SMS Application Development

# SAP Mobile Platform 3.0

# Contents

# SMS Application Development

You can create SMS applications using predefined application states and custom states that you develop.

## Developing SMS Applications

Develop an SMS application to manage consumer services by sending and receiving SMS messages.

1. *Starting the Server*

   Start the SMS Builder server.
2. *Launching the SMS Builder Web UI*

   Use a Web browser to launch the SMS Builder Web UI and log in to the server.
3. *Developing Applications*

   Application states are basic building blocks that you can link sequentially to model application task flows. Applications are executed by the processing engine at runtime.
4. *Developing Custom Application States*

   Custom state development using the State SDK is a Java development task you can perform with or without a development IDE, such as Eclipse or NetBeans. After you develop and deploy custom states, you can use them to develop applications.
5. *Activating Applications*

   You must activate applications before you can test or run them. If you modify an active application and save changes, you must reactivate the application before changes are applied to the active version.
6. *Testing Applications*

   Test applications using the built-in application simulator.

## Starting the Server

Start the SMS Builder server.

**Prerequisites**

Install SAP® Mobile Platform SDK, including SMS Toolkit.

### Task

The start-up scripts (`run.sh` and `run.bat`) expect the location of the `java` application to be in the PATH. If you plan to run a start-up script in production, modify the script to reflect your system architecture. You need not have root privilege to execute a start-up script.

Change to the *`SMSBUILDER_HOME`* directory, and run the command or commands for your platform:

| Platform | Command |
|---|---|
| Windows | `bin\run.bat` |
| AIX or Linux | `chmod 755 bin/run.sh`<br>`bin/run.sh start` |

### See also

- *Launching the SMS Builder Web UI* on page 2

## Launching the SMS Builder Web UI

Use a Web browser to launch the SMS Builder Web UI and log in to the server.

### Prerequisites

1. If SMS Builder is running on the same machine as SAP Mobile Platform, configure the HTTP port.
2. Start the SMS Builder server.

### Task

1. Open a browser and navigate to either:
   - `http://localhost:<HTTP_port>/brand`, where *HTTP_port* is the HTTP port number, or
   - `http://server:<HTTP_port>/brand`, where *server* is the name of the machine on which the SMS Builder server is running.
2. Log in.

   The default login credentials are:
   - User Name – `admin`
   - Password – `Brand!23`

   If you created a new admin password, use it to log in.

**See also**

- *Starting the Server* on page 1

## Configuring the HTTP Port

The default HTTP port number is 8080, which is the same port that SAP Mobile Platform uses for its OData proxy. If SMS Builder is running on the same machine as SAP Mobile Platform, set the HTTP port number to a value that is not being used by SAP Mobile Platform.

1. Verify which port numbers SAP Mobile Platform is using, by checking the SMP_HOME `\Server\config_master\org.eclipse.gemini.web.tomcat\default-server.xml` file.
2. In the SMS Builder installation, edit the SMSBUILDER_HOME`\conf\org.ops4j.pax.web.properties` file, and set the HTTP port to a value that SAP Mobile Platform is not using.

# Developing Applications

Application states are basic building blocks that you can link sequentially to model application task flows. Applications are executed by the processing engine at runtime.

Two application types, interactive and event, differ by both how they are invoked and how they perform. Interactive applications provide rich, user-interactive mobile services, and are typically invoked when mobile customers send a keyword to a preassigned short or long code. Event applications work non-interactively, such as batch processes that send campaign messages, and are typically invoked by events, such as scheduled times or triggers.

You can create applications:

- From scratch
- Using provided application templates
- By importing application files from another computer

You can visually compose a mobile-messaging application, test it using a built-in simulator, and deploy it, ready to be used by mobile consumers.

