transaction error.

FatalTransactionExceptionStop class
This exception can be thrown from a transactional Steplet to indicate that the transaction failed in a way that cannot be corrected or recovered from.

Syntax
```
public class  FatalTransactionExceptionStop extends
AgentryException
```

Members
All members of FatalTransactionExceptionStop, including inherited members. **Constructors**

Modifier and Type	Constructor	Description
public	*FatalTransactionException-Stop(String, String, String)* on page 424	Constructs a new FatalTransactionExceptionStop object that will cause an error notification window to appear on the client.

Modifier and Type	Constructor	Description
public	*FatalTransactionException-Stop(String, String, String, Throwable)* on page 425	Constructs a new FatalTransactionExceptionStop object that will cause an error notification window to appear on the client.

Inherited members from AgentryException

Modifier and Type	Member	Description
public	*AgentryException(String)* on page 395	Constructs a new AgentryException object.
public	*AgentryException(String, Throwable)* on page 395	Constructs a new AgentryException object.
public	*AgentryException(String, String, String, Throwable)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.
public	*AgentryException(String, String, String)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.
public final String	*getNotificationText()* on page 397	Returns the notification window text.
public final String	*getNotificationTitle()* on page 397	Returns the notification window title.
public final String	*getOkButtonLabel()* on page 397	Returns the notification window button label.

Usage

It will remove the transaction from the client and save it to the failed transactions queue on the Agentry Server. It will display an error notification window to the client user. The tranmit will always be stopped (the user will not be given the choice to stop)

FatalTransactionExceptionStop(String, String, String) constructor

Constructs a new FatalTransactionExceptionStop object that will cause an error notification window to appear on the client.

Syntax

```
public FatalTransactionExceptionStop ( String title , String text ,
String okLabel )
```

Parameters

- **title –** The window title for the error displayed on the client.
- **text –** The window text for the error displayed on the client.
- **okLabel –** The label for the acknowledgment button in the client's error window.

Usage

This constructor creates an exception that results in a "Fatal with message, always stop" transaction error.

FatalTransactionExceptionStop(String, String, String, Throwable) constructor

Constructs a new FatalTransactionExceptionStop object that will cause an error notification window to appear on the client.

Syntax

```
public  FatalTransactionExceptionStop(String title, String text,
String okLabel, Throwable cause)
```

Parameters

- **title –** The window title for the error displayed on the client.
- **text –** The window text for the error displayed on the client.
- **okLabel –** The label for the acknowledgment button in the client's error window.
- **cause –** The causing exception

Usage

This constructor creates an exception that results in a "Fatal with message, always stop" transaction error.

FetchSession class

The FetchSession class encapsulates the processing involved in a fetch.

Syntax

```
public class FetchSession extends Session
```

Members

All members of FetchSession, including inherited members. **Constructors**

Modifier and Type	Constructor	Description
public	*FetchSession(String, Server, SessionData, User)* on page 428	Construct a new session.

Methods

Modifier and Type	Method	Description
public void	*beginClientExchange()* on page 428	This method is called by the server prior to the "Client Exchange Steps" within the fetch are executed.
public void	*beginFetchObjectRead()* on page 428	This method is called by the server prior to the execution of the "Object Read Steps" for the fetch.
public void	*beginFetchRemoval()* on page 429	This method is called by the server prior to the "Removal Steps" within the fetch are executed.
public void	*beginServerExchange()* on page 429	This method is called by the server prior to the "Server Exchange Steps" within the fetch are executed.
public void	*endClientExchange()* on page 429	This method is called after the "Client Exchange Steps" for the fetch have been successfully completed.
public void	*endFetchObjectRead()* on page 429	This method is called after the "Object Read Steps" for the fetch have been successfully completed.
public void	*endFetchRemoval()* on page 430	This method is called after the "Removal Steps" for the fetch have been successfully completed.
public void	*endServerExchange()* on page 430	This method is called after the "Server Exchange Steps" for the fetch have been successfully completed.

Inherited members from Session

Modifier and Type	Member	Description
public final void	*debug(String)* on page 503	Write the given message to a debug log, if debugging is enabled.
public String	*getName()* on page 503	Returns the name of the session, as configured in the Agentry application.
public Server	*getServer()* on page 503	Returns the Server singleton object that the Java system connection is currently using.
public SessionData	*getSessionData()* on page 504	Returns the session data for this session.
public User	*getUser()* on page 504	Returns the user for this session, if any.
protected	*Session(String, Server, SessionData, User)* on page 502	Construct a new session, and save each of the arguments so that they can be retrieved later via the getName(), getServer(), getSessionData(), and getUser() methods.
protected	*Session(String, Server, SessionData)* on page 502	Construct a new session, and save each of the arguments so that they can be retrieved later via the getName(), getServer(), and getSessionData() methods.
public void	*sessionAborted()* on page 504	This is called if the session is aborted (e.g., by an exception).

Usage

In brief, a fetch is the component of the application that defines how data is synchronized with the back end system. It is made up of steps (which are implemented by the Steplet class in the Java system connection), each of which perform a specific task related to the synchronization process. These steps are organized into groups within the fetch for specific areas of the data synchronization. These areas include the "Object Read", "Client Exchange", "Server Exchange", and "Removal" steps.

This class contains methods to perform actions before and after each of these groups of steps. The default implementation of these methods in the AJAPI perform no additional specific actions. A designer can extend this class if special processing is required before or after each of these groups of steps are processed. If this class is extended, the Server class must also be extended and its `createFetchSession()` method must be overridden to return the designer implemented subclass of the `FetchSession` class.

FetchSession(String, Server, SessionData, User) constructor

Construct a new session.

Syntax

```
public  FetchSession ( String fetchName ,  Server server ,
SessionData sessionData ,  User user )
```

Parameters

- **fetchName –** The fetch name, as configured in the Agentry application.
- **server –** The `Server` object that the Java System connection was configured to use.
- **sessionData –** Session data for this fetch.
- **user –** The client user performing the fetch.

Usage

This constructor is called by the Server.createFetchSession method. Subclasses should implement a constructor with the same signature as this one.

beginClientExchange() method

This method is called by the server prior to the "Client Exchange Steps" within the fetch are executed.

Syntax

```
public void beginClientExchange ()
```

Usage

Any processing that should take place prior to the execution of these steps should be implemented in this method.

beginFetchObjectRead() method

This method is called by the server prior to the execution of the "Object Read Steps" for the fetch.

Syntax

```
public void beginFetchObjectRead ()
```

Usage

Any processing that should take place prior to the execution of these steps should be implemented in this method.

beginFetchRemoval() method

This method is called by the server prior to the "Removal Steps" within the fetch are executed.

Syntax

```
public void beginFetchRemoval ()
```

Usage

Any processing that should take place prior to the execution of these steps should be implemented in this method.

beginServerExchange() method

This method is called by the server prior to the "Server Exchange Steps" within the fetch are executed.

Syntax

```
public void beginServerExchange ()
```

Usage

Any processing that should take place prior to the execution of these steps should be implemented in this method.

endClientExchange() method

This method is called after the "Client Exchange Steps" for the fetch have been successfully completed.

Syntax

```
public void endClientExchange ()
```

Usage

Any processing that should take place after the execution of these steps should be implemented in this method.

endFetchObjectRead() method

This method is called after the "Object Read Steps" for the fetch have been successfully completed.

Syntax

```
public void endFetchObjectRead ()
```

Usage

Any processing that should take place after the execution of these steps should be implemented in this method.

endFetchRemoval() method

This method is called after the "Removal Steps" for the fetch have been successfully completed.

Syntax

```
public void endFetchRemoval ()
```

Usage

Any processing that should take place after the execution of these steps should be implemented in this method.

endServerExchange() method

This method is called after the "Server Exchange Steps" for the fetch have been successfully completed.

Syntax

```
public void endServerExchange ()
```

Usage

Any processing that should take place after the execution of these steps should be implemented in this method.

LoginBlockedException class

This exception is thrown from the login methods of the Server class to indicate that the user has been blocked from accessing this system connection.

Syntax

```
public class LoginBlockedException extends LoginException
```

Members

All members of LoginBlockedException, including inherited members. **Constructors**

Modifier and Type	Constructor	Description
public	*LoginBlockedException()* on page 432	Constructs a new LoginBlocke-dException object.
public	*LoginBlockedExcep-tion(String)* on page 432	Constructs a new LoginBlocke-dException object.
public	*LoginBlockedExcep-tion(String, Throwable)* on page 432	Constructs a new LoginBlocke-dException object.

Inherited members from LoginException

Modifier and Type	Member	Description
public	*LoginException()* on page 434	Constructs a new login exception.
public	*LoginException(String)* on page 434	Constructs a new login exception with the given error message, which will be passed to the Agentry client.
public	*LoginException(String, Throwable)* on page 434	Constructs a new login exception with the given error message, which will be passed to the Agentry client.

Inherited members from AgentryException

Modifier and Type	Member	Description
public	*AgentryException(String)* on page 395	Constructs a new AgentryException object.
public	*AgentryException(String, Throwable)* on page 395	Constructs a new AgentryException object.
public	*AgentryException(String, String, String, Throwable)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.
public	*AgentryException(String, String, String)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.
public final String	*getNotificationText()* on page 397	Returns the notification window text.
public final String	*getNotificationTitle()* on page 397	Returns the notification window title.
public final String	*getOkButtonLabel()* on page 397	Returns the notification window button label.

Usage

It indicates that no further logins should be attempted until the problem is corrected. It can be used, for example, to indicate that the user does not have sufficient privileges to access a remote system, or that the user's account has been locked out due to too many incorrect passwords.

LoginBlockedException() constructor

Constructs a new LoginBlockedException object.

Syntax

```
public   LoginBlockedException ()
```

Usage

The client will report a default error message.

LoginBlockedException(String) constructor

Constructs a new LoginBlockedException object.

Syntax

```
public   LoginBlockedException ( String message )
```

Parameters

- **message –** The error message, which will be displayed on the client.

LoginBlockedException(String, Throwable) constructor

Constructs a new LoginBlockedException object.

Syntax

```
public   LoginBlockedException ( String message ,  Throwable
cause )
```

Parameters

- **message –** The error message, which will be displayed on the client.
- **cause –** The exception that triggered this exception.

LoginException class

This is the base class for all login/authentication exceptions.

Syntax

```
public class  LoginException extends AgentryException
```

Derived classes

- *LoginBlockedException* on page 430
- *LoginSkippedException* on page 435
- *PasswordExpiredCannotChangeException* on page 437
- *PasswordExpiredException* on page 439

- *PasswordInvalidException* on page 442
- *PasswordWarningCannotChangeException* on page 444
- *PasswordWarningException* on page 446

Members

All members of LoginException, including inherited members. **Constructors**

Modifier and Type	Constructor	Description
public	*LoginException()* on page 434	Constructs a new login exception.
public	*LoginException(String)* on page 434	Constructs a new login exception with the given error message, which will be passed to the Agentry client.
public	*LoginException(String, Throwable)* on page 434	Constructs a new login exception with the given error message, which will be passed to the Agentry client.

Inherited members from AgentryException

Modifier and Type	Member	Description
public	*AgentryException(String)* on page 395	Constructs a new AgentryException object.
public	*AgentryException(String, Throwable)* on page 395	Constructs a new AgentryException object.
public	*AgentryException(String, String, String, Throwable)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.
public	*AgentryException(String, String, String)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.
public final String	*getNotificationText()* on page 397	Returns the notification window text.
public final String	*getNotificationTitle()* on page 397	Returns the notification window title.

Modifier and Type	Member	Description
public final String	*getOkButtonLabel()* on page 397	Returns the notification window button label.

Usage

Subclasses of this exception are thrown from the login methods of the Server class.

If given message text, this class will place the message text into the `notificationText` field of AgentryException as well as using it for the exception's message. This is done to make it easier for the Java Back End to retrieve the unadulterated message text, since the C++ exception wrapper classes sometimes augment the exception's message text.

LoginException() constructor

Constructs a new login exception.

Syntax

```
public   LoginException ()
```

Usage

The client will report a default error message.

LoginException(String) constructor

Constructs a new login exception with the given error message, which will be passed to the Agentry client.

Syntax

```
public   LoginException ( String message )
```

Parameters

- **message –** The error message.

LoginException(String, Throwable) constructor

Constructs a new login exception with the given error message, which will be passed to the Agentry client.

Syntax

```
public   LoginException ( String message ,  Throwable cause )
```

Parameters

- **message –** The error message.
- **cause –** The underlying exception that triggered this exception.

LoginSkippedException class

This exception can be thrown from the login methods of the Server class to indicate that the system connection is not authenticating the user at all, and that some other system connection must be relied upon to perform the authentication.

Syntax

```
public class  LoginSkippedException extends LoginException
```

Members

All members of LoginSkippedException, including inherited members. **Constructors**

Modifier and Type	Constructor	Description
public	*LoginSkippedException()* on page 436	Constructs a new LoginSkippedException object.
public	*LoginSkippedException(String)* on page 436	Constructs a new LoginSkippedException object.
public	*LoginSkippedException(String, Throwable)* on page 437	Constructs a new LoginSkippedException object.

Inherited members from LoginException

Modifier and Type	Member	Description
public	*LoginException()* on page 434	Constructs a new login exception.
public	*LoginException(String)* on page 434	Constructs a new login exception with the given error message, which will be passed to the Agentry client.
public	*LoginException(String, Throwable)* on page 434	Constructs a new login exception with the given error message, which will be passed to the Agentry client.

Inherited members from AgentryException

Modifier and Type	Member	Description
public	*AgentryException(String)* on page 395	Constructs a new AgentryException object.

Modifier and Type	Member	Description
public	*AgentryException(String, Throwable)* on page 395	Constructs a new AgentryException object.
public	*AgentryException(String, String, String, Throwable)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.
public	*AgentryException(String, String, String)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.
public final String	*getNotificationText()* on page 397	Returns the notification window text.
public final String	*getNotificationTitle()* on page 397	Returns the notification window title.
public final String	*getOkButtonLabel()* on page 397	Returns the notification window button label.

Usage

It is functionally equivalent to setting the `enableAuthentication` setting for the Java System connection (in the `Agentry.ini` file) to `false`, except that it can be thrown on a per-user basis.

LoginSkippedException() constructor

Constructs a new LoginSkippedException object.

Syntax

```
public   LoginSkippedException ()
```

Usage

The client will report a default error message.

LoginSkippedException(String) constructor

Constructs a new LoginSkippedException object.

Syntax

```
public   LoginSkippedException ( String message )
```

Parameters

- **message –** The error message, which will be displayed on the client.

LoginSkippedException(String, Throwable) constructor

Constructs a new LoginSkippedException object.

Syntax

```
public  LoginSkippedException ( String message ,  Throwable
cause )
```

Parameters

- **message –** The error message, which will be displayed on the client.
- **cause –** The exception that triggered this exception.

PasswordExpiredCannotChangeException class

This exception can be thrown from the login methods of the Server class to indicate that the user's password has expired, and that this system connection does not support changing it.

Syntax

```
public class  PasswordExpiredCannotChangeException extends
LoginException
```

Members

All members of PasswordExpiredCannotChangeException, including inherited members.

Constructors

Modifier and Type	Constructor	Description
public	*PasswordExpiredCannotChangeException()* on page 439	Constructs a new PasswordExpiredCannotChangeException object.
public	*PasswordExpiredCannotChangeException(String)* on page 439	Constructs a new LoginBlockedException object.
public	*PasswordExpiredCannotChangeException(String, Throwable)* on page 439	Constructs a new LoginBlockedException object.

Inherited members from LoginException

Modifier and Type	Member	Description
public	*LoginException()* on page 434	Constructs a new login exception.
public	*LoginException(String)* on page 434	Constructs a new login exception with the given error message, which will be passed to the Agentry client.
public	*LoginException(String, Throwable)* on page 434	Constructs a new login exception with the given error message, which will be passed to the Agentry client.