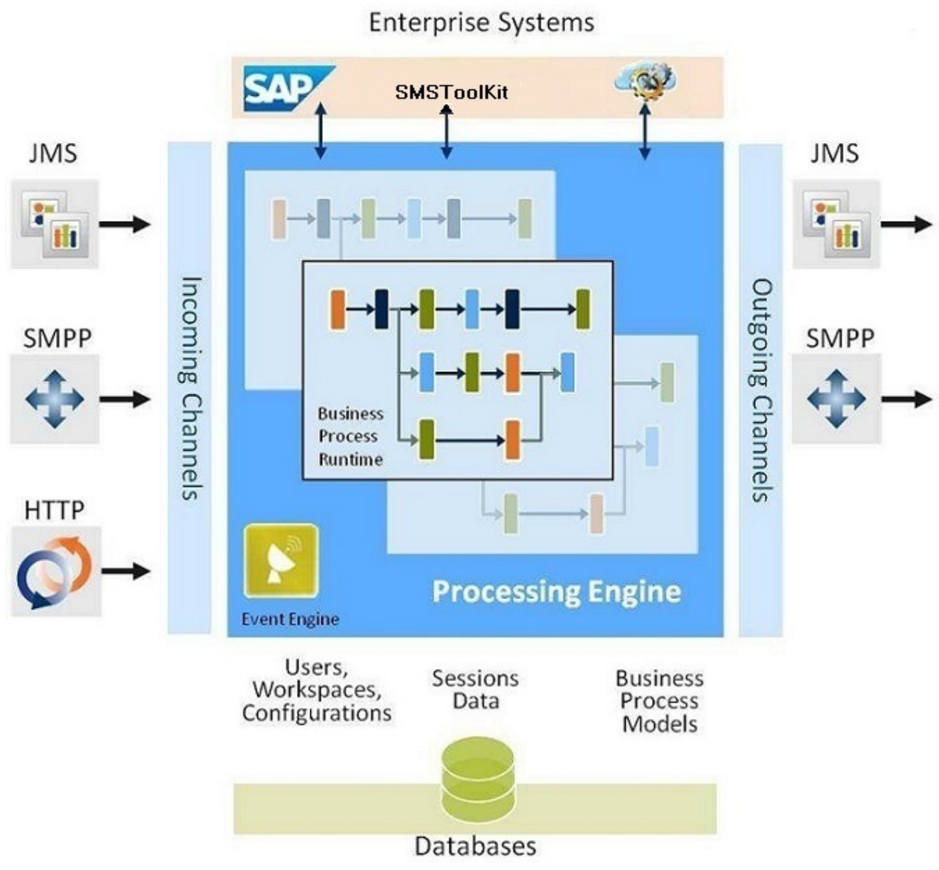
## Messaging Server

The core of SMS Builder is the messaging server.

The server components include:

- Processing engine – manages application life cycles, and provides the runtime environment.
- Event engine – invokes applications based on scheduled events.
- Session manager – tracks active sessions and terminates expired sessions.

- Channel manager – manages incoming and outgoing communication channels.



## Application States

States are basic building blocks that you can link sequentially to model application-process flows.

Application states are either:

- Standalone – implemented natively.
- Service – proxy to a Web service or aggregated Web services that are exposed through the service-oriented architecture (SOA) layer.

You can meet customer requirements by developing custom states using the State SDK. You can add custom states dynamically using the plug-in mechanism that is enabled by the OSGi services registry.

Create applications using the Application Composer Web tool. Application types include:

- Interactive – provide a user-interactive mobile service; typically invoked when mobile consumers send a keyword to a preassigned short code.
- Event – designed for batch processing; invoked by events, such as scheduled times, system triggers, or external triggers.

Most states can be used in either application type. However, there are a few states that are available only to a specific application type. For example, you can use the Process Subscriber state only in event applications, because it relies on the callback mechanism provided by the processing engine. You can use Application Call and Application Call Return states only in interactive applications, because these states do not support the callback mechanism. The Application Composer prevents you from adding invalid states to an application.

## Base States

Base states provide standalone functionality, without dependency on or interaction with external services. You commonly use base states to construct process flows.

Base states perform functions such as calling applications, comparing and copying variables, incrementing counters, sending SMS messages, and setting session variable values.

### See also

- *Application Call State* on page 89
- *Application Call Return State* on page 92
- *Compare Typed Variables State* on page 94
- *Compare Variables State* on page 96
- *Copy Variables State* on page 98
- *Counter State* on page 100
- *Goto Application State* on page 104
- *Send SMS State* on page 109
- *Set Variable State* on page 120
- *Start Application State* on page 122

## Subscriber States

Applications that contain subscriber states have access to subscriber storage, which stores attributes that are useful in push campaigns.

Subscriber storage is nondurable storage for staging, or in-transit storage, pending batch transfer to the system of record. The database schema is designed to be generic, and is not fully optimized for large scale or more domain-specific purposes.

### See also

- *Add Subscriber State* on page 86
- *Get Subscriber State* on page 101
- *Process Subscriber State* on page 106
- *Update Subscriber State* on page 123

## USSD States

SMS Builder delivers Unstructured Supplementary Service Data (USSD) states via Java Messaging Service (JMS) to external USSD channels.