Inherited members from AgentryException

Modifier and Type	Member	Description
public	*AgentryException(String)* on page 395	Constructs a new AgentryException object.
public	*AgentryException(String, Throwable)* on page 395	Constructs a new AgentryException object.
public	*AgentryException(String, String, String, Throwable)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.
public	*AgentryException(String, String, String)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.
public final String	*getNotificationText()* on page 397	Returns the notification window text.
public final String	*getNotificationTitle()* on page 397	Returns the notification window title.
public final String	*getOkButtonLabel()* on page 397	Returns the notification window button label.

Usage

The user will not be allowed to proceed with their transmission until they change their password via some other means (such as via another system).

This exception should be used instead of PasswordExpiredException if the various password-changing methods of User have not been implemented.

PasswordExpiredCannotChangeException() constructor
Constructs a new PasswordExpiredCannotChangeException object.

Syntax
```
public   PasswordExpiredCannotChangeException ()
```

Usage
The client will report a default error message.

PasswordExpiredCannotChangeException(String) constructor
Constructs a new LoginBlockedException object.

Syntax
```
public   PasswordExpiredCannotChangeException ( String message )
```

Parameters

- **message –** The error message, which will be displayed on the client.

PasswordExpiredCannotChangeException(String, Throwable) constructor
Constructs a new LoginBlockedException object.

Syntax
```
public   PasswordExpiredCannotChangeException ( String message ,
Throwable cause )
```

Parameters

- **message –** The error message, which will be displayed on the client.
- **cause –** The exception that triggered this exception.

PasswordExpiredException class
This exception can be thrown from the login methods of the Server class to indicate that the user's password has expired and must be changed before the client's transmission will be allowed to proceed.

Syntax
```
public class  PasswordExpiredException extends LoginException
```

Members
All members of PasswordExpiredException, including inherited members. **Constructors**

Modifier and Type	Constructor	Description
public	*PasswordExpiredException()* on page 441	Constructs a new PasswordExpiredException object.
public	*PasswordExpiredException(String)* on page 441	Constructs a new LoginBlockedException object.
public	*PasswordExpiredException(String, Throwable)* on page 441	Constructs a new LoginBlockedException object.

Inherited members from LoginException

Modifier and Type	Member	Description
public	*LoginException()* on page 434	Constructs a new login exception.
public	*LoginException(String)* on page 434	Constructs a new login exception with the given error message, which will be passed to the Agentry client.
public	*LoginException(String, Throwable)* on page 434	Constructs a new login exception with the given error message, which will be passed to the Agentry client.

Inherited members from AgentryException

Modifier and Type	Member	Description
public	*AgentryException(String)* on page 395	Constructs a new AgentryException object.
public	*AgentryException(String, Throwable)* on page 395	Constructs a new AgentryException object.
public	*AgentryException(String, String, String, Throwable)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.
public	*AgentryException(String, String, String)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.

Modifier and Type	Member	Description
public final String	*getNotificationText()* on page 397	Returns the notification window text.
public final String	*getNotificationTitle()* on page 397	Returns the notification window title.
public final String	*getOkButtonLabel()* on page 397	Returns the notification window button label.

Usage
The user will be prompted to enter a new password; if they do, the various password-changing methods of the User class will be invoked.

PasswordExpiredException() constructor
Constructs a new PasswordExpiredException object.

Syntax
`public PasswordExpiredException ()`

Usage
The client will report a default error message.

PasswordExpiredException(String) constructor
Constructs a new LoginBlockedException object.

Syntax
`public PasswordExpiredException ( String` *message* `)`

Parameters

- **message –** The error message, which will be displayed on the client.

PasswordExpiredException(String, Throwable) constructor
Constructs a new LoginBlockedException object.

Syntax
`public PasswordExpiredException ( String` *message* `, Throwable` *cause* `)`

Parameters

- **message –** The error message, which will be displayed on the client.
- **cause –** The exception that triggered this exception.

PasswordInvalidException class

This exception can be thrown from the various login methods of the Server class to indicate that the user's password was wrong.

Syntax

```
public class  PasswordInvalidException extends LoginException
```

Members

All members of PasswordInvalidException, including inherited members. **Constructors**

Modifier and Type	Constructor	Description
public	*PasswordInvalidException()* on page 443	Constructs a new PasswordInvalidException object.
public	*PasswordInvalidException(String)* on page 443	Constructs a new PasswordInvalidException object.
public	*PasswordInvalidException(String, Throwable)* on page 444	Constructs a new PasswordInvalidException object.

Inherited members from LoginException

Modifier and Type	Member	Description
public	*LoginException()* on page 434	Constructs a new login exception.
public	*LoginException(String)* on page 434	Constructs a new login exception with the given error message, which will be passed to the Agentry client.
public	*LoginException(String, Throwable)* on page 434	Constructs a new login exception with the given error message, which will be passed to the Agentry client.

Inherited members from AgentryException

Modifier and Type	Member	Description
public	*AgentryException(String)* on page 395	Constructs a new AgentryException object.

Modifier and Type	Member	Description
public	*AgentryException(String, Throwable)* on page 395	Constructs a new AgentryException object.
public	*AgentryException(String, String, String, Throwable)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.
public	*AgentryException(String, String, String)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.
public final String	*getNotificationText()* on page 397	Returns the notification window text.
public final String	*getNotificationTitle()* on page 397	Returns the notification window title.
public final String	*getOkButtonLabel()* on page 397	Returns the notification window button label.

Usage
It can also be used to indicate other transitory authentication failures (such as the remote system being unreachable).

PasswordInvalidException() constructor
Constructs a new PasswordInvalidException object.

Syntax
```
public   PasswordInvalidException ()
```

Usage
The client will report a default error message.

PasswordInvalidException(String) constructor
Constructs a new PasswordInvalidException object.

Syntax
```
public   PasswordInvalidException ( String message )
```

Parameters

- **message –** The error message, which will be displayed on the client.

PasswordInvalidException(String, Throwable) constructor

Constructs a new PasswordInvalidException object.

Syntax

```
public  PasswordInvalidException ( String message ,  Throwable
cause )
```

Parameters

- **message –** The error message, which will be displayed on the client.
- **cause –** The exception that triggered this exception.

PasswordWarningCannotChangeException class

This exception can be thrown from the login methods of the Server class to indicate that the user's password is going to expire soon, and that this system connection does not support changing it.

Syntax

```
public class  PasswordWarningCannotChangeException extends
LoginException
```

Members

All members of PasswordWarningCannotChangeException, including inherited members.

Constructors

Modifier and Type	Constructor	Description
public	*PasswordWarningCannotChangeException()* on page 446	Constructs a new PasswordWarningCannotChangeException object.
public	*PasswordWarningCannotChangeException(String)* on page 446	Constructs a new PasswordWarningCannotChangeException object.
public	*PasswordWarningCannotChangeException(String, Throwable)* on page 446	Constructs a new PasswordWarningCannotChangeException object.

Inherited members from LoginException

Modifier and Type	Member	Description
public	*LoginException()* on page 434	Constructs a new login exception.

Modifier and Type	Member	Description
public	*LoginException(String)* on page 434	Constructs a new login exception with the given error message, which will be passed to the Agentry client.
public	*LoginException(String, Throwable)* on page 434	Constructs a new login exception with the given error message, which will be passed to the Agentry client.

Inherited members from AgentryException

Modifier and Type	Member	Description
public	*AgentryException(String)* on page 395	Constructs a new AgentryException object.
public	*AgentryException(String, Throwable)* on page 395	Constructs a new AgentryException object.
public	*AgentryException(String, String, String, Throwable)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.
public	*AgentryException(String, String, String)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.
public final String	*getNotificationText()* on page 397	Returns the notification window text.
public final String	*getNotificationTitle()* on page 397	Returns the notification window title.
public final String	*getOkButtonLabel()* on page 397	Returns the notification window button label.

Usage

The user will not be allowed to proceed with their transmission until they change their password via some other means (such as via another system).

This exception should be used instead of PasswordWarningException if the various password-changing methods of User have not been implemented.

PasswordWarningCannotChangeException() constructor

Constructs a new PasswordWarningCannotChangeException object.

Syntax

`public PasswordWarningCannotChangeException ()`

Usage

The client will report a default error message.

PasswordWarningCannotChangeException(String) constructor

Constructs a new PasswordWarningCannotChangeException object.

Syntax

`public PasswordWarningCannotChangeException ( String` *message* `)`

Parameters

- **message –** The error message, which will be displayed on the client.

PasswordWarningCannotChangeException(String, Throwable) constructor

Constructs a new PasswordWarningCannotChangeException object.

Syntax

`public PasswordWarningCannotChangeException ( String` *message* `,`
`Throwable` *cause* `)`

Parameters

- **message –** The error message, which will be displayed on the client.
- **cause –** The exception that triggered this exception.

PasswordWarningException class

This exception can be thrown from the login methods of the Server class to indicate that the user's password is going to expire soon.

Syntax

`public class PasswordWarningException extends LoginException`

Members

All members of PasswordWarningException, including inherited members. **Constructors**

Modifier and Type	Constructor	Description
public	*PasswordWarningException()* on page 448	Constructs a new PasswordWarningException object.
public	*PasswordWarningException(String)* on page 448	Constructs a new PasswordWarningException object.
public	*PasswordWarningException(String, Throwable)* on page 448	Constructs a new PasswordWarningException object.

Inherited members from LoginException

Modifier and Type	Member	Description
public	*LoginException()* on page 434	Constructs a new login exception.
public	*LoginException(String)* on page 434	Constructs a new login exception with the given error message, which will be passed to the Agentry client.
public	*LoginException(String, Throwable)* on page 434	Constructs a new login exception with the given error message, which will be passed to the Agentry client.

Inherited members from AgentryException

Modifier and Type	Member	Description
public	*AgentryException(String)* on page 395	Constructs a new AgentryException object.
public	*AgentryException(String, Throwable)* on page 395	Constructs a new AgentryException object.
public	*AgentryException(String, String, String, Throwable)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.
public	*AgentryException(String, String, String)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.

Modifier and Type	Member	Description
public final String	*getNotificationText()* on page 397	Returns the notification window text.
public final String	*getNotificationTitle()* on page 397	Returns the notification window title.
public final String	*getOkButtonLabel()* on page 397	Returns the notification window button label.

Usage
The user will be given the opportunity to change their password in response to this exception; if they do, the various password-changing methods of the User class will be invoked.

PasswordWarningException() constructor
Constructs a new PasswordWarningException object.

Syntax
```
public   PasswordWarningException ()
```

Usage
The client will report a default error message.

PasswordWarningException(String) constructor
Constructs a new PasswordWarningException object.

Syntax
```
public   PasswordWarningException ( String message )
```

Parameters

- **message –** The error message, which will be displayed on the client.

PasswordWarningException(String, Throwable) constructor
Constructs a new PasswordWarningException object.

Syntax
```
public   PasswordWarningException ( String message ,  Throwable
cause )
```

Parameters

- **message –** The error message, which will be displayed on the client.
- **cause –** The exception that triggered this exception.

PushSession class

The PushSession class encapsulates the user-independent part of the processing involved in a push within an Agentry-based application.

Syntax

```
public class  PushSession extends Session
```

Members

All members of PushSession, including inherited members. **Constructors**

Modifier and Type	Constructor	Description
public	*PushSession(String, Server, SessionData)* on page 452	Construct a new session.

Methods

Modifier and Type	Method	Description
public final void	*beginPushError()* on page 452	Deprecated. As of Agentry 5.2.8, code for this method should be moved into a subclass of PushUserSession. This method is no longer supported; it is here as a final declaration in order to force subclasses to be converted over to using PushUserSession.
public void	*beginPushReadStep()* on page 453	This method is called by the server prior to the execution of the "Object Read" steps for the push.
public void	*beginPushRemoval()* on page 453	This method is called by the server prior to the execution of the "Removal" steps for the push.

Modifier and Type	Method	Description
public final void	*beginPushResponse()* on page 453	Deprecated. As of Agentry 5.2.8, code for this method should be moved into a subclass of PushUserSession. This method is no longer supported; it is here as a final declaration in order to force subclasses to be converted over to using PushUserSession.
public void	*beginPushRetrieval()* on page 454	This method is called by the server prior to the execution of the "Retrieval" steps for the push.
public final void	*endPushError()* on page 454	Deprecated. As of Agentry 5.2.8, code for this method should be moved into a subclass of PushUserSession. This method is no longer supported; it is here as a final declaration in order to force subclasses to be converted over to using PushUserSession.
public void	*endPushReadStep()* on page 454	This method is called by the server after the "Object Read" Steps for the push have been successfully executed.
public void	*endPushRemoval()* on page 455	This method is called by the server after the "Removal" steps for the push have been successfully executed.
public final void	*endPushResponse()* on page 455	Deprecated. As of Agentry 5.2.8, code for this method should be moved into a subclass of PushUserSession. This method is no longer supported; it is here as a final declaration in order to force subclasses to be converted over to using PushUserSession.

Modifier and Type	Method	Description
public void	*endPushRetrieval()* on page 455	This method is called by the server after the "Retrieval" steps for the push have been successfully executed.

Inherited members from Session

Modifier and Type	Member	Description
public final void	*debug(String)* on page 503	Write the given message to a debug log, if debugging is enabled.
public String	*getName()* on page 503	Returns the name of the session, as configured in the Agentry application.
public Server	*getServer()* on page 503	Returns the Server singleton object that the Java system connection is currently using.
public SessionData	*getSessionData()* on page 504	Returns the session data for this session.
public User	*getUser()* on page 504	Returns the user for this session, if any.
protected	*Session(String, Server, SessionData, User)* on page 502	Construct a new session, and save each of the arguments so that they can be retrieved later via the getName(), getServer(), getSessionData(), and getUser() methods.
protected	*Session(String, Server, SessionData)* on page 502	Construct a new session, and save each of the arguments so that they can be retrieved later via the getName(), getServer(), and getSessionData() methods.
public void	*sessionAborted()* on page 504	This is called if the session is aborted (e.g., by an exception).

Usage

In brief, a push is an application component that defines the transfer of data between the client and the server. Unlike fetches, the transfer within a push is initiated by the server, rather than the client. A push is made up of steps, grouped in varying categories. The steps in each of these groups are run separately. The groups are "Retrieval", "Removal", "Object Read", "Response" and "Error"; this class handles the "Retrieval", "Removal", and "Object Read" step groups, as those are independent of users. The remaining groups, "Response" and "Error", are user-dependent and are handled by the PushUserSession class.

This class contains methods to perform actions before and after each of these groups of steps. The default implementation of these methods in the AJAPI perform no additional specific actions. A designer can extend this class if special processing is required before or after each of these groups of steps are processed. If this class is extended, the Server class must also be extended and its `createPushSession` method must be overridden to return the new subclass.

PushSession(String, Server, SessionData) constructor

Construct a new session.

Syntax

`public PushSession ( String` *pushName* `, Server` *server* `, SessionData` *sessionData* `)`

Parameters

- **pushName –** The fetch name, as configured in the Agentry application.
- **server –** The `Server` object that the Java System connection was configured to use.
- **sessionData –** Session data for this fetch.

Usage

This constructor is called by the Server.createPushSession method. Subclasses should implement a constructor with the same signature as this one.

beginPushError() method [deprecated]

Deprecated. As of Agentry 5.2.8, code for this method should be moved into a subclass of PushUserSession. This method is no longer supported; it is here as a final declaration in order to force subclasses to be converted over to using PushUserSession.

Syntax

`public final void beginPushError ()`

Usage

DeprecatedAs of Agentry 5.2.8, code for this method should be moved into a subclass of PushUserSession.

beginPushReadStep() method
This method is called by the server prior to the execution of the "Object Read" steps for the push.

Syntax
`public void beginPushReadStep ()` throws `AgentryException`

Exceptions

- **AgentryException class –** if the push should be aborted for some reason. Throwing an exception from this method will prevent the object read steps from being executed on other system connections as well, and will cause sessionAborted to be invoked.

Usage
Any processing that should take place prior to the execution of these steps should be implemented in this method.

beginPushRemoval() method
This method is called by the server prior to the execution of the "Removal" steps for the push.