USSD states prompt subscribers for input, and send text notifications and menu-based requests.

**Note:** By default, USSD states are disabled. USSD is a custom protocol that mobile operators can implement. To develop USSD applications, contact SAP® Professional Services.

### See also

- *Send USSD Input State* on page 111
- *Send USSD Menu State* on page 112
- *Send USSD Text State* on page 118

## Custom States

You can develop Java custom application states to extend the functionality of SMS Builder, and to meet client-specific requirements.

Custom states are typically developed by:

- SAP® personnel to implement client-specific requirements.
- Third parties for plug-in applications to meet client requirements.

To integrate new custom states, develop Java components using the provided APIs, and customize the product by installing custom-state bundles.

## Input and Output Parameters

Application states can have input and output parameters. Input parameters allow states to receive input from consumers, other states, and applications. Output parameters allow states to save values in session variables that can be used by other states or applications.

Input parameters contain the information a state requires to perform its task. Input parameters can be constant values, or values copied from a variable in the current user session.

Output parameters allow states to return values. All output parameters are available as variables.

### See also

- *Custom State Variables* on page 46
- *Defining Input Variables* on page 48
- *Defining Output Variables* on page 50
- *Accessing Input Variables* on page 51
- *List Variables* on page 52

### State Machine

A state machine defines an application process flow at runtime. During development, you can compose an application task flow visually using the Application Composer. When you activate the application, the process flow is converted to a state machine.

States are elements of a state-machine system. An application usually has many states, and can include different types of states. Each state has a previous state and a following state, unless it is the initial state or the final state. There can be only one initial state, but, depending on user interaction, there can be many final states.

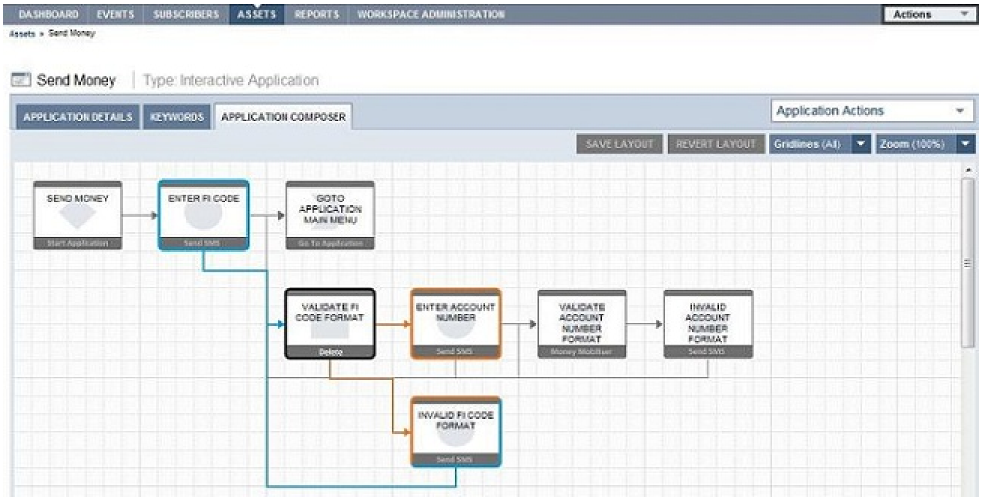
An initial state is the first state in an application, and only handles state transitions to follow-up states, based on transition rules. The initial-state is Start Application, which is created automatically when you create an application, and cannot be deleted. By default, the name of the initial state is the same as the name of the application.

## Application Composer

To visually develop applications, use the Application Composer.

The Application Composer state layout view lets you visualize the processing steps of the application task flow. You can create states and draw transitions between them. The Application Composer enables application developers to:

- Visualize states in the application using an automatic layout
- Drag and drop states to rearrange the layout
- Highlight the context, dependencies, and transitions of states
- Zoom in and out to see a complete or partial application layout
- Set the grid line type



The Layout Canvas shows the application flow, from left to right, on a grid line background. The flow consists of states (shown in boxes) and transitions that connect two states (depicted as lines with arrows). State boxes include the name of the state instance, the type, and a watermark pattern that define the state type. In complex applications, transition lines may overlap.

When you highlight a state, all of its transition lines and states they connect to are highlighted. To highlight a state, move the cursor over the state icon and left-click. The dependent states and transition lines display in different colors:

- The selected state displays a dark gray border; for example, the Validate FI Code Format state in the screen above. When you select a state, the text at the bottom of the state icon changes to **Delete**.
- States that transition to the highlighted state display a blue border and a blue transition line.
- States to which the highlighted state transitions display an orange border and an orange transition line.
- States that transition both to and from the highlighted state have borders that are half blue and half orange (dual mode); for example, the Invalid FI Code Format state in the screen above.