Syntax
`public void beginPushRemoval ()` throws `AgentryException`

Exceptions

- **AgentryException class –** if the push should be aborted for some reason. Throwing an exception from this method will prevent the removal steps from being executed on other system connections as well, and will cause sessionAborted to be invoked.

Usage
Any processing that should take place prior to the execution of these steps should be implemented in this method.

beginPushResponse() method [deprecated]
Deprecated. As of Agentry 5.2.8, code for this method should be moved into a subclass of PushUserSession. This method is no longer supported; it is here as a final declaration in order to force subclasses to be converted over to using PushUserSession.

Syntax
`public final void beginPushResponse ()`

Usage
DeprecatedAs of Agentry 5.2.8, code for this method should be moved into a subclass of PushUserSession.

beginPushRetrieval() method

This method is called by the server prior to the execution of the "Retrieval" steps for the push.

Syntax

`public void beginPushRetrieval ()` throws `AgentryException`

Exceptions

- **AgentryException class –** if the push should be aborted for some reason. Throwing an exception from this method will prevent the retrieval steps from being executed on other system connections as well, and will cause sessionAborted to be invoked.

Usage

Any processing that should take place prior to the execution of these steps should be implemented in this method.

endPushError() method [deprecated]

Deprecated. As of Agentry 5.2.8, code for this method should be moved into a subclass of PushUserSession. This method is no longer supported; it is here as a final declaration in order to force subclasses to be converted over to using PushUserSession.

Syntax

`public final void endPushError ()`

Usage

DeprecatedAs of Agentry 5.2.8, code for this method should be moved into a subclass of PushUserSession.

endPushReadStep() method

This method is called by the server after the "Object Read" Steps for the push have been successfully executed.

Syntax

`public void endPushReadStep ()`

Usage

Any processing that should take place prior to the execution of these steps should be implemented in this method.

endPushRemoval() method
This method is called by the server after the "Removal" steps for the push have been successfully executed.

Syntax
```
public void endPushRemoval ()
```

Usage
Any processing that should take place prior to the execution of these steps should be implemented in this method.

endPushResponse() method [deprecated]
Deprecated. As of Agentry 5.2.8, code for this method should be moved into a subclass of PushUserSession. This method is no longer supported; it is here as a final declaration in order to force subclasses to be converted over to using PushUserSession.

Syntax
```
public final void endPushResponse ()
```

Usage
DeprecatedAs of Agentry 5.2.8, code for this method should be moved into a subclass of PushUserSession.

endPushRetrieval() method
This method is called by the server after the "Retrieval" steps for the push have been successfully executed.

Syntax
```
public void endPushRetrieval ()
```

Usage
Any processing that should take place prior to the execution of these steps should be implemented in this method.

PushUserSession class
The PushSession class encapsulates the user-independent part of the processing involved in a push within an Agentry-based application.

Syntax
```
public class PushUserSession extends Session
```

Members
All members of PushUserSession, including inherited members. **Constructors**

Modifier and Type	Constructor	Description
public	*PushUserSession(String, Server, SessionData, User)* on page 458	Construct a new session.

Methods

Modifier and Type	Method	Description
public void	*beginDisablePush()* on page 459	This method is called by the server prior to disabling a user push on any of the system connections.
public void	*beginEnablePush()* on page 459	This method is called by the server prior to enabling a user push on any of the system connections.
public void	*beginPushError()* on page 459	This method is called by the server prior to the execution of the "Error Steps" for the push.
public void	*beginPushResponse()* on page 460	This method is called by the server prior to the execution of the "Response Steps" for the push.
public void	*disablePush()* on page 460	This method is called when a user requests that a push be disabled on their behalf.
public void	*enablePush()* on page 460	This method is called when a user requests that a push be enabled on their behalf.
public void	*endDisablePush()* on page 460	This method is called after a user push has been disabled on all of the system connections.
public void	*endEnablePush()* on page 461	This method is called after a user push has been enabled on all of the system connections.

Modifier and Type	Method	Description
public void	*endPushError()* on page 461	This method is called by the server after the "Error Steps" for the push have been successfully executed.
public void	*endPushResponse()* on page 461	This method is called by the server after the "Response Steps" for the push have been successfully executed.

Inherited members from Session

Modifier and Type	Member	Description
public final void	*debug(String)* on page 503	Write the given message to a debug log, if debugging is enabled.
public String	*getName()* on page 503	Returns the name of the session, as configured in the Agentry application.
public Server	*getServer()* on page 503	Returns the Server singleton object that the Java system connection is currently using.
public SessionData	*getSessionData()* on page 504	Returns the session data for this session.
public User	*getUser()* on page 504	Returns the user for this session, if any.
protected	*Session(String, Server, SessionData, User)* on page 502	Construct a new session, and save each of the arguments so that they can be retrieved later via the getName(), getServer(), getSessionData(), and getUser() methods.
protected	*Session(String, Server, SessionData)* on page 502	Construct a new session, and save each of the arguments so that they can be retrieved later via the getName(), getServer(), and getSessionData() methods.

Modifier and Type	Member	Description
public void	*sessionAborted()* on page 504	This is called if the session is aborted (e.g., by an exception).

Usage

In brief, a push is an application component that defines the transfer of data between the client and the server. Unlike fetches, the transfer within a push is initiated by the server, rather than the client. A push is made up of steps, grouped in varying categories. The steps in each of these groups are run separately. The groups are "Retrieval", "Removal", "Object Read", "Response" and "Error"; this class handles the "Response" and "Error" groups, which are user-dependent.

This class contains methods to perform actions before and after each of these groups of steps. The default implementation of these methods in the AJ-API perform no additional specific actions. A designer can extend this class if special processing is required before or after each of these groups of steps are processed. If this class is extended, the Server class must also be extended and its `createPushUserSession` method must be overridden to return the new subclass.

This class also provides methods for notifying the Agentry server when it should enable or disable push events for a particular user.

PushUserSession(String, Server, SessionData, User) constructor

Construct a new session.

Syntax

`public PushUserSession ( String` *pushName* `, Server` *server* `, SessionData` *sessionData* `, User` *user* `)`

Parameters

- **pushName –** The fetch name, as configured in the Agentry application.
- **server –** The `Server` object that the Java System connection was configured to use.
- **sessionData –** Session data for this fetch.
- **user –** The client user performing the fetch.

Usage

This constructor is called by the Server.createPushUserSession method. Subclasses should implement a constructor with the same signature as this one.

beginDisablePush() method

This method is called by the server prior to disabling a user push on any of the system connections.

Syntax

`public void beginDisablePush ()` throws `AgentryException`

Exceptions

- **AgentryException class –** to block disabling of the user push.

Usage

It can be used to start a remote transaction.

beginEnablePush() method

This method is called by the server prior to enabling a user push on any of the system connections.

Syntax

`public void beginEnablePush ()` throws `AgentryException`

Exceptions

- **AgentryException class –** if an error occurs.

Usage

It can be used to start a remote transaction.

beginPushError() method

This method is called by the server prior to the execution of the "Error Steps" for the push.

Syntax

`public void beginPushError ()` throws `AgentryException`

Exceptions

- **AgentryException class –** if the push should be aborted for some reason. Throwing an exception from this method will prevent the response steps from being executed on other system connections as well, and will cause sessionAborted to be invoked.

Usage

Any processing that should take place prior to the execution of these steps should be implemented in this method.

beginPushResponse() method

This method is called by the server prior to the execution of the "Response Steps" for the push.

Syntax

`public void beginPushResponse ()` throws `AgentryException`

Exceptions

- **AgentryException class –** if the push should be aborted for some reason. Throwing an exception from this method will prevent the response steps from being executed on other system connections as well, and will cause sessionAborted to be invoked.

Usage

Any processing that should take place prior to the execution of these steps should be implemented in this method.

disablePush() method

This method is called when a user requests that a push be disabled on their behalf.

Syntax

`public void disablePush ()` throws `AgentryException`

Exceptions

- **AgentryException class –** if an error occurs.

enablePush() method

This method is called when a user requests that a push be enabled on their behalf.

Syntax

`public void enablePush ()` throws `AgentryException`

Exceptions

- **AgentryException class –** if an error occurs.

endDisablePush() method

This method is called after a user push has been disabled on all of the system connections.

Syntax

`public void endDisablePush ()`

Usage

It can be used to commit a remote transaction. It cannot fail.

endEnablePush() method

This method is called after a user push has been enabled on all of the system connections.

Syntax

```
public  void  endEnablePush ()
```

Usage

It can be used to commit a remote transaction. It cannot fail.

endPushError() method

This method is called by the server after the "Error Steps" for the push have been successfully executed.

Syntax

```
public  void  endPushError ()
```

Usage

Any processing that should take place prior to the execution of these steps should be implemented in this method.

endPushResponse() method

This method is called by the server after the "Response Steps" for the push have been successfully executed.

Syntax

```
public  void  endPushResponse ()
```

Usage

Any processing that should take place prior to the execution of these steps should be implemented in this method.

RetryTransactionException class

This exception can be thrown from a transactional Steplet to indicate that the transaction failed temporarily, and that it should be retried by the client.

Syntax

```
public class  RetryTransactionException extends
AgentryException
```

Members

All members of RetryTransactionException, including inherited members. **Constructors**

Modifier and Type	Constructor	Description
public	*RetryTransactionException(String, String, String)* on page 463	Constructs a new RetryTransactionException object.
public	*RetryTransactionException(String, String, String, Throwable)* on page 463	Constructs a new RetryTransactionException object.

Inherited members from AgentryException

Modifier and Type	Member	Description
public	*AgentryException(String)* on page 395	Constructs a new AgentryException object.
public	*AgentryException(String, Throwable)* on page 395	Constructs a new AgentryException object.
public	*AgentryException(String, String, String, Throwable)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.
public	*AgentryException(String, String, String)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.
public final String	*getNotificationText()* on page 397	Returns the notification window text.
public final String	*getNotificationTitle()* on page 397	Returns the notification window title.
public final String	*getOkButtonLabel()* on page 397	Returns the notification window button label.

Usage

The client will resend the transaction immediately, and if it fails a second time then the transaction will be marked as fatal.

This exception is equivalent to the "Retry Without Change" transaction error fatality.

RetryTransactionException(String, String, String) constructor

Constructs a new RetryTransactionException object.

Syntax

`public RetryTransactionException ( String` *title* `, String` *notification* `, String` *okButtonLabel* `)`

Parameters

- **title –** The window title for the notification displayed on the client.
- **notification –** The window text for the notification displayed on the client.
- **okButtonLabel –** The label for the acknowledgment button in the client's notification window.

RetryTransactionException(String, String, String, Throwable) constructor

Constructs a new RetryTransactionException object.

Syntax

`public RetryTransactionException ( String` *title* `, String` *text* `, String` *okLabel* `, Throwable` *cause* `)`

Parameters

- **title –** The window title for the error displayed on the client.
- **text –** The window text for the error displayed on the client.
- **okLabel –** The label for the acknowledgment button in the client's error window.
- **cause –** The causing exception

RetryTransactionWithChangeException class

This exception can be thrown from a transactional Steplet to indicate that the transaction failed in a correctable manner, and that it should be retried by the client after prompting the user to make changes to the transaction.

Syntax

```
public class RetryTransactionWithChangeException extends
AgentryException
```

Members

All members of RetryTransactionWithChangeException, including inherited members.
Constructors

Modifier and Type	Constructor	Description
public	*RetryTransactionWithChangeException(String, String, String)* on page 465	Constructs a new RetryTransactionWithChangeException object.
public	*RetryTransactionWithChangeException(String, String, String, Throwable)* on page 465	Constructs a new RetryTransactionWithChangeException object.

Inherited members from AgentryException

Modifier and Type	Member	Description
public	*AgentryException(String)* on page 395	Constructs a new AgentryException object.
public	*AgentryException(String, Throwable)* on page 395	Constructs a new AgentryException object.
public	*AgentryException(String, String, String, Throwable)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.
public	*AgentryException(String, String, String)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.
public final String	*getNotificationText()* on page 397	Returns the notification window text.
public final String	*getNotificationTitle()* on page 397	Returns the notification window title.
public final String	*getOkButtonLabel()* on page 397	Returns the notification window button label.

Usage

This exception is equivalent to the "Retry With Change" transaction error fatality.

RetryTransactionWithChangeException(String, String, String) constructor

Constructs a new RetryTransactionWithChangeException object.

Syntax

`public RetryTransactionWithChangeException ( String` *title* `, String` *text* `, String` *okLabel* `)`

Parameters

- **title –** The window title for the notification displayed on the client.
- **text –** The text for the notification window displayed on the client, which should describe to the user what changes they need to make in order for the transaction to succeed.
- **okLabel –** The label for the acknowledgment button in the client's notification window.

RetryTransactionWithChangeException(String, String, String, Throwable) constructor

Constructs a new RetryTransactionWithChangeException object.

Syntax

`public RetryTransactionWithChangeException ( String` *title* `, String` *text* `, String` *okLabel* `, Throwable` *cause* `)`

Parameters

- **title –** The window title for the error displayed on the client.
- **text –** The window text for the error displayed on the client.
- **okLabel –** The label for the acknowledgment button in the client's error window.
- **cause –** The causing exception

Server class

The Server Java class is intended to encapsulate the Java system connection within the Agentry Server.

Syntax

```
public class  Server extends AgentryJavaBackEndManagedObject
```

Members

All members of Server, including inherited members. **Constructors**

Modifier and Type	Constructor	Description
public	*Server()* on page 476	Constructs a new Server object.

Methods

Modifier and Type	Method	Description
public ComplexTableSession	*createComplexTableSession(String, SessionData, User)* on page 476	Factory method that creates a new ComplexTableSession object.
public DataTableSession	*createDataTableSession(String, SessionData, User)* on page 477	Factory method that creates a new DataTableSession object.
public final FetchSession	*createFetchSession(String, Server, SessionData, User)* on page 478	Deprecated. Use createFetchSession(String, SessionData, User) (i.e. remove the Server argument). This method is no longer supported; override createFetchSession(String, SessionData, User) instead.
public FetchSession	*createFetchSession(String, SessionData, User)* on page 478	Factory method that creates a new FetchSession object.
public final FetchSession	*createPushSession(String, Server, SessionData)* on page 479	Deprecated. Use createPushSession(String, SessionData) (i.e. remove the Server argument). This method is no longer supported; override createPushSession(String, SessionData) instead.
public PushSession	*createPushSession(String, SessionData)* on page 479	Factory method that creates a new PushSession object for a push session that is not tied to a specific user.
public final FetchSession	*createPushUserSession(String, Server, SessionData, User)* on page 480	Deprecated. Use createPushUserSession(String, SessionData, User) (i.e. remove the Server argument). This method is no longer supported; override createPushUserSession(String, SessionData, User) instead.
public PushUserSession	*createPushUserSession(String, SessionData, User)* on page 480	Factory method that creates a new PushUserSession object for a push session.

Modifier and Type	Method	Description
public final ServiceEventSession	*createServiceEventSession(String, Server, SessionData)* on page 481	Deprecated. Use createServiceEventSession(String, SessionData) (i.e. remove the Server argument). This method is no longer supported; override createServiceEventSession(String, SessionData) instead.
public ServiceEventSession	*createServiceEventSession(String, SessionData)* on page 482	Factory method that creates a new ServiceEventSession object.
public final FetchSession	*createTransactionSession(String, Server, SessionData, User)* on page 482	Deprecated. Use createTransactionSession(String, SessionData, User) (i.e. remove the Server argument). This method is no longer supported; override createTransactionSession(String, SessionData, User) instead.
public TransactionSession	*createTransactionSession(String, SessionData, User)* on page 483	Factory method that creates a new TransactionSession object.
public final User	*createUser(String, int)* on page 483	Deprecated. Use createUser(String) instead. This method is no longer supported; override createUser(String) instead.
public User	*createUser(String)* on page 484	Factory method that creates a new User object.
public final void	*debug(String)* on page 484	Writes a debug message to the Agentry Server's Java System Connection log file.
public final String	*decryptPassword(String)* on page 485	Decodes a password that has been encoded via Agentry's quickPW utility.
public File	*findConfigurationFile(String)* on page 485	Locates a configuration file in the Agentry application's deployment returns a.