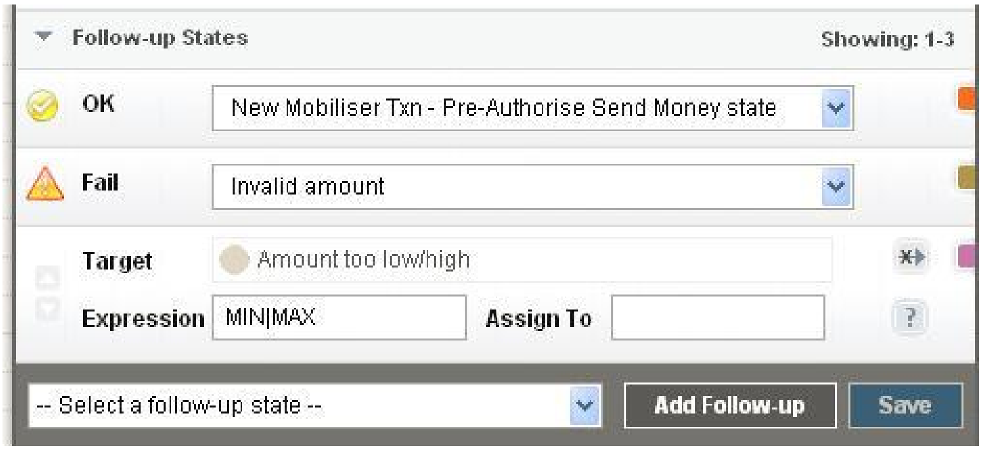
## State Transitions

Some state transitions are determined by matching regular expressions with text supplied by consumers. Other states have specific transitions that define follow-up states, which state developers define in the code.

The OK and Fail transitions do not use pattern matching; such transitions are based on states' code, and validation provided by, or events in, back-end systems. Some states do not require OK or Fail transitions. If a state does require one of these transitions, and you do not specify a follow-up state, the application terminates.

For dynamic transitions, a state's code has the option to return an expression, which provides the input to the pattern-matching mechanism. Dynamic transitions also provide a way to transition to success or failure outcomes, and may replace the OK and Fail transitions. Dynamic transitions can communicate information back to applications about certain validation problems.

This example includes an OK transition, a Fail transition, and a dynamic transition that uses the expression MIN|MAX.



OK: New Mobiliser Txn - Pre-Authorise Send Money state

Fail: Invalid amount

Target: Amount too low/high

Expression: MIN|MAX

Assign To:

**See also**

- *Controlling State Transitions with Regular Expressions* on page 9

## Controlling State Transitions with Regular Expressions

You can control state transitions by defining regular expressions. When expressions match user-input strings, the state transitions to the follow-up state.

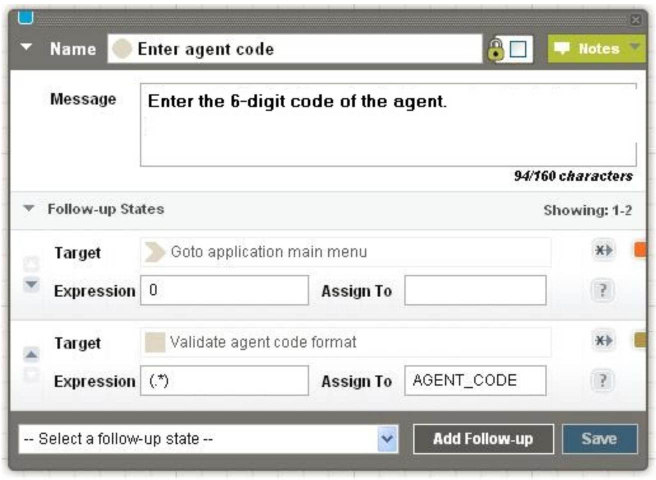
Some states expect user input to control the transition to follow-up states. Input can be provided either by consumers in response to the Send SMS state, or as dynamic output from either a SMS Builder state, or a third-party custom state. Dynamic values allow external systems to communicate specific context information back to the application.

A regular expression can contain any combination of characters. You can test regular expressions during application development using a built-in tool. Sample regular expressions are:

| Regular Expression | Matches |
|---|---|
| `.*` | Any value in the Expression field. |
| `(.*)` | Any value in the Expression field; assigns the expression to a session variable. |

In more complex cases, you can break a regular expression into multiple regular-expression groups and assign them to separate session variables.

For a complete description of regular expressions, see: *http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html*.



Name: Enter agent code

Message: Enter the 6-digit code of the agent.

94/160 characters

| | | | |
|---|---|---|---|
| Target | Goto application main menu | | |
| Expression | 0 | Assign To | |
| Target | Validate agent code format | | |
| Expression | (.*) | Assign To | AGENT_CODE |

In the state editor, Target identifies the state that follows the current state if its Expression value matches the input. If the input matches more than one Expression value, a list of matches is created. The first entry in the list is the first matching pattern, continuing with other states in the order in which they appear in the state editor. For example, if the input is `0`, the follow-up state is Goto Application Main Menu, even though `0` also matches the second expression. If the input is anything other than `0`, it matches the second expression, and the value is assigned to the session variable AGENT_CODE, because the value of Expression is surrounded by parentheses. To move an expression up or down in the Follow-up States list, use the arrows on the left side of the editor.

## Testing Regular Expressions

As you develop applications in the Application Composer, you can test regular expressions to determine whether they match alphanumeric strings.

1. In the Application Composer, select a state.
2. Click the **?** icon to the right of the Assign To field for a follow-up state.

   The expression tester opens and populates Expression and Assign To fields with follow-up state values from the state editor.
3. Enter the value to test in the **Text to Test** field, and click **Test**.

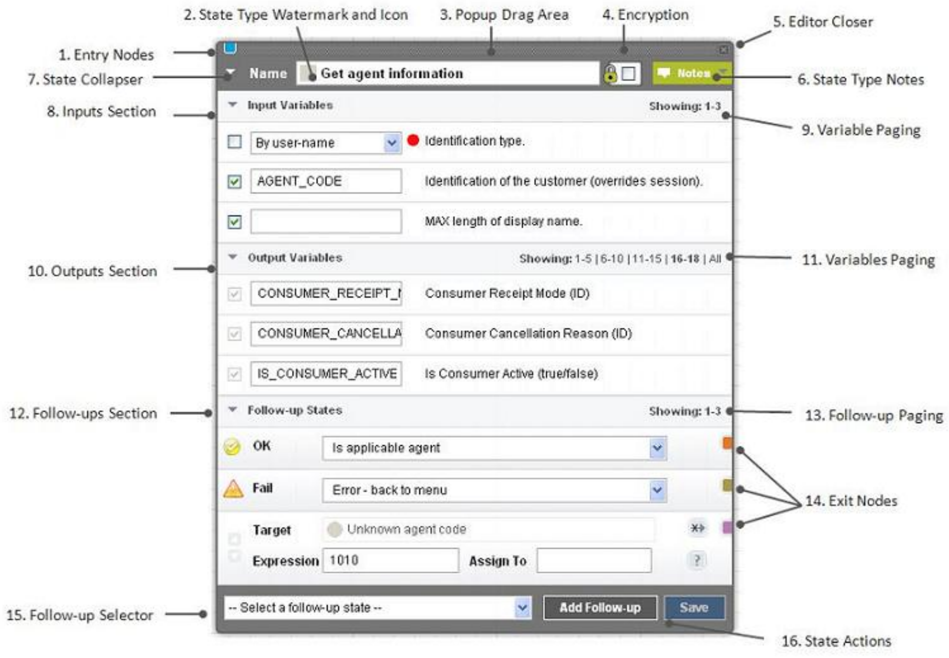
The result is either:

- `Match` – value in Expression field matches the value in Text to Test field.
- `No Match` – expression value does not match Text to Test value.

## State Editor

In the state editor, you can edit state properties, define follow-up states, test regular expressions for follow-up transitions, and detach the current state from follow-up states.

The state editor window opens automatically when you select a state in the Application Composer. Depending on the state type, the state editor displays various options, context-sensitive links, and entry fields.