Modifier and Type	Method	Description
public static final String	*getImplementationVersion()* on page 485	Retrieves the implementation version of the AJAPI release that this Server class is from.
public static Server	*getInstance()* on page 486	Return the singleton instance of this class.
public static final String	*getSpecificationVersion()* on page 486	Retrieves the specification version of the AJAPI that this Server implements.
public String	*getTimeZone()* on page 486	This is called by the Agentry server to find out what time zone is being used by whatever remote server this implementation is communicating with.
public LoginEnumeration	*login(String, String, SessionData)* on page 487	Deprecated. Override login(User, String, SessionData) instead. This method is called when a user initially connects to the Agentry Server from a client application and that server's system connection's enableAuthentication option is set to true in the Agentry.ini file.
public void	*login(User, String, SessionData)* on page 487	This method authenticates a client user against the Java System Connection.

Modifier and Type	Method	Description
public LoginEnumeration	*loginBlocked(String, StringBuffer)* on page 488	Deprecated. Override loginBlocked(User, String, StringBuffer, SessionData) instead. This method is called by the Agentry Server when authentication of a client user is blocked, either because this class returned a blocked login from the login(String, String, SessionData) or loginPreviousUser(String, String, SessionData) methods, or because another system connection blocked the login.
public void	*loginBlocked(User, StringBuffer, SessionData)* on page 489	Deprecated. Override loginBlocked(User, String, StringBuffer, SessionData) instead. This method is called by the Agentry Server when authentication of a client user is blocked, either because this class threw LoginBlockedException from the login, loginPreviousUser, or loginFailed methods, or because another system connection blocked the login.
public void	*loginBlocked(User, String, StringBuffer, SessionData)* on page 489	This method is called by the Agentry Server when authentication of a client user is blocked, either because this class threw LoginBlockedException from the login, loginPreviousUser, or loginFailed methods, or because another system connection blocked the login.

Modifier and Type	Method	Description
public LoginEnumeration	*loginFailed(String, StringBuffer)* on page 490	Deprecated. Override loginFailed(User, String, LoginFailureReason, StringBuffer, SessionData) instead. This method is called by the Agentry Server when authentication of a client user fails, either because this class returned a failed login from the login or loginPreviousUser methods, or because another system connection failed the login.
public void	*loginFailed(User, String, LoginFailureReason, StringBuffer, SessionData)* on page 491	This method is called by the Agentry Server when authentication of a client user fails, either because this class threw PasswordInvalidException from the login or loginPreviousUser methods, or because another system connection reported a login failure.
public LoginEnumeration	*loginPreviousUser(String, String, SessionData)* on page 492	Deprecated. Override loginPreviousUser(User, String, SessionData) instead. This method is called when a user has previously logged into Agentry successfully, and is now logging in again due to having been disconnected.
public void	*loginPreviousUser(User, String, SessionData)* on page 492	This method is called when a user has previously logged into Agentry successfully, and is now logging in again due to having been disconnected.
final void	*setDebugEnabled(boolean)* on page 493	Deprecated. This is only here because the Agentry server will call it. Setter method called by the Agentry server to enable/disable debugging.

Modifier and Type	Method	Description
public void	*shutdown()* on page 493	This method is called by the Agentry Server when the Java system connection is being shut down.
public void	*startup()* on page 493	This method is called by the Agentry Server when the Java system connection starts up and creates an instance of this class; it is called immediately after the class is constructed.

Usage

The bulk of the methods within this class are factory methods for various other object types. If the designer overrides one of the classes constructed by these factory methods, a subclass of the `Server` class must also be created. This implementation must override the appropriate factory methods to construct objects of the appropriate type.

In addition to these factory methods, there are also methods related to login and logout, server startup and shutdown, and debugging. By default, these methods perform no application-related processing; they merely print a message to the debug log indicating that these events have or are about to occur. If additional processing is required for an application, these methods should be overridden in a subclass of the `Server` class.

When an Agentry Server with a Java System Connection is started, the server will construct a singleton instance of the `Server` class or a subclass of it, as specified by the `serverClass` setting in the Java system connection section of the `Agentry.ini` file. This `Server` object will persist until the Agentry Server is shutdown. If the designer has implemented an extension of the `Server` class, this new class must be named in the `serverClass` option in the `Agentry.ini` file in the section containing the configuration options for the Java Interface system connection.

Note, the constructor is public even though this class is a singleton. This is mainly for legacy reasons: the `Agentry.ini` file names a subclass of this class instead of naming a factory class, and we kept it that way rather than changing it to take a factory class name to maintain compatibility with AJAPI version 4 (since the server supports both versions of the AJAPI).

As such, in this version of the AJAPI, this class is not a strict singleton, although it should be treated as such. Applications should not try to create another instance of it, although it is acceptable (since it is hard to avoid) to do so in unit tests.

Eventually, this class will likely become a true singleton and the Agentry server will adopt the factory pattern to instantiate it, so you may wish to do so now in unit tests using a factory similar to:

```
public class ServerFactory
{
       private static class LazyHolder
       {
               private static final Server _instance = new Server();
       }

       public static Server getServerInstance()
       {
               return LazyHolder._instance;
       }
}
```

(Note, this example follows the "Initialization on Demand Holder" pattern; see *this article on Wikipedia* for information on how/why it works.)

Server.LoginEnumeration enum

Deprecated. These constants are only used by the deprecated login methods. New code should be fixed to use the new exception-based login methods. Return values for the login method.

Members

All members of LoginEnumeration, including inherited members. **Variables**

Modifier and Type	Variable	Description
public	*Login_Invalid* on page 473	Indicates that the login and password are not valid for any user profile on the system or that the login attempt failed for some reason other than the user being blocked, or the user having an expired password.
public	*Login_InvalidBlocked* on page 474	Indicates that the user has been blocked from accessing the remote system.
public	*Login_Pass* on page 474	Indicates that the user was not validated by the system connection.
public	*Login_ Valid* on page 474	Indicates that the user ID and password are valid and that the user should be allowed to proceed with their transmission to the Agentry Server.

Modifier and Type	Variable	Description
public	*Login_ ValidPasswordExpired* on page 474	Indicates that the user ID is valid but that the password has expired.
public	*Login_ ValidPasswordExpired-NoChange* on page 474	Indicates that the user ID is valid, but that the password has expired.
public	*Login_ ValidPasswordWarning* on page 475	Indicates that user ID and password values are valid, but that the password will be expiring in the near future.
public	*Login_ ValidPasswordWarning-NoChange* on page 475	Indicates that the user ID and password values are valid, but that the password will be expiring in the near future.

Methods

Modifier and Type	Method	Description
public void	*throwException()* on page 473	Throws an exception that corresponds to this enum value.

throwException() method

Throws an exception that corresponds to this enum value.

Syntax

```
public void throwException () throws LoginException
```

Exceptions

- **LoginException class –** the exception that corresponds to this enum.

Login_Invalid variable

Indicates that the login and password are not valid for any user profile on the system or that the login attempt failed for some reason other than the user being blocked, or the user having an expired password.

Syntax

```
public  Login_Invalid
```

Login_InvalidBlocked variable

Indicates that the user has been blocked from accessing the remote system.

Syntax

```
public   Login_InvalidBlocked
```

Login_Pass variable

Indicates that the user was not validated by the system connection.

Syntax

```
public   Login_Pass
```

Usage

Another system connection must validate the user before the user will be allowed to access the Agentry Server from the client application. Returning this value has the same effect as setting `enableAuthentication` to `false` in the Java section of `Agentry.ini`, except that it can be used on a per-user basis.

Login_Valid variable

Indicates that the user ID and password are valid and that the user should be allowed to proceed with their transmission to the Agentry Server.

Syntax

```
public   Login_Valid
```

Login_ValidPasswordExpired variable

Indicates that the user ID is valid but that the password has expired.

Syntax

```
public   Login_ValidPasswordExpired
```

Usage

The user will be prompted to change their password on the client application before being allowed to proceed with the transmission.

Login_ValidPasswordExpiredNoChange variable

Indicates that the user ID is valid, but that the password has expired.

Syntax

```
public   Login_ValidPasswordExpiredNoChange
```

Usage

The user will not be allowed to change the password from within the Agentry-based application and will not be allowed to proceed with their transmission until the password has been updated.

Login_ValidPasswordWarning variable

Indicates that user ID and password values are valid, but that the password will be expiring in the near future.

Syntax

```
public   Login_ValidPasswordWarning
```

Usage

The user will be prompted to change his or her password on the client application, but can bypass the change and still be allowed to proceed with the transmission to the Agentry Server.

Login_ValidPasswordWarningNoChange variable

Indicates that the user ID and password values are valid, but that the password will be expiring in the near future.

Syntax

```
public   Login_ValidPasswordWarningNoChange
```

Usage

The user will be notified that their password is near its expiration, but will not be allowed to modify the password value from within the Agentry-based application.

Server.LoginFailureReason enum

These are used in loginFailed to indicate the reason why a login failed.

Members

All members of LoginFailureReason, including inherited members. **Variables**

Modifier and Type	Variable	Description
public	*NoBackEndsAuthenticated* on page 476	Indicates that the login failed because no back-ends were configured to authenticate.
public	*PasswordExpiredCannotChange* on page 476	Indicates that the login failed because a password was expired and Agentry cannot change it.

Modifier and Type	Variable	Description
public	*PasswordInvalid* on page 476	Indicates that the login failed due to an invalid password.

NoBackEndsAuthenticated variable

Indicates that the login failed because no back-ends were configured to authenticate.

Syntax

```
public   NoBackEndsAuthenticated
```

PasswordExpiredCannotChange variable

Indicates that the login failed because a password was expired and Agentry cannot change it.

Syntax

```
public   PasswordExpiredCannotChange
```

PasswordInvalid variable

Indicates that the login failed due to an invalid password.

Syntax

```
public   PasswordInvalid
```

Server() constructor

Constructs a new Server object.

Syntax

```
public   Server ()
```

Usage

This constructor is called by the Agentry server when the Java Interface system connection is started.

See the main class documentation for notes on why this constructor is public even though this class is a singleton.

createComplexTableSession(String, SessionData, User) method

Factory method that creates a new ComplexTableSession object.

Syntax

```
public  ComplexTableSession  createComplexTableSession ( String
tableName ,  SessionData sessionData ,  User user )
```

Parameters

- **tableName –** The name of the complex table being processed, as configured by the designer in the Agentry Editor.
- **sessionData –** Provides access to current session data, such as Agentry Server Data Markup Language (SDML) values.
- **user –** Represents the client user that is retrieving the complex table.

Returns
A new ComplexTableSession object.

Usage
This method is called by the Agentry Server whenever a complex table synchronization is to be processed. If the ComplexTableSession class is extended, then this method must be overridden to return the new subclass.

createDataTableSession(String, SessionData, User) method
Factory method that creates a new DataTableSession object.

Syntax
`public DataTableSession createDataTableSession(String` *tableName*`, SessionData` *sessionData*`, User` *user*`)`

Parameters

- **tableName –** The name of the data table being processed, as configured by the designer in the Agentry Editor.
- **sessionData –** Provides access to current session data, such as Agentry Server Data Markup Language (SDML) values.
- **user –** Represents the client user that is retrieving the complex table.

Returns
A new DataTableSession object.

Usage
This method is called by the Agentry Server whenever a data table retrieval is to be processed. If the DataTableSession class is extended, then this method must be overridden to return the new subclass.

createFetchSession(String, Server, SessionData, User) method [deprecated]

Deprecated. Use createFetchSession(String, SessionData, User) (i.e. remove the Server argument). This method is no longer supported; override createFetchSession(String, SessionData, User) instead.

Syntax

```
public final FetchSession createFetchSession ( String name ,
Server server , SessionData data , User user )
```

Parameters

- **name –** not used
- **server –** not used
- **data –** not used
- **user –** not used

Returns

nothing, throws `UnsupportedOperationException`.

Exceptions

- **UnsupportedOperationException –** to indicate that it is no longer supported.

Usage

This method is only here to cause compilation errors in legacy code, and may be removed in a future release.

createFetchSession(String, SessionData, User) method

Factory method that creates a new FetchSession object.

Syntax

```
public FetchSession createFetchSession ( String fetchName ,
SessionData sessionData , User user )
```

Parameters

- **fetchName –** The name of the fetch being processed, as configured by the designer in the Agentry Editor.
- **sessionData –** Provides access to current session data, such as Agentry Server Data Markup Language (SDML) values.
- **user –** Represents the client user that is performing the fetch.

Returns

A new FetchSession object.

Usage

It is called by the Agentry Server whenever a fetch is requested by the client application. If the FetchSession class is extended, then this method must be overridden to return the new subclass.

createPushSession(String, Server, SessionData) method [deprecated]

Deprecated. Use createPushSession(String, SessionData) (i.e. remove the Server argument). This method is no longer supported; override createPushSession(String, SessionData) instead.

Syntax

`public final FetchSession createPushSession ( String` *name* `, Server` *server* `, SessionData` *data* `)`

Parameters

- **name –** not used
- **server –** not used
- **data –** not used

Returns

nothing, throws `UnsupportedOperationException`.

Exceptions

- **UnsupportedOperationException –** to indicate that it is no longer supported.

Usage

This method is only here to cause compilation errors in legacy code, and may be removed in a future release.

createPushSession(String, SessionData) method

Factory method that creates a new PushSession object for a push session that is not tied to a specific user.

Syntax

`public PushSession createPushSession ( String` *pushName* `, SessionData` *sessionData* `)`

Parameters

- **pushName –** The name of the push being processed, as configured by the designer in the Agentry Editor.

- **sessionData –** Provides access to current session data, such as Agentry Server Data Markup Language (SDML) values.

Returns
A new PushSession object.

Usage
It is called by the Agentry Server whenever a push is being processed by the server. If the PushSession class is extended, then this method must be overridden to return the new subclass.

createPushUserSession(String, Server, SessionData, User) method [deprecated]
Deprecated. Use createPushUserSession(String, SessionData, User) (i.e. remove the Server argument). This method is no longer supported; override createPushUserSession(String, SessionData, User) instead.

Syntax
`public final FetchSession createPushUserSession ( String` *name* `, Server` *server* `, SessionData` *data* `, User` *user* `)`

Parameters

- **name –** not used
- **server –** not used
- **data –** not used
- **user –** not used

Returns
nothing, throws `UnsupportedOperationException`.

Exceptions

- **UnsupportedOperationException –** to indicate that it is no longer supported.

Usage
This method is only here to cause compilation errors in legacy code, and may be removed in a future release.

createPushUserSession(String, SessionData, User) method
Factory method that creates a new PushUserSession object for a push session.

Syntax
`public PushUserSession createPushUserSession ( String` *pushName* `, SessionData` *sessionData* `, User` *user* `)`

Parameters

- **pushName –** The name of the push being processed, as configured by the designer in the Agentry Editor.
- **sessionData –** Provides access to current session data, such as Agentry Server Data Markup Language (SDML) values.
- **user –** Represents the client user that is performing the push.

Returns
A new PushUserSession object.

Usage
It is called by the Agentry Server whenever the user-specific steps of a push are being processed by the server. If the PushUserSession class is extended, then this method must be overridden to return the new subclass.

createServiceEventSession(String, Server, SessionData) method [deprecated]

Deprecated. Use createServiceEventSession(String, SessionData) (i.e. remove the Server argument). This method is no longer supported; override createServiceEventSession(String, SessionData) instead.

Syntax
```
public final ServiceEventSession createServiceEventSession
( String serviceName , Server server , SessionData sessionData )
```

Parameters

- **serviceName –** not used
- **server –** not used
- **sessionData –** not used

Returns
nothing, throws `UnsupportedOperationException`.