Name: Get agent information

Input Variables — Showing: 1-3

| | | |
|---|---|---|
| ☐ | By user-name | ● Identification type. |
| ☑ | AGENT_CODE | Identification of the customer (overrides session). |
| ☑ | | MAX length of display name. |

Output Variables — Showing: 1-5 \| 6-10 \| 11-15 \| 16-18 \| All

| | | |
|---|---|---|
| ☑ | CONSUMER_RECEIPT_I | Consumer Receipt Mode (ID) |
| ☑ | CONSUMER_CANCELLA | Consumer Cancellation Reason (ID) |
| ☑ | IS_CONSUMER_ACTIVE | Is Consumer Active (true/false) |

Follow-up States — Showing: 1-3

| | |
|---|---|
| OK | Is applicable agent |
| Fail | Error - back to menu |
| Target | Unknown agent code |
| Expression | 1010 |
| Assign To | |

-- Select a follow-up state --

State editor fields and controls are:

1. Entry Nodes – identify links to other states that transitions to this state. If you click an entry node, a state editor opens for the corresponding state. If you hover over an entry node, you see the state name with which it is associated.
2. State Type Watermark and Icon – icon associated with the state type. The watermark allows you to quickly recognize state types in the editor and in the layout view.
3. Pop-up Drag Area – you can move the state editor anywhere within the Application Composer by clicking the header and dragging.
4. Encryption – encrypts incoming and outgoing messages, which are saved in message logs.
5. Editor Closer – closes the state editor. If you have pending changes that have not been saved, you are prompted to either save or discard these changes.
6. State Type Notes – to view or edit notes that describe a state's function, input and output variables, and follow-up state transitions, click the down arrow.

7. State Collapser – shrink or enlarge the state editor.
8. Inputs Section – input variable names and values. Click the down arrow to close this section.
9. Variables Paging for Inputs Section – if a state contains more than five input variables, you can page through the others by selecting the relevant page set. To display all input variables, click **All**.
10. Outputs Section – output variable names and values. Click the down arrow to close this section.
11. Variables Paging for Outputs Section – if a state contains more than five output variables, you can page through the others by selecting the relevant page set. To display all output variables, click **All**.
12. Follow-up Section – configure, change, and test follow-up states. To collapse this section, click the down arrow.
13. Follow-up Paging – three follow-up states appear on each page. To see more follow-up states, select the relevant page set.
14. Exit Nodes – identify links to other states that this state transitions to. If you select an exit node, a state editor opens for the next state. If you hover over an exit node, you see the state name with which it is associated.
15. Follow-up Selector – select the follow-up state. All states, except the Goto Application state, allow you to add a follow-up state.
16. State Actions – **Add Follow-up State** and **Save**.

## Adding States to Applications

You can add new states in the Application Composer. When you create a new application, a Start Application state is created automatically, as the initial application state.

1. In the Application Composer, select an existing state.
2. In the state editor, expand the list of follow-up states, and select a state.
3. Click **Add Follow-up**.
   The new state appears in the Application Composer. A transition line connects the current state to the new state.

A new state is automatically assigned the name New **State Type** State. Change the name, because state names must be unique.

## Editing State Properties

You can edit state properties and state transitions in the Application Composer.

1. In the Application Composer, select the state you want to edit.
2. In the state editor, configure state properties.

   These changes are immediately saved to the database:
   - Adding a new follow-up state

- Adding a transition to an existing state
- Removing a transition from an existing state
- Moving a transition up or down in the list of follow-up states

3. For other changes, click **Save**.

### Removing States

In the Application Composer, you can remove states from an application. Removing a state permanently deletes the state and transition lines that are connected to it from the application.

1. In the Application Composer, select the state to remove.
2. Click **Delete**.

If you remove a state that has follow-up states, these states may be orphaned.

### Removing State Transitions

Removing a state transition permanently deletes the transition, but does not remove any follow-up states to which it is connected.

1. In the Application Composer, select the state with the follow-up transition you want to delete.
2. In the state editor, to the right of the Target State field, click the asterisk-arrowhead icon:

#### Next

To reattach orphaned states, add a new transition using the follow-up selector.

## Developing Interactive Applications

Interactive applications provide rich, user-interactive mobile services, and are typically invoked when mobile customers send a keyword to a preassigned short or long code.

1. In the Dashboard screen, at the bottom of the My Applications module, select **Create Interactive Application**.
2. On the Application Details tab, enter:
   - Name – the main identifier for an application. SAP recommends that you do not use duplicate names within a workspace.
   - Category – (optional) select the application category from the list. You can use categories to group applications together for managing and reporting.
   - Active From – the date and time the application becomes active, based on the server date and time.
   - Active To – the date and time the application ceases to be active, based on the server date and time.
   - Timeout (secs) – an interactive application establishes conversations with mobile subscribers. When a conversation starts, a unique session is established for the

conversation. The session terminates (or times out) when there is no conversation for more than the number of seconds you enter here. The default value is 450 seconds (7 minutes and 30 seconds).
- Session Limit Response – the message that is sent to mobile subscribers when the application cannot start or carry on a conversation for various reasons; the most common reason being too many conversations are already taking place, exceeding the system capacity. In this case, the default message is sent to mobile subscribers. For example, the message may say "Service busy, try again in few minutes."