Exceptions

- **UnsupportedOperationException –** to indicate that it is no longer supported.

Usage
This method is only here to cause compilation errors in legacy code, and may be removed in a future release.

createServiceEventSession(String, SessionData) method

Factory method that creates a new ServiceEventSession object.

Syntax

`public ServiceEventSession createServiceEventSession ( String` *serviceName* `, SessionData` *sessionData* `)`

Parameters

- **serviceName –** The name of the service event being processed, as configured by the designer in the Agentry Editor.
- **sessionData –** Provides access to current session data, such as Agentry Server Data Markup Language (SDML) values.

Returns

A new ServiceEventSession object.

Usage

This method is called by the Agentry Server whenever a service event is to be processed. If the ServiceEventSession class is extended, then this method must be overridden to return the new subclass.

createTransactionSession(String, Server, SessionData, User) method [deprecated]

Deprecated. Use createTransactionSession(String, SessionData, User) (i.e. remove the Server argument). This method is no longer supported; override createTransactionSession(String, SessionData, User) instead.

Syntax

`public final FetchSession createTransactionSession ( String` *name* `, Server` *server* `, SessionData` *data* `, User` *user* `)`

Parameters

- **name –** not used
- **server –** not used
- **data –** not used
- **user –** not used

Returns

nothing, throws `UnsupportedOperationException`.

Exceptions

- **UnsupportedOperationException –** to indicate that it is no longer supported.

Usage
This method is only here to cause compilation errors in legacy code, and may be removed in a future release.

createTransactionSession(String, SessionData, User) method
Factory method that creates a new TransactionSession object.

Syntax
`public TransactionSession createTransactionSession ( String` *transactionName* `, SessionData` *sessionData* `, User` *user* `)`

Parameters

- **transactionName –** The name of the transaction being processed, as configured by the designer in the Agentry Editor.
- **sessionData –** Provides access to current session data, such as Agentry Server Data Markup Language (SDML) values.
- **user –** Represents the client user that is performing the transaction.

Returns
A new TransactionSession object.

Usage
This method is called by the Agentry Server whenever a transaction is to be processed. If the TransactionSession class is extended, then this method must be overridden to return the new subclass.

createUser(String, int) method [deprecated]
Deprecated. Use createUser(String) instead. This method is no longer supported; override createUser(String) instead.

Syntax
`public final User createUser ( String` *name* `, int` *x* `)`

Parameters

- **name –** not used
- **x –** not used

Returns
nothing, throws `UnsupportedOperationException`.

Exceptions

- **UnsupportedOperationException –** to indicate that it is no longer supported.

Usage
This method is only here to cause compilation errors in legacy code, and may be removed in a future release.

createUser(String) method
Factory method that creates a new User object.

Syntax
`public User createUser ( String` *name* `)`

Parameters

- **name –** The user name value from the Agentry client application.

Returns
A new User object.

Usage
This method is called by the Agentry Server to create objects that represent client users. If a new subclass of User is created for an application, then this method must be extended to construct the new subclass.

debug(String) method
Writes a debug message to the Agentry Server's Java System Connection log file.

Syntax
`public final void debug ( String` *serverMessage* `)`

Parameters

- **serverMessage –** The message to log

Usage
This message will only appear in the log file if debug logging is turned on for the Java system connection.

This method is a convenience method that calls into the Java Logging API to do the actual logging, and assumes that Agentry's default Java Logging configuration is in place (which will route log messages back to the Agentry server). It will log to a logger named "com.syclo.agentry.Server", at the `FINE` level (which translates to log detail level 3 in Agentry).

When invoked outside of Agentry (e.g. in unit tests), this will log to the console, as that is Java's normal default logging configuration.

decryptPassword(String) method

Decodes a password that has been encoded via Agentry's quickPW utility.

Syntax

`public final String decryptPassword ( String` *password* `)`

Parameters

- **password –** The encoded password.

Returns

The decoded password.

findConfigurationFile(String) method

Locates a configuration file in the Agentry application's deployment returns a.

Syntax

`public File findConfigurationFile ( String` *filename* `)`

Parameters

- **filename –** the name of the configuration file

Returns

aFile object referencing the file in the correct directory

Usage

File object that can be used to access the file. If the file does not exist, this method will return an object that can be used to create the file in an appropriate location.

getImplementationVersion() method

Retrieves the implementation version of the AJAPI release that this Server class is from.

Syntax

`public static final String getImplementationVersion ()`

Returns

The implementation version, which is the same as the full version (including build number) of the Agentry Server that this class's JAR file came with. This will return the empty string if you are running out of raw class files and not a released AJAPI JAR.

Usage

This is the same as the version of the Agentry Server that the AJAPI JAR file came with.

This information is retrieved from the AJAPI JAR manifest file via the Package class.

Note that this does not necessarily reflect the version of the API that this JAR implements; for example, if this JAR implements version 4 of the AJAPI, but it comes with version 5.0.0.3 of the Agentry server, this will return 5.0.0.3, not 4! If you want to know the AJAPI version that is implemented, call getSpecificationVersion() instead.

getInstance() method

Return the singleton instance of this class.

Syntax

```
public static Server getInstance()
```

Returns

the running instance of this class

Usage

Note that this method will not create the server; you have to call the constructor once to do that (see the constructor docs for the reasons why).

getSpecificationVersion() method

Retrieves the specification version of the AJAPI that this Server implements.

Syntax

```
public static final String getSpecificationVersion()
```

Returns

The AJAPI version that this AJAPI package implements. This will return the empty string if you are running out of raw class files and not a released AJAPI JAR.

Usage

This will tell you what version of the AJAPI you are using, but not what version of Agentry it came with; if you want to know the latter, call getImplementationVersion() instead.

getTimeZone() method

This is called by the Agentry server to find out what time zone is being used by whatever remote server this implementation is communicating with.

Syntax

```
public String getTimeZone()
```

Returns

The timezone name.

Usage
This will override the `timeZoneName` setting in the `Agentry.ini` file.

If this method returns a name that the Agentry server doesn't recognize (and odds are it won't, especially on Windows), then it needs to be mapped in the `[TimeZoneAliases]` section of the `Agentry.ini` file. If this method returns an empty string (the default implementation), then the `timeZoneName` setting in the `Agentry.ini` file will be used.

login(String, String, SessionData) method [deprecated]

Deprecated. Override login(User, String, SessionData) instead. This method is called when a user initially connects to the Agentry Server from a client application and that server's system connection's enableAuthentication option is set to true in the Agentry.ini file.

Syntax
`public LoginEnumeration login ( String` *userId* `, String` *password* `, SessionData` *sessionData* `)`

Parameters

- **userId –** The user ID from the client application for the current user.
- **password –** The password for the current user, as entered on the client application.
- **sessionData –** Provides access to current session data, such as Agentry Server Data Markup Language (SDML) values.

Returns
One of the constants from Server.LoginEnumeration.

Usage
The return value of this function indicates whether or not the user ID and password are valid. By default, this method returns the enumerated value Login_Pass, which means that this system connection is not responsible for authenticating the user. Override this method to implement logic to perform full validation of the user against a remote system.

login(User, String, SessionData) method

This method authenticates a client user against the Java System Connection.

Syntax
`public void login ( User` *user* `, String` *password* `, SessionData` *sessionData* `)` throws `LoginException`

Parameters

- **user –** The User object that identifies the client user. The user name can be read from this object.

- **password –** The password for the current user, as entered on the client application.
- **sessionData –** Provides access to current session data, such as Agentry Server Data Markup Language (SDML) values.

Exceptions

- **LoginException class –** if the login fails for any reason, or if the login succeeds but an exceptional condition exists (such as an expired or soon-to-be expired password).

Usage

This method is called when a user initially connects to the Agentry Server from a client application and the `enableAuthentication` option is set to `true` in the Java section of the `Agentry.ini` file. Override this method to implement logic to perform full validation of the user against a remote system.

This method should return normally if the authentication of the user succeeds. If login fails for any reason, the appropriate LoginException subclass should be thrown. An exception should also be thrown for other conditions such as expired or soon-to-be-expiring passwords. By default, this method throws LoginSkippedException, which means that this system connection is not responsible for authenticating the user (it is equivalent to setting `enableAuthentication` to false in `Agentry.ini`, except that it can be thrown on a per-user basis).

If you throw PasswordInvalidException or LoginBlockedException, then either loginFailed or loginBlocked, respectively, will be called. From those methods you can return a more detailed error message indicating why the login failed.

loginBlocked(String, StringBuffer) method [deprecated]

Deprecated. Override loginBlocked(User, String, StringBuffer, SessionData) instead. This method is called by the Agentry Server when authentication of a client user is blocked, either because this class returned a blocked login from the login(String, String, SessionData) or loginPreviousUser(String, String, SessionData) methods, or because another system connection blocked the login.

Syntax

```
public LoginEnumeration loginBlocked ( String userId ,
StringBuffer error )
```

Parameters

- **userId –** The user ID for the user whose login attempt has failed.
- **error –** A StringBuffer that contains the error message that was returned by the system connection that blocked the login. This message will ultimately be logged on the server and displayed on the client. The error message can be changed by modifying the contents of this buffer.

Returns

Always returns Server.LoginEnumeration#Login_InvalidBlocked

Usage

It should clean up any user-related resources; it can also return additional information about why the login failed.

loginBlocked(User, StringBuffer, SessionData) method [deprecated]

Deprecated. Override loginBlocked(User, String, StringBuffer, SessionData) instead. This method is called by the Agentry Server when authentication of a client user is blocked, either because this class threw LoginBlockedException from the login, loginPreviousUser, or loginFailed methods, or because another system connection blocked the login.

Syntax

`public void loginBlocked ( User` *user* `, StringBuffer` *error* `, SessionData` *sessionData* `)`

Parameters

- **user –** The User object for the user whose login attempt was blocked.
- **error –** A StringBuffer that contains the error message that was returned by the system connection that blocked the login. This message will ultimately be logged on the server and displayed on the client. The error message can be changed by modifying the contents of this buffer.
- **sessionData –** The session data. In addition to its usual contents, this data will contain additional information about which system connection blocked the login. This information will be available as the values for the SDML keys `failed.backend.id` (the system connection number) and `failed.backend.name` (the system connection name, as configured in `Agentry.ini`).

Usage

It should clean up any user-related resources.

loginBlocked(User, String, StringBuffer, SessionData) method

This method is called by the Agentry Server when authentication of a client user is blocked, either because this class threw LoginBlockedException from the login, loginPreviousUser, or loginFailed methods, or because another system connection blocked the login.

Syntax

`public void loginBlocked ( User` *user* `, String` *userId* `, StringBuffer` *error* `, SessionData` *sessionData* `)`

Parameters

- **user –** The User object for the user whose login attempt has blocked. This can be `null`, if the login was blocked by another system connection before the createUser and login methods of this system connection were called.
- **userId –** The user name of the user that was logging in. This parameter might be useful if `user` is `null`, but you still need to take some sort of action for the user for some reason. If `user` is not `null`, then this parameter will be equal to the user name contained in `user`.
- **error –** A StringBuffer that contains the error message that was returned by the system connection that blocked the login. This message will ultimately be logged on the server and displayed on the client. The error message can be changed by modifying the contents of this buffer.
- **sessionData –** The session data. In addition to its usual contents, this data will contain additional information about which system connection blocked the login. This information will be available as the values for the SDML keys `failed.backend.id` (the system connection number) and `failed.backend.name` (the system connection name, as configured in `Agentry.ini`).

Usage
It should clean up any user-related resources.

loginFailed(String, StringBuffer) method [deprecated]
Deprecated. Override loginFailed(User, String, LoginFailureReason, StringBuffer, SessionData) instead. This method is called by the Agentry Server when authentication of a client user fails, either because this class returned a failed login from the login or loginPreviousUser methods, or because another system connection failed the login.

Syntax
```
public  LoginEnumeration  loginFailed ( String userId ,
StringBuffer error )
```

Parameters

- **userId –** The user ID for the user whose login attempt has failed.
- **error –** A StringBuffer that will be written to the user's debug log by the Agentry server when this method returns. It should be used to log useful error information within the user's debug log about why the login failed; it will always be logged regardless of the Agentry server's debug settings.

Returns
Login_Invalid or Login_InvalidBlocked. If the latter is returned, then loginBlocked(String, StringBuffer) will be invoked as well.

Usage
It should clean up any user-related resources; it can also return additional information about why the login failed.

loginFailed(User, String, LoginFailureReason, StringBuffer, SessionData) method
This method is called by the Agentry Server when authentication of a client user fails, either because this class threw PasswordInvalidException from the login or loginPreviousUser methods, or because another system connection reported a login failure.

Syntax
`public void loginFailed ( User` *user* `, String` *userId* `, LoginFailureReason` *reason* `, StringBuffer` *error* `, SessionData` *sessionData* `)` throws `LoginBlockedException`

Parameters

- **user –** The User object for the user whose login attempt has failed. This can be `null`, if the login was failed by another system connection before the createUser and login methods of this system connection were called.
- **userId –** The user name of the user that was logging in. This parameter might be useful if `user` is `null`, but you still need to take some sort of action for the user for some reason. If `user` is not `null`, then this parameter will be equal to the user name contained in `user`.
- **reason –** The reason for the login failure.
- **error –** A StringBuffer that contains the error message that was returned by the system connection that failed the login. This message will ultimately be logged on the server and displayed on the client. The error message can be changed by modifying the contents of this buffer.
- **sessionData –** The session data. In addition to its usual contents, this data will contain additional information about which system connection failed the login, if the failure reason was not NoBackEndsAuthenticated. This information will be available as the values for the SDML keys `failed.backend.id` (the system connection number) and `failed.backend.name` (the system connection name, as configured in `Agentry.ini`).

Exceptions

- **LoginBlockedException class –** if the login failure should be treated as a blocked login instead. This will trigger a subsequent call to loginBlocked in this system connection, as well as the equivalent in other system connections.

Usage
It should clean up any user-related resources.

loginPreviousUser(String, String, SessionData) method [deprecated]

Deprecated. Override loginPreviousUser(User, String, SessionData) instead. This method is called when a user has previously logged into Agentry successfully, and is now logging in again due to having been disconnected.

Syntax

`public LoginEnumeration loginPreviousUser ( String` *userId* `, String` *password* `, SessionData` *sessionData* `)`

Parameters

- **userId –** The user ID from the client application for the current user.
- **password –** The password for the current user, as entered on the client application.
- **sessionData –** Provides access to current session data, such as Agentry Server Data Markup Language (SDML) values.

Returns

One of the constants from Server.LoginEnumeration.

Usage

This method is only called if both the `enableAuthentication` and `enablePreviousUserAuthentication` options are `true` in the `Agentry.ini` file for the Java system connection. It should function in the same manner as login(String, String, SessionData).

The default implementation of this method returns Login_Pass.

loginPreviousUser(User, String, SessionData) method

This method is called when a user has previously logged into Agentry successfully, and is now logging in again due to having been disconnected.

Syntax

`public void loginPreviousUser ( User` *user* `, String` *password* `, SessionData` *sessionData* `)` throws `LoginException`

Parameters

- **user –** The User object that identifies the client user. The user name can be read from this object.
- **password –** The password for the current user, as entered on the client application.
- **sessionData –** Provides access to current session data, such as Agentry Server Data Markup Language (SDML) values.

Exceptions

- **LoginException class –** if the login fails for any reason, or if the login succeeds but an exceptional condition exists (such as an expired or soon-to-be expired password).

Usage

This method is only called if both the `enableAuthentication` and `enablePreviousUserAuthentication` options are `true` in the `Agentry.ini` file for the Java system connection. It should function in the same manner as login(User, String, SessionData).

The default implementation of this method throws LoginSkippedException.

setDebugEnabled(boolean) method [deprecated]

Deprecated. This is only here because the Agentry server will call it. Setter method called by the Agentry server to enable/disable debugging.

Syntax

`final void setDebugEnabled ( boolean` *debug* `)`

Parameters

- **debug –** true if debugging is enabled, false if not.

Usage

Subclasses must never call this.

shutdown() method

This method is called by the Agentry Server when the Java system connection is being shut down.

Syntax

```
public void shutdown ()
```

Usage

It should perform any cleanup that needs to be done.

startup() method

This method is called by the Agentry Server when the Java system connection starts up and creates an instance of this class; it is called immediately after the class is constructed.

Syntax

```
public void startup ()
```

Usage

It exists as a complement to the shutdown method. Current versions of Agentry ignore the value returned by this method, so there's really nothing you can do here that you couldn't just do in the constructor.

ServiceEvent class

This class implements a Java Callback Service Event in Agentry.

Syntax

```
public class  ServiceEvent extends
AgentryJavaBackEndManagedObject
```

Members

All members of ServiceEvent, including inherited members. **Variables**

Modifier and Type	Variable	Description
protected Server	*_server* on page 496	The active Server implementation in the Agentry Java system connection.
protected SessionData	*_sessionData* on page 496	Session data for this service event session.

Constructors

Modifier and Type	Constructor	Description
public	*ServiceEvent(Server, SessionData, CallbackInterface)* on page 495	Constructs a new ServiceEvent object.

Methods

Modifier and Type	Method	Description
public final void	*dataReceived(Object)* on page 495	This method should be called once you have obtained an object's data from the remote system.

Usage

In a Java Callback Service Event, a remote enterprise system initiates a call into this class, then retrieves data for a single Agentry object and passes that data back to the Agentry server. (The Agentry object can be a collection, if you need to handle multiple objects of the same type at once.) The general process works like this:

1. The enterprise system somehow triggers a call to a method in your custom subclass of `ServiceEvent`.
2. Your custom method acquires data from the enterprise system for a single Agentry object and stores it into a custom Java object. This object is implemented the same way as objects returned by ComplexTable or Steplet - it must contain public fields that can be mapped back to fields in an Agentry object.
3. Your custom method then calls the dataReceived method with the new object, which communicates that object back to Agentry.
4. Agentry processes the object, maps it to the Agentry object, and fires the various steps defined for the Service Event in the application for handling the object, which in turn will cause the various methods of `ServiceEventSession` to be invoked.

How the enterprise system actually triggers a call into this class is up to you. Possible methods might include remote RMI calls into the Agentry JVM, receiving JMS messages, or whatever else you can come up with.

ServiceEvent(Server, SessionData, CallbackInterface) constructor

Constructs a new ServiceEvent object.

Syntax

`public ServiceEvent ( Server` *server* `, SessionData` *sessionData* `, CallbackInterface` *cbi* `)`

Parameters

- **server –** The active `Server` object, which will be stored into _server.
- **sessionData –** The session data for this service event, which will be stored into _sessionData.
- **cbi –** The native callback object provided by Agentry; do not use this object directly, it will be handled by the dataReceived method of this class.

Usage

Subclasses should provide a constructor with the same arguments, and pass them untouched to this constructor. This constructor will store its arguments into the corresponding member variables of this class, which can then be accessed directly by subclasses.

dataReceived(Object) method

This method should be called once you have obtained an object's data from the remote system.

Syntax

`public final void dataReceived ( Object` *data* `)` throws `AgentryException`

Parameters

- **data –** The object to send back to Agentry.

Exceptions

- **AgentryException class –** if an error occurs.

Usage

It will pass that object back to the Agentry server, which will then copy the object's data to the corresponding object in the Agentry application using the mappings configured in the Agentry Editor for this service event.

_server variable

The active Server implementation in the Agentry Java system connection.

Syntax

```
protected Server _server
```

_sessionData variable

Session data for this service event session.

Syntax

```
protected SessionData _sessionData
```

ServiceEventSession class

The ServiceEventSession class encapsulates the processing involved in a service event.

Syntax

```
public class ServiceEventSession extends Session
```

Members

All members of ServiceEventSession, including inherited members. **Constructors**

Modifier and Type	Constructor	Description
public	*ServiceEventSession(String, Server, SessionData)* on page 499	Construct a new session.

Methods

Modifier and Type	Method	Description
public void	*beginDataAndUpdateSteps()* on page 499	This method is called by the server prior to the execution of the "Data State Steps" and "Update Steps" (which are grouped together) for the service event.
public void	*beginReadSteps()* on page 499	This method is called by the server prior to the execution of the "Read Steps" for the service event.
public void	*beginServiceEventError()* on page 500	This method is called by the server prior to the execution of the "Error Steps" for the service event.
public void	*endDataAndUpdateSteps()* on page 500	This method is called after the "Data State Steps" and "Update Steps" (which are grouped together) for the service event have been successfully completed.
public void	*endReadSteps()* on page 500	This method is called after the "Read Steps" for the service event have been successfully completed.
public void	*endServiceEventError()* on page 500	This method is called after the "Error Steps" for the service event have been successfully completed.

Inherited members from Session

Modifier and Type	Member	Description
public final void	*debug(String)* on page 503	Write the given message to a debug log, if debugging is enabled.
public String	*getName()* on page 503	Returns the name of the session, as configured in the Agentry application.

Modifier and Type	Member	Description
public Server	*getServer()* on page 503	Returns the Server singleton object that the Java system connection is currently using.
public SessionData	*getSessionData()* on page 504	Returns the session data for this session.
public User	*getUser()* on page 504	Returns the user for this session, if any.
protected	*Session(String, Server, SessionData, User)* on page 502	Construct a new session, and save each of the arguments so that they can be retrieved later via the getName(), getServer(), getSessionData(), and getUser() methods.
protected	*Session(String, Server, SessionData)* on page 502	Construct a new session, and save each of the arguments so that they can be retrieved later via the getName(), getServer(), and getSessionData() methods.
public void	*sessionAborted()* on page 504	This is called if the session is aborted (e.g., by an exception).

Usage

In brief, a service event is the component of the application that defines how data is synchronized between the Agentry Server and another external server application. It is made up of steps (which are implemented by the Steplet class in the Java system connection), each of which perform a specific task related to the synchronization process. These steps are organized into groups within the service event for specific areas of the data synchronization. These areas include the "Read", "Data State", "Update", and "Error Handling" steps.

This class contains methods to perform actions before and after each of these groups of steps. The default implementation of these methods in the AJAPI perform no additional specific actions. A designer can extend this class if special processing is required before or after each of these groups of steps are processed. If this class is extended, the Server class must also be extended and its `createServiceEventSession` method must be overridden to return the designer implemented subclass of the `ServiceEventSession` class.

ServiceEventSession(String, Server, SessionData) constructor

Construct a new session.

Syntax

`public ServiceEventSession ( String` *serviceEventName* `, Server` *server* `, SessionData` *sessionData* `)`

Parameters

- **serviceEventName –** The fetch name, as configured in the Agentry application.
- **server –** The `Server` object that the Java System connection was configured to use.
- **sessionData –** Session data for this fetch.

Usage

This constructor is called by the Server.createServiceEventSession method. Subclasses should implement a constructor with the same signature.

beginDataAndUpdateSteps() method

This method is called by the server prior to the execution of the "Data State Steps" and "Update Steps" (which are grouped together) for the service event.

Syntax

```
public void beginDataAndUpdateSteps ()
```

Usage

Any processing that should take place prior to the execution of these steps should be implemented in this method.

beginReadSteps() method

This method is called by the server prior to the execution of the "Read Steps" for the service event.

Syntax

```
public void beginReadSteps ()
```

Usage

Any processing that should take place prior to the execution of these steps should be implemented in this method.

beginServiceEventError() method

This method is called by the server prior to the execution of the "Error Steps" for the service event.

Syntax

```
public void beginServiceEventError()
```

Usage

Any processing that should take place prior to the execution of these steps should be implemented in this method.

endDataAndUpdateSteps() method

This method is called after the "Data State Steps" and "Update Steps" (which are grouped together) for the service event have been successfully completed.

Syntax

```
public void endDataAndUpdateSteps()
```

Usage

Any processing that should take place after the execution of these steps should be implemented in this method.

endReadSteps() method

This method is called after the "Read Steps" for the service event have been successfully completed.

Syntax

```
public void endReadSteps()
```

Usage

Any processing that should take place after the execution of these steps should be implemented in this method.

endServiceEventError() method

This method is called after the "Error Steps" for the service event have been successfully completed.

Syntax

```
public void endServiceEventError()
```

Usage

Any processing that should take place after the execution of these steps should be implemented in this method.

Session class

This is the base class for the various session types in Agentry.

Syntax

```
public class  Session extends AgentryJavaBackEndManagedObject
```

Derived classes

- *ComplexTableSession* on page 399
- *DataTableSession* on page 414
- *FetchSession* on page 425
- *PushSession* on page 449
- *PushUserSession* on page 455
- *ServiceEventSession* on page 496
- *TransactionSession* on page 519

Members

All members of Session, including inherited members. **Constructors**

Modifier and Type	Constructor	Description
protected	*Session(String, Server, SessionData, User)* on page 502	Construct a new session, and save each of the arguments so that they can be retrieved later via the getName(), getServer(), getSessionData(), and getUser() methods.
protected	*Session(String, Server, SessionData)* on page 502	Construct a new session, and save each of the arguments so that they can be retrieved later via the getName(), getServer(), and getSessionData() methods.

Methods

Modifier and Type	Method	Description
public final void	*debug(String)* on page 503	Write the given message to a debug log, if debugging is enabled.
public String	*getName()* on page 503	Returns the name of the session, as configured in the Agentry application.

Modifier and Type	Method	Description
public Server	*getServer()* on page 503	Returns the Server singleton object that the Java system connection is currently using.
public SessionData	*getSessionData()* on page 504	Returns the session data for this session.
public User	*getUser()* on page 504	Returns the user for this session, if any.
public void	*sessionAborted()* on page 504	This is called if the session is aborted (e.g., by an exception).

Usage

A session generally refers to the carrying out of a particular action, fetch, push, transaction, etc. in an Agentry application. It contains information about the server and user executing the session, and also holds a SessionData object that provides access back into the Agentry server to retrieve application-specific data.

Session(String, Server, SessionData, User) constructor

Construct a new session, and save each of the arguments so that they can be retrieved later via the getName(), getServer(), getSessionData(), and getUser() methods.

Syntax

`protected Session ( String` *name* `, Server` *server* `, SessionData` *sessionData* `, User` *user* `)`

Parameters

- **name –** The session name, as configured in the Agentry application.
- **server –** The Server object that the Java System connection was configured to use.
- **sessionData –** Session data for this fetch.
- **user –** The client user performing the fetch.

Session(String, Server, SessionData) constructor

Construct a new session, and save each of the arguments so that they can be retrieved later via the getName(), getServer(), and getSessionData() methods.

Syntax

`protected Session ( String` *name* `, Server` *server* `, SessionData` *sessionData* `)`

Parameters

- **name –** The session name, as configured in the Agentry application.
- **server –** The Server object that the Java System connection was configured to use.
- **sessionData –** Session data for this fetch.

debug(String) method

Write the given message to a debug log, if debugging is enabled.

Syntax

`public final void debug ( String` *message* `)`

Parameters

- **message –** The message to log

Usage

If this is a user-specific session then the message will be written to the user log, otherwise it will be written to the server log.

This method is simply a convenience method that calls `User.debug()` if the session has a user, or `Server.debug()` if it doesn't.

getName() method

Returns the name of the session, as configured in the Agentry application.

Syntax

```
public String getName ()
```

Returns

The session name.

getServer() method

Returns the Server singleton object that the Java system connection is currently using.

Syntax

```
public Server getServer ()
```

Returns

The server instance

Usage

This will be an instance of the class named in the `serverClass` option of the Java system connection in the `agentry.ini` file.

getSessionData() method

Returns the session data for this session.

Syntax

```
public SessionData getSessionData ()
```

Returns

The session data.

getUser() method

Returns the user for this session, if any.

Syntax

```
public User getUser ()
```

Returns

an object of whatever class is being returned by the active implementation of the `Server` class, or `null` if this session is not user-specific.

sessionAborted() method

This is called if the session is aborted (e.g., by an exception).

Syntax

```
public void sessionAborted ()
```

Steplet class

The Steplet class within the AJAPI encapsulates the data synchronization for a step application component.

Syntax

```
public abstract class Steplet extends
AgentryJavaBackEndManagedObject
```

Members

All members of Steplet, including inherited members. **Variables**

Modifier and Type	Variable	Description
protected Session	*_session* on page 511	Session data, set by the constructor.

Constructors

Modifier and Type	Constructor	Description
public	*Steplet(FetchSession)* on page 507	Constructs a new Steplet that will be used as part of a fetch.
public	*Steplet(PushSession)* on page 507	Constructs a new Steplet that will be used as part of a push.
public	*Steplet(PushUserSession)* on page 507	Constructs a new Steplet that will be used as part of a push.
public	*Steplet(TransactionSession)* on page 507	Constructs a new Steplet that will be used as part of a transaction.
public	*Steplet(ServiceEventSession)* on page 508	Constructs a new Steplet that will be used as part of a service event.

Methods

Modifier and Type	Method	Description
public abstract boolean	*doSteplet()* on page 508	Perform the necessary actions of this steplet.
public String	*getNotificationText()* on page 509	This method is called for Transaction Error-Handling Steplets.
public String	*getNotificationTitle()* on page 509	This method is called for Transaction Error-Handling Steplets.
public String	*getOkButtonLabel()* on page 509	This method is called for Transaction Error-Handling Steplets.
public Object	*getReturnData()* on page 510	The Agentry server will call this method to obtain the data produced by the doSteplet method.
public Session	*getSession()* on page 510	Returns the Session object for this steplet.

Modifier and Type	Method	Description
public String	*notificationText()* on page 511	Deprecated. Override getNotificationText() instead. The default implementation of this method will call that method. This method may be made final or removed in a future release. Use getNotificationText() instead.
public String	*notificationTitle()* on page 511	Deprecated. Override getNotificationTitle() instead. The default implementation of this method will call that method. This method may be made final or removed in a future release. Use getNotificationTitle() instead.
public String	*okButtonLabel()* on page 511	Deprecated. Override getOkButtonLabel() instead. The default implementation of this method will call that method. This method may be made final or removed in a future release. Use getOkButtonLabel() instead.

Usage

The Agentry Editor provides a template subclass of this class for each step in the application that uses a Java system connection; these subclasses must override the methods in this class to implement their behavior. A basic steplet needs to implement doSteplet() and getReturnData(). Steplets that will be used in Agentry transaction error-handling steps can also override the notificationTitle(), notificationText(), and okButtonLabel() methods to control the contents of a failed transaction's error notification windows.

A typical steplet implementation will implement doSteplet() to retrieve a set of data from a remote system, package that data into an object or array of objects, and save the data in a member field. The implementation of the getReturnData() method will then return that data as either a single object or an array of objects. These objects will will in turn contain publicly-visible fields that are mapped within the Agentry application (via the Agentry Editor) to the fields of corresponding Agentry objects. The Agentry server will read data directly from the fields of these objects; the server will not use getter/setter methods to read them.

A steplet can throw StepletStopException to stop processing of itself but allow processing of subsequence steplets in a session to continue, or it can throw StepletAbortException to stop processing of itself and any subsequent steplets in the session. It can also throw a BusinessLogicException exception to report an error message to the client's transmit window. Steplets that are used in transaction steps can also throw RetryTransactionException, RetryTransactionWithChangeException, or FatalTransactionException to abort the transaction in various ways.

Steplet(FetchSession) constructor

Constructs a new Steplet that will be used as part of a fetch.

Syntax

`public   Steplet ( FetchSession` *session* `)`

Parameters

- **session –** Fetch session information, stored into the `_session` member variable.

Steplet(PushSession) constructor

Constructs a new Steplet that will be used as part of a push.

Syntax

`public   Steplet ( PushSession` *session* `)`

Parameters

- **session –** Push session information, stored into the `_session` member variable

Steplet(PushUserSession) constructor

Constructs a new Steplet that will be used as part of a push.