3. Click **Save**.
4. (Optional) To save the application to the local file system, click **Export**.
   The application is exported to an application XML file. You can transfer the XML file to other SMS Builder workspaces or instances. You can also use the file to back up the application, or store it in the source control management system.

   **Note:** The Export button is disabled until you save an application the first time.
5. Add a keyword to the application.
6. Design the application task flow.
7. Activate the application.
8. Test the application.

### See also

- *Activating Applications* on page 22
- *Testing Interactive Applications* on page 24

## Adding Keywords to Applications

A keyword identifies an application within a workspace. Create at least one keyword for each interactive application.

1. Select the **Keywords** tab, and enter values for these fields.
   - Add New Keyword – enter plain text or regular expressions. SAP recommends that a keyword be unique for each application in the same workspace.
   - Active From – the date and time the keyword becomes active, based on the server date and time.
   - Active To – the date and time the keyword ceases to be active, based on the server date and time.
2. To save the keyword, click the diskette icon.

   After you save a keyword, another Add New Keyword field appears, allowing you to add another keyword.

### See also

- *Searching for a Keyword* on page 15
- *Short Codes, Long Codes, and Keywords* on page 16

## Searching for a Keyword

Keywords should be unique within a workspace. The keyword-search tool enables application developers to see if a keyword is assigned to any applications.

If you use a regular expression to define a keyword, the keyword search tool cannot detect duplicates.

1. In the Interactive Applications window, select the **Keywords** tab.
2. Enter the keyword for which to search, and click **Search**.

If any applications in the workspace already use the keyword, this information appears on the screen:

- Used by – the application name.
- Approved – indicates whether the application is active. False means that either the application is inactive, or the application has never been activated, so the status is draft.

### See also

- *Adding Keywords to Applications* on page 14
- *Short Codes, Long Codes, and Keywords* on page 16

## Designing Application Task Flows

The key to effective application development is defining the task flows involved in modeling business processes. In the Application Composer, you can graphically design an application task flow.

The first time you open the Application Composer, you see the Start Application state. If you select the state, the state editor opens, which allows you to add follow-up states.

You can rearrange a layout by dragging and dropping state icons. To get a better view of state transitions, you may want to rearrange the layout, particularly when transition lines overlap. You can drag and drop state icons into fixed-grid positions on the canvas. The canvas does not allow free-form positions. Transition lines are automatically positioned, and you cannot move them.

- To move a state, select it, and drag it to an alternate grid position.
  While moving, the state icon appears transparent, and the target grid positions are highlighted when the mouse enters the grid area.
- To delete a state, select the state, and click **Delete**.

  When you delete a state, all transitions to and from other states are deleted. However, corresponding states and all of their downstream flows are not deleted. States that are not connected to other states become orphans, but they are still accessible from the follow-up state list, and you can connect them to other states.
- To save a rearranged layout to the database, click **Save Layout**.

- To revert the application layout to the last one saved in the database, click **Revert Layout**.
- To change the grid lines, expand the **Gridlines** list, and select All, Partial, or None.
- To zoom in or out, expand the **Zoom** list, and select the magnification you want to see, relative to the initial display.

  If you zoom out from the default 100% view, you must reset the zoom level back to 100% before you can make any layout changes.

### See also

- *Developing Event Applications* on page 17
- *Activating Applications* on page 22
- *Creating Events* on page 19
- *Assigning Events to Applications* on page 20
- *Activating Events* on page 23
- *Testing Event Applications* on page 26

## Short Codes, Long Codes, and Keywords

A short code or long code plus a keyword identifies an interactive application within a workspace.

Each workspace has a unique short or long code. For incoming messages, the processing engine compares the destination MSISDN with the short or long code list to find a matching workspace. Once a matching workspace is identified, the processing engine compares the message content with keywords assigned to applications in the workspace. A workspace can contain many applications, which should all have unique keywords. At runtime, the processing engine stops when it finds the first matching keyword, and calls the corresponding application.

A short code is a special telephone number, significantly shorter than a full telephone number that can be used to address SMS and MMS messages from some mobile phones or fixed phones, and is limited to national borders. A long code is a longer number and is available internationally.