Syntax

`public   Steplet ( PushUserSession` *session* `)`

Parameters

- **session –** Push user session information, stored into the `_session` member variable

Steplet(TransactionSession) constructor

Constructs a new Steplet that will be used as part of a transaction.

Syntax

`public   Steplet ( TransactionSession` *session* `)`

Parameters

- **session –** Transaction session information, stored into the `_session` member variable

Steplet(ServiceEventSession) constructor

Constructs a new Steplet that will be used as part of a service event.

Syntax

`public   Steplet ( ServiceEventSession` *session* `)`

Parameters

- **session –** Service event session information, stored into the `_session` member variable

doSteplet() method

Perform the necessary actions of this steplet.

Syntax

`public  abstract boolean  doSteplet ()` throws `AgentryException`

Returns

For fetch, push, transaction update step, and service event read, data, and update steplets: `true` if the steplet produced data that the Agentry server should read from the `_returnData` field, or `false` if no data was produced. For transaction data state and error handling steplets, and service event error handling steplets: the meaning of the return value is configured in the Agentry Editor.

Exceptions

- **AgentryException class –** if an error occurs.

Usage

A steplet can obtain various parameters from Agentry via the session information stored in the `_session` member variable. Steplet objects that retrieve data that will be read by the Agentry server should store that data in a public field named `_returnData`, which the Agentry Server will read via reflection if this method returns `true`.

A steplet can throw StepletStopException to stop processing of itself but allow processing of subsequence steplets in a session to continue, or it can throw StepletAbortException to stop processing of itself and any subsequent steplets in the session. It can also throw a BusinessLogicException exception to report an error message to the client's transmit window. Steplets used in transactions can also throw any of the transaction-related exceptions (FatalTransactionException, RetryTransactionWithChangeException, and RetryTransactionException).

getNotificationText() method
This method is called for Transaction Error-Handling Steplets.

Syntax
```
public String getNotificationText ()
```

Returns
the notification window text, or an empty string to use the text from the original exception that caused this error-handling steplet to be invoked.

Usage
It is intended to provide the error-handling steplet with a chance to override the notification window text from the original transaction failure exception. This method overrides the window text for the notification window; if it returns an empty string, then the text from the original exception will be used.

getNotificationTitle() method
This method is called for Transaction Error-Handling Steplets.

Syntax
```
public String getNotificationTitle ()
```

Returns
the notification window title, or an empty string to use the title from the original exception that caused this error-handling steplet to be invoked.

Usage
It is intended to provide the error-handling steplet with a chance to override the notification window text from the original transaction failure exception. This method overrides the window title for the notification window; if it returns an empty string, then the title from the original exception will be used.

getOkButtonLabel() method
This method is called for Transaction Error-Handling Steplets.

Syntax
```
public String getOkButtonLabel ()
```

Returns
the button label, or an empty string to use the label from the original exception that caused this error-handling steplet to be invoked.

Usage

It is intended to provide the error-handling steplet with a chance to override the notification window text from the original transaction failure exception. This method overrides the button label for the notification window; if it returns an empty string, then the button label from the original exception will be used.

getReturnData() method

The Agentry server will call this method to obtain the data produced by the doSteplet method.

Syntax

```
public Object getReturnData()
```

Returns

A data object or an array of data objects

Usage

It will only be called if doSteplet returned `true` and the steplet is being used by a data or fetch step in the Agentry application.

The objects returned by this method will be mapped to Agentry objects according to the field mappings defined in the Agentry application. This method should return either a single such object, or an array of them.

The implementation of this method typically should be very simple, in that it should just return some data that was built up in doSteplet. All of the "heavy lifting" should be done in that method.

Migration tip: If you are migrating an application from AJAPI version 4 (which returned steplet data via a special field named `_returnData`), just override this method to return the value of your steplet's `_returnData` field. If you do not override this method, the Agentry server will attempt to read that field anyways (for backwards compatibility), but this behavior is considered deprecated and should not be relied on in future versions of Agentry. It's also faster to have this method return the field, since otherwise the Agentry server will need to use reflection to read it.

getSession() method

Returns the Session object for this steplet.

Syntax

```
public Session getSession()
```

Returns

the session.

notificationText() method [deprecated]
Deprecated. Override getNotificationText() instead. The default implementation of this method will call that method. This method may be made final or removed in a future release. Use getNotificationText() instead.

Syntax
```
public String notificationText ()
```

Returns
the text

notificationTitle() method [deprecated]
Deprecated. Override getNotificationTitle() instead. The default implementation of this method will call that method. This method may be made final or removed in a future release. Use getNotificationTitle() instead.

Syntax
```
public String notificationTitle ()
```

Returns
the title

okButtonLabel() method [deprecated]
Deprecated. Override getOkButtonLabel() instead. The default implementation of this method will call that method. This method may be made final or removed in a future release. Use getOkButtonLabel() instead.

Syntax
```
public String okButtonLabel ()
```

Returns
the label

_session variable
Session data, set by the constructor.

Syntax
```
protected Session _session
```

StepletAbortException class

This exception can be thrown from a Steplet object to abort processing of that steplet and any subsequent steplets in the session.

Syntax

`public class  StepletAbortException extends AgentryException`

Members

All members of StepletAbortException, including inherited members. **Constructors**

Modifier and Type	Constructor	Description
public	*StepletAbortException(String)* on page 513	Constructs a new StepletAbortException object.

Inherited members from AgentryException

Modifier and Type	Member	Description
public	*AgentryException(String)* on page 395	Constructs a new AgentryException object.
public	*AgentryException(String, Throwable)* on page 395	Constructs a new AgentryException object.
public	*AgentryException(String, String, String, Throwable)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.
public	*AgentryException(String, String, String)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.
public final String	*getNotificationText()* on page 397	Returns the notification window text.
public final String	*getNotificationTitle()* on page 397	Returns the notification window title.
public final String	*getOkButtonLabel()* on page 397	Returns the notification window button label.

Usage

The Agentry Server will roll back any transactional work that has been done so far for the step, and will not execute any subsequent steps. It will trigger the `sessionAborted` method of the appropriate session object.

StepletAbortException(String) constructor

Constructs a new StepletAbortException object.

Syntax

```
public   StepletAbortException ( String message )
```

Parameters

- **message –** The error message to log to the server's event log.

StepletStopException class

This exception can be thrown from a Steplet object to stop the processing of the currently executing step and signal to the Agentry Server that any transaction-based work completed thus far should be committed.

Syntax

```
public class  StepletStopException extends AgentryException
```

Members

All members of StepletStopException, including inherited members. **Constructors**

Modifier and Type	Constructor	Description
public	*StepletStopException(String)* on page 514	Constructs a new StepletStopException object.

Inherited members from AgentryException

Modifier and Type	Member	Description
public	*AgentryException(String)* on page 395	Constructs a new AgentryException object.
public	*AgentryException(String, Throwable)* on page 395	Constructs a new AgentryException object.
public	*AgentryException(String, String, String, Throwable)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.
public	*AgentryException(String, String, String)* on page 396	Constructs a new AgentryException object, for methods that support throwing exceptions that may appear in the client.

Modifier and Type	Member	Description
public final String	*getNotificationText()* on page 397	Returns the notification window text.
public final String	*getNotificationTitle()* on page 397	Returns the notification window title.
public final String	*getOkButtonLabel()* on page 397	Returns the notification window button label.

Usage

The Agentry Server will then continue processing any remaining steps.

StepletStopException(String) constructor

Constructs a new StepletStopException object.

Syntax

`public StepletStopException ( String` *message* `)`

Parameters

- **message –** The error message to log

SycloCalendar class

This class extends GregorianCalendar with methods for detecting Agentry's "invalid date" value.

Syntax

```
public class  SycloCalendar extends GregorianCalendar
```

Members

All members of SycloCalendar, including inherited members. **Constructors**

Modifier and Type	Constructor	Description
public	*SycloCalendar(GregorianCalendar)* on page 515	Constructs a new SycloCalendar object using the data from an existing GregorianCalendar object and the default locale.
public	*SycloCalendar(GregorianCalendar, Locale)* on page 516	Constructs a new SycloCalendar object using the data from an existing GregorianCalendar object and the given locale.

Modifier and Type	Constructor	Description
public	*SycloCalendar()* on page 516	Constructs a new SycloCalendar object.
public	*SycloCalendar(int, int, int, int, int, int)* on page 516	Constructs a new SycloCalendar object.
public	*SycloCalendar(int, int, int, int, int)* on page 517	Constructs a new SycloCalendar object.
public	*SycloCalendar(int, int, int)* on page 517	Constructs a new SycloCalendar object.
public	*SycloCalendar(Locale)* on page 517	Constructs a new SycloCalendar object.
public	*SycloCalendar(TimeZone, Locale)* on page 517	Constructs a new SycloCalendar object.
public	*SycloCalendar(TimeZone)* on page 518	Constructs a new SycloCalendar object.

Methods

Modifier and Type	Method	Description
public static GregorianCalendar	*getInvalidTimeAndDate()* on page 518	Returns the Agentry invalid date value.
public boolean	*isInvalidTimeAndDate()* on page 518	Checks to see if this object contains Agentry's "invalid date" value.
public static boolean	*isInvalidTimeAndDate(GregorianCalendar)* on page 518	Returns whether the given date matches Agentry's "invalid date" value.

SycloCalendar(GregorianCalendar) constructor

Constructs a new SycloCalendar object using the data from an existing GregorianCalendar object and the default locale.

Syntax

`public   SycloCalendar ( GregorianCalendar` *cal* `)`

Parameters

- **cal –** The existing object

Usage
If you need a non-default locale, use SycloCalendar(GregorianCalendar, Locale); this constructor cannot read the locale from the given calendar object because GregorianCalendar does not provide a method for doing that.

SycloCalendar(GregorianCalendar, Locale) constructor
Constructs a new SycloCalendar object using the data from an existing GregorianCalendar object and the given locale.

Syntax
`public SycloCalendar ( GregorianCalendar` *cal* `, Locale` *locale* `)`

Parameters

- **cal –** The existing object
- **locale –** The locale

SycloCalendar() constructor
Constructs a new SycloCalendar object.

Syntax
`public SycloCalendar ()`

SycloCalendar(int, int, int, int, int, int) constructor
Constructs a new SycloCalendar object.

Syntax
`public SycloCalendar ( int` *year* `, int` *month* `, int` *dayOfMonth* `, int` *hourOfDay* `, int` *minute* `, int` *second* `)`

Parameters

- **year –** The year
- **month –** The month
- **dayOfMonth –** The day of the month
- **hourOfDay –** The hours
- **minute –** The minutes
- **second –** The seconds

SycloCalendar(int, int, int, int, int) constructor

Constructs a new SycloCalendar object.

Syntax

`public SycloCalendar ( int` *year* `, int` *month* `, int` *dayOfMonth* `, int` *hourOfDay* `, int` *minute* `)`

Parameters

- **year –** The year
- **month –** The month
- **dayOfMonth –** The day of the month
- **hourOfDay –** The hours
- **minute –** The minutes

SycloCalendar(int, int, int) constructor

Constructs a new SycloCalendar object.

Syntax

`public SycloCalendar ( int` *year* `, int` *month* `, int` *dayOfMonth* `)`

Parameters

- **year –** The year
- **month –** The month
- **dayOfMonth –** The day of the month

SycloCalendar(Locale) constructor

Constructs a new SycloCalendar object.

Syntax

`public SycloCalendar ( Locale` *locale* `)`

Parameters

- **locale –** The locale

SycloCalendar(TimeZone, Locale) constructor

Constructs a new SycloCalendar object.

Syntax

`public SycloCalendar ( TimeZone` *zone* `, Locale` *locale* `)`

Parameters

- **zone –** The time zone
- **locale –** The locale

SycloCalendar(TimeZone) constructor
Constructs a new SycloCalendar object.

Syntax
`public SycloCalendar ( TimeZone` *zone* `)`

Parameters

- **zone –** The time zone

getInvalidTimeAndDate() method
Returns the Agentry invalid date value.

Syntax
`public static GregorianCalendar getInvalidTimeAndDate ()`

Returns
the invalid date.

isInvalidTimeAndDate() method
Checks to see if this object contains Agentry's "invalid date" value.

Syntax
`public boolean isInvalidTimeAndDate ()`

Returns
`true` if the date is invalid, or `false` if not.

isInvalidTimeAndDate(GregorianCalendar) method
Returns whether the given date matches Agentry's "invalid date" value.

Syntax
`public static boolean isInvalidTimeAndDate ( GregorianCalendar` *testDate* `)`

Parameters

- **testDate –** the date to check

Returns
`true` if the date is invalid, or `false` if not.

TransactionSession class

The TransactionSession class encapsulates the processing related to transactions.

Syntax

```
public class  TransactionSession extends Session
```

Members

All members of TransactionSession, including inherited members. **Constructors**

Modifier and Type	Constructor	Description
public	*TransactionSession(String, Server, SessionData, User)* on page 520	Construct a new session.

Methods

Modifier and Type	Method	Description
public void	*beginTransaction()* on page 521	This method is called by the Agentry Server prior to executing the steps for the transaction.
public void	*endTransaction()* on page 521	This method is called after the steps for the transaction have been successfully processed.

Inherited members from Session

Modifier and Type	Member	Description
public final void	*debug(String)* on page 503	Write the given message to a debug log, if debugging is enabled.
public String	*getName()* on page 503	Returns the name of the session, as configured in the Agentry application.
public Server	*getServer()* on page 503	Returns the Server singleton object that the Java system connection is currently using.

Modifier and Type	Member	Description
public SessionData	*getSessionData()* on page 504	Returns the session data for this session.
public User	*getUser()* on page 504	Returns the user for this session, if any.
protected	*Session(String, Server, Session-Data, User)* on page 502	Construct a new session, and save each of the arguments so that they can be retrieved later via the getName(), getServer(), getSessionData(), and getUs-er() methods.
protected	*Session(String, Server, Session-Data)* on page 502	Construct a new session, and save each of the arguments so that they can be retrieved later via the getName(), getServer(), and getSessionData() methods.
public void	*sessionAborted()* on page 504	This is called if the session is aborted (e.g., by an exception).

Usage

In brief, transactions are the application component that defines data modifications made by the user on the client application. These changes are then transmitted to the Agentry Server for processing. The processing for a transaction is defined by its steps. This class contains methods that allow the implementation of processing that may be required before or after the transaction steps are processed.

A designer can extend this class if special processing is required before or after a transaction is processed. If this class is extended, the Server class must also be extended and its `createTransactionSession` method must be overridden to return the designer implemented subclass of the `TransactionSession` class.

TransactionSession(String, Server, SessionData, User) constructor

Construct a new session.

Syntax

`public TransactionSession ( String` *transactionName* `, Server` *server* `, SessionData` *sessionData* `, User` *user* `)`

Parameters

- **transactionName –** The fetch name, as configured in the Agentry application.

- **server –** The Server object that the Java System connection was configured to use.
- **sessionData –** Session data for this fetch.
- **user –** The client user performing the fetch.

Usage

This constructor is called by the Server.createTransactionSession method. Subclasses should implement a constructor with the same signature.

beginTransaction() method

This method is called by the Agentry Server prior to executing the steps for the transaction.

Syntax

```
public void beginTransaction()
```

Usage

Any processing that should take place at this point should be implemented in this method.

endTransaction() method

This method is called after the steps for the transaction have been successfully processed.

Syntax

```
public void endTransaction()
```

Usage

Any processing that should take place at this point should be implemented in this method.

User class

This class represents an Agentry client user.

Syntax

```
public class User extends AgentryJavaBackEndManagedObject
```

Members

All members of User, including inherited members. **Variables**

Modifier and Type	Variable	Description
protected String	*_name* on page 531	User name.

Constructors

Modifier and Type	Constructor	Description
public	*User(String)* on page 525	This is the constructor method for objects of type User.

Methods

Modifier and Type	Method	Description
public final GregorianCalendar	*backendTimeAndDate()* on page 525	Deprecated. This method has been renamed to getSystem-ConnectionTime(). This meth-od has been renamed to getSys-temConnectionTime().
public void	*beginChangePassword()* on page 526	This is the first method called when a user is attempting to change their password.
public ChangePasswordResult	*changePassword(String, String)* on page 526	This method is called when a user is attempting to change their password.
public void	*changePasswordFailed(String-Buffer)* on page 527	This method is called when the changePassword(String, String) method returns any value other than ChangePassword_Success or ChangePassword_NotHan-dled.
public void	*changePasswordSessionAbor-ted()* on page 527	This method is called if the password change operation is aborted for any reason.
public final void	*debug(String)* on page 527	Writes a debugging message to the user's log file on the Agentry server.
public void	*endChangePassword()* on page 528	This method is called when the user's password has been suc-cessfully changed.
public String	*getName()* on page 528	Returns the user's name.
public GregorianCalendar	*getSystemConnectionTime()* on page 528	This is called by the Agentry server to find out what time the Java system connection thinks it is right now.
public final void	*getTimeZone(StringBuffer)* on page 529	Deprecated. This method has been moved to Server#getTime-Zone(). This method is no lon-ger supported.

Modifier and Type	Method	Description
public void	*loggedIn()* on page 529	This method is called after a user has been successfully logged in.
public void	*loggedOut()* on page 529	This method is called after the transmission has been completed and after the user is logged out of the system.
public void	*reLoggedIn()* on page 530	This method is called when a user logs into the Agentry Server and the server still has a previous login session for that user.
public void	*revalidate(String)* on page 530	This method authenticates a client user against the Java System Connection.
public void	*timedOut()* on page 531	This method is called in the event a user session times out.
public void	*update(GregorianCalendar)* on page 531	This method is called periodically (once every second or so) by the Agentry Server.

Usage

This class is created by the Server#createUser(String) factory method. It contains methods for notifying the application of successful login, logout, and other events. It also contains methods which can be overridden to allow Agentry to change a user's password on a remote system.

Applications can extend this class to implement their own behavior; however, if you do so you must also override the Server#createUser(String) factory method to return the new subclass. You must then change the `serverClass` setting in the `Agentry.ini` configuration file to tell the Agentry Java system connection to use your new Server subclass.

User.ChangePasswordResult enum

Outcomes for password changes, returned by the User#changePassword(String, String) method and its ilk.

Members

All members of ChangePasswordResult, including inherited members. **Variables**

Modifier and Type	Variable	Description
public	*ChangePassword_Blocked* on page 524	The user has been blocked; the password has not been changed.
public	*ChangePassword_Failure* on page 524	The attempt to change the user's password has failed.
public	*ChangePassword_NotHandled* on page 525	The user's password change is not handled by this System Connection.
public	*ChangePassword_Success* on page 525	The user's password has been successfully changed.

Methods

Modifier and Type	Method	Description
public int	*getValue()* on page 524	Called by Agentry to retrieve the integer value for the enum.

getValue() method

Called by Agentry to retrieve the integer value for the enum.

Syntax

```
public int getValue()
```

Returns

the value

ChangePassword_Blocked variable

The user has been blocked; the password has not been changed.

Syntax

```
public  ChangePassword_Blocked
```

ChangePassword_Failure variable

The attempt to change the user's password has failed.

Syntax

```
public  ChangePassword_Failure
```

ChangePassword_NotHandled variable

The user's password change is not handled by this System Connection.

Syntax

`public   ChangePassword_NotHandled`

Usage

(Normally used in environments where multiple back-end systems are in place).

ChangePassword_Success variable

The user's password has been successfully changed.

Syntax

`public   ChangePassword_Success`

User(String) constructor

This is the constructor method for objects of type User.

Syntax

`public   User ( String` *name* `)`

Parameters

- **name –** The user ID as entered on the client application. This value is accessible by calling the name() member method.

Usage

The API class Server will call this method prior to attempting to log the user into the system and/or perform the data synchronization with the Java interface.

backendTimeAndDate() method [deprecated]

Deprecated. This method has been renamed to getSystemConnectionTime(). This method has been renamed to getSystemConnectionTime().

Syntax

`public  final  GregorianCalendar  backendTimeAndDate ()` throws `AgentryException`

Returns

nothing, throws `UnsupportedOperationException`.

Exceptions

- **AgentryException class –** not thrown

- **UnsupportedOperationException –** to indicate that the method is no longer supported and should not be called.

Usage

Override that method instead.

beginChangePassword() method

This is the first method called when a user is attempting to change their password.

Syntax

```
public void beginChangePassword()
```

Usage

It is intended to allow the designer to implement any functionality or processing that may be necessary prior to changing the user's password. Typically this method will start a password change transaction with a remote server. The actual password change functionality should not be a part of this processing.

This method will only be called if authentication is enabled for the Agentry Java system connection.

changePassword(String, String) method

This method is called when a user is attempting to change their password.

Syntax

```
public ChangePasswordResult changePassword(String oldPassword,
String newPassword)
```

Parameters

- **oldPassword –** The current password for the user.
- **newPassword –** The value that user's password should be changed to.

Returns

one of the constants from User.ChangePasswordResult

Usage

This method can be overridden to implement password changing against a remote system. The return value indicates the success or failure, and the reason for the failure, of the change password attempt.

This method will only be called if authentication is enabled for the Agentry Java system connection.

changePasswordFailed(StringBuffer) method

This method is called when the changePassword(String, String) method returns any value other than ChangePassword_Success or ChangePassword_NotHandled.

Syntax

`public void changePasswordFailed ( StringBuffer` *errorString* `)`

Parameters

- **errorString –** This is a string value that can be set to a text value. This value will be written out to the user's debug log. It should indicate the reason why the password change failed.

Usage

This method can be overridden to perform any necessary processing in the event a password change fails.

This method will only be called if authentication is enabled for the Agentry Java system connection.

changePasswordSessionAborted() method

This method is called if the password change operation is aborted for any reason.

Syntax

`public void changePasswordSessionAborted ()`

Usage

It should roll back anything that was started by the beginChangePassword() method.

This method will only be called if authentication is enabled for the Agentry Java system connection.

debug(String) method

Writes a debugging message to the user's log file on the Agentry server.

Syntax

`public final void debug ( String` *userMessage* `)`

Parameters

- **userMessage –** The message to log

Usage

This will only work if per-user logging is turned on in `AgentryLogging.ini`.

This method is a convenience method that calls into the Java Logging API to do the actual logging, and assumes that Agentry's default Java Logging configuration is in place (which will route log messages back to the Agentry server). It will log to a logger named "com.syclo.agentry.Server", at the `FINE` level (which translates to log detail level 3 in Agentry).

When invoked outside of Agentry (e.g. in unit tests), this will log to the console, as that is Java's normal default logging configuration.

endChangePassword() method

This method is called when the user's password has been successfully changed.

Syntax

`public void endChangePassword()`

Usage

Designers can override this method to perform any additional processing that may be required after the user's password has been modified. This method will still be called even if changePasswordFailed() is called before it.

This method will only be called if authentication is enabled for the Agentry Java system connection.

getName() method

Returns the user's name.

Syntax

`public String getName()`

Returns

the user name

getSystemConnectionTime() method

This is called by the Agentry server to find out what time the Java system connection thinks it is right now.

Syntax

`public GregorianCalendar getSystemConnectionTime()` throws `AgentryException`

Returns

The current date and time from the remote system's perspective.

Exceptions

- **AgentryException class –** if the current date and time cannot be determined.

Usage

Implementations that are communicating with remote servers should override this method to return the time on the remote server, if possible, in order to help Agentry calculate the difference between the time on the client and the time on the remote system. If you override the Server#getTimeZone method or set the Java system connection's time zone explicitly in `Agentry.ini`, then you should override this method to return the current time in the same time zone that that method is reporting.

getTimeZone(StringBuffer) method [deprecated]

Deprecated. This method has been moved to Server#getTimeZone(). This method is no longer supported.

Syntax

```
public  final  void  getTimeZone ( StringBuffer tz )
```

Parameters

- **tz –** not used

Exceptions

- **UnsupportedOperationException –** to indicate that the method is no longer supported and should not be called.

Usage

Override Server#getTimeZone() instead, as the time zone affects the entire Java system connection and not a specific user.

loggedIn() method

This method is called after a user has been successfully logged in.

Syntax

```
public  void  loggedIn ()
```

Usage

The default version of this method writes a message to the user's debug log, if debugging is active, and returns. This method can be overridden if additional processing is required after a user has successfully logged in and before the transmission is processed.

loggedOut() method

This method is called after the transmission has been completed and after the user is logged out of the system.

Syntax

```
public  void  loggedOut ()
```

Usage
This is the last method called prior to the destruction of a User object. The default version of this method performs only the single task of logging a message to the user's debug log, if debugging is enabled. This method can be overridden to disconnect a user from a remote system or perform other cleanup.

reLoggedIn() method
This method is called when a user logs into the Agentry Server and the server still has a previous login session for that user.

Syntax
`public void reLoggedIn ()`

Usage
This can occur if a user loses network connectivity in the middle of a transmission. This method can be overridden to perform any special processing that may be needed in this situation.

revalidate(String) method
This method authenticates a client user against the Java System Connection.

Syntax
`public void revalidate ( String` *password* `)` throws `LoginException`

Parameters

- **password –** The password for the current user, as entered on the client application.

Exceptions

- **LoginException class –** if the login fails for any reason, or if the login succeeds but an exceptional condition exists (such as an expired or soon-to-be expired password).

Usage
This method is called when a user reconnects to the Agentry Server from a client application and the `enableAuthentication` option is set to `true` in the Java section of the `Agentry.ini` file. Override this method to implement logic to perform full validation of the user against a remote system.

This method should return normally if the authentication of the user succeeds. If authentication fails for any reason, the appropriate LoginException subclass should be thrown. An exception should also be thrown for other conditions such as expired or soon-to-be-expiring passwords. By default, this method throws PasswordInvalidException, which means that the password is not valid for the user.

timedOut() method

This method is called in the event a user session times out.

Syntax

```
public void timedOut ()
```

Usage

A time out occurs when a transmission is idle for longer than the configured maximum time limit. This method can be overridden to perform any tasks that may be needed in this event.

In the event of a timeout, the user will also be logged out of the Agentry Server, which will trigger calls to the loggedOut() method as well.

update(GregorianCalendar) method

This method is called periodically (once every second or so) by the Agentry Server.

Syntax

`public void update ( GregorianCalendar` *update* `)`

Parameters

- **update –** The time when the timer tick occurred, should be "now" more or less.

Usage

This method can be overridden if some sort of routine maintenance is required, such as sending a keep-alive to a remote server.

_name variable

User name.

Syntax

```
protected String _name
```

Usage

Can be obtained via the getName() method.

SessionData interface

The SessionData interface is used throughout the AJAPI classes.

Syntax

```
public interface SessionData
```

Members

All members of SessionData, including inherited members. **Constructors**

Modifier and Type	Constructor	Description
public SessionData	*sessionData(String)* on page 533	Returns a new SessionData object, configured with the given SDML name prefix, that has access to the same session data as this object.

Methods

Modifier and Type	Method	Description
public String	*eval(String)* on page 534	Evaluates the given string as an Agentry SDML expression and returns the result.
public boolean	*getBoolean(String)* on page 534	Returns the specified property as a Boolean value.
public byte[]	*getBytes(String)* on page 534	Returns the specified property as an array of bytes.
public double	*getDouble(String)* on page 535	Returns the specified property as a double-precision floating point decimal value.
public float	*getFloat(String)* on page 535	Returns the specified property as a floating point decimal value.
public int	*getInteger(String)* on page 535	Returns the specified property as an integer value.
public long	*getLong(String)* on page 535	Returns the specified property as a long integer value.
public String	*getString(String)* on page 536	Returns the specified property as a string.
public GregorianCalendar	*getTimeAndDate(String)* on page 536	Returns the specified property as a date contained in a GregorianCalendar object.
public String	*getTimeAndDate(String, String)* on page 536	Returns the specified property as a date string using the given format, specified as an Agentry date format string (not a Java date format string!).

Usage

It can also be used within the designer-implemented extensions of those classes. This class encapsulates the Server Data Markup Language (SDML) functionality available in Agentry. Through a `SessionData` object, the designer can access the data specific to the current session.

This class contains several getter methods to return specified pieces of data. Each of these methods returns the data as a different data type, such as a string or integer. All of these methods take a single argument of type `String` that specifies the data to return. A an example, to retrieve the data for a string property within a transaction named "Description", the following line of code would be used:

```
String desc = _sessionData.getString("Description");
```

When calling these methods, it is important to make sure that the appropriate method is called for the desired data type. No errors will be reported for mismatched data types. For example, if a property within a transaction is of type integer, and the value is retrieved by calling getString, the value will be returned as a String value. In some cases this may be desirable behavior, but in others it can cause undesirable results.

The primary implementation of this interface is a private class that can only be instantiated by the Agentry Server, since it calls back into the running Agentry server to obtain its data. For unit-testing purposes, you can also create a new subclass or mock implementation of this interface; one example of such a testing version is TestSessionData.

sessionData(String) constructor

Returns a new SessionData object, configured with the given SDML name prefix, that has access to the same session data as this object.

Syntax

`public SessionData sessionData ( String` *sessionData* `)`

Parameters

- **sessionData –** The new SDML name prefix.

Returns

A new `SessionData` object that prefixes all property references with the prefix given by `sessionData`.

Usage

This can be used to create a session data object that is effectively restricted to only accessing data that starts with the given prefix.

eval(String) method

Evaluates the given string as an Agentry SDML expression and returns the result.

Syntax

`public String eval ( String` *sdmlString* `)`

Parameters

- **sdmlString –** The SDML string to evaluate (without the enclosing angle brackets).

Returns

The result as a string.

getBoolean(String) method

Returns the specified property as a Boolean value.

Syntax

`public boolean getBoolean ( String` *property* `)`

Parameters

- **property –** The property name

Returns

The property value as a Boolean.

getBytes(String) method

Returns the specified property as an array of bytes.

Syntax

`public byte[] getBytes ( String` *property* `)` throws `AgentryException`

Parameters

- **property –** The property name

Returns

The property value as an array of bytes

Exceptions

- **AgentryException class –** if the property cannot be interpreted as bytes

getDouble(String) method
Returns the specified property as a double-precision floating point decimal value.

Syntax
`public double getDouble ( String` *property* `)`

Parameters

- **property –** The property name

Returns
The property value as a double.

getFloat(String) method
Returns the specified property as a floating point decimal value.

Syntax
`public float getFloat ( String` *property* `)`

Parameters

- **property –** The property name

Returns
The property value as a float.

getInteger(String) method
Returns the specified property as an integer value.

Syntax
`public int getInteger ( String` *property* `)`

Parameters

- **property –** The property name

Returns
The property value as an integer.

getLong(String) method
Returns the specified property as a long integer value.

Syntax
`public long getLong ( String` *property* `)`

Parameters

- **property –** The property name

Returns
The property value as a long integer.

getString(String) method
Returns the specified property as a string.

Syntax
`public String getString ( String` *property* `)`

Parameters

- **property –** The property name

Returns
The property value as a string.

getTimeAndDate(String) method
Returns the specified property as a date contained in a GregorianCalendar object.

Syntax
`public GregorianCalendar getTimeAndDate ( String` *property* `)` throws `AgentryException`

Parameters

- **property –** The property name

Returns
The property value as a date. If the property was empty, then this will return the current date.

Exceptions

- **AgentryException class –** if the value cannot be parsed as a date.

getTimeAndDate(String, String) method
Returns the specified property as a date string using the given format, specified as an Agentry date format string (not a Java date format string!).

Syntax
`public String getTimeAndDate ( String` *property* `, String` *format* `)`

Parameters

- **property –** The property name
- **format –** The date format to use. Note that this is an Agentry Server date format, not the format used by e.g., `SimpleDateFormat`!

Returns
The property value as a formatted string.

Usage
If you prefer to use Java date format strings, then just call getTimeAndDate(String) to get a `Calendar` object and feed it to a `SimpleDateFormat` object.

Index