SMS Builder uses short codes and long codes differently from how they are used in the mobile-operator world. Short codes are often associated with mobile services, such as interactive applications, and they are assigned by the mobile operator to the owner of the service.

For example, company XYZ wants to provide a mobile service for paying street-parking fines in the financial district of San Francisco. XYZ applies for an assigned short code from a mobile operator. Typically, the short code (9999)  is advertised on billboards in the financial district area: “To pay parking fines with your mobile phone, text “SFpay to 9999.” When a mobile subscriber texts `SFpay` to 9999, the message first reaches the mobile operator. The operator, in turn, routes it to SMS Builder. When SMS Builder receives the message, the processing engine maps the destination MSISDN (9999) to a workspace. Once the workspace is identified, the engine looks at the keyword `SFpay` and maps it to the corresponding interactive application in that workspace. The first matching application is chosen.

A keyword can be a simple string like "coupon," or a regular expression. Optionally, you can associate a date range with a keyword, which controls the length of time a keyword remains active. A keyword's date range takes precedence over an application's date range: if an application's date range expires, but the keyword date range is still active, the application remains active until the keyword dates expire. When keyword dates are empty, the application defines the date range.

Best practices:

- Verify that an interactive application acting as an entry point has at least one assigned keyword.
- Use the keyword-search tool to verify that a keyword is assigned to only one application in the workspace.
- If you define a regular expression as a keyword, verify that the regular expression does not overlap with keywords that are already in use by other applications. The keyword-search tool does not work for regular expressions.

### See also

- *Adding Keywords to Applications* on page 14
- *Searching for a Keyword* on page 15

## Developing Event Applications

Event applications work non-interactively, such as batch processes that send campaign messages, and are typically invoked by events, such as scheduled times, system triggers, or external triggers. An event application can send outbound messages but has no user-interactive capability.

After you create and activate an event application, you can assign an event to it. When the event is active, it invokes the event application. You can assign an event to only one event application.

1. On the Dashboard screen, at the bottom of the My Applications module, select **Create Event Application**.
2. On the Application Details tab, enter:
   - Name – the main identifier for an application. SAP recommends that you do not use duplicate names within a workspace.
   - Category – (optional) select the application category from the list. You can use categories to group applications together for managing and reporting.
   - Active From – the date and time the application becomes active, based on the server date and time.

- Active To – the date and time the application ceases to be active, based on the server date and time.

3. To save the application, expand Advanced Settings, and click **Save**.
4. Select the **Application Composer** tab, and define the application states and the task flow.
5. Activate the application.
6. Create an event and assign it to the application.
7. Activate the event.
8. Test the application.
9. (Optional) To export the application, expand Advanced Settings, and click **Export**.

   **Note:** The Export button is disabled until you save an application the first time.

   The application is exported to an XML file, and saved to the local file system. You can transfer the XML file to other SMS Builder workspaces or instances. You can also use the file to back up the application, or store the XML in the source control management system.

### See also

- *Designing Application Task Flows* on page 15
- *Activating Applications* on page 22
- *Creating Events* on page 19
- *Assigning Events to Applications* on page 20
- *Activating Events* on page 23
- *Testing Event Applications* on page 26

## Events

A SMS Builder event invokes an event application. Event applications are designed for batch processing, and are triggered by events, such as scheduled times.

You assign an event to an event application, so that when the event occurs, the application is invoked. For example, you can create a promotional event that is scheduled between November 1 and November 30. Within this event runtime, you can define event windows that specify when to invoke the event application. You can define event windows by setting start and stop dates and times. You can also define recurring windows, for example, to occur daily, by setting start and stop times.

The event model is a container for storing configuration details and relationships, including active runtime, event windows (manual or recurring), the event application to trigger when an event window is current, and all related interactive applications.

If you assign an event to an interactive application, no one can delete the application.

## Creating Events

Create an event to invoke an event application.

1. In the Dashboard screen, at the bottom of the My Events module, select **Create New Event**.
2. On the Event Details tab, enter:
   - Name – the main identifier for an event. Duplicate names within a workspace are allowed, but not recommended.
   - Category – (optional) select a category from the list. You can use categories to filter events.
   - Runtime From – the date and time the event becomes active, based on the server date and time.
   - Runtime To – the date and time the event ceases to be active, based on the server date and time.
   - Description – (optional) a description of the event's purpose.
3. Click **Save**.

### Next

1. Set up event windows.
2. Assign the event to an active event application.
3. Activate the event.

### See also

- *Developing Event Applications* on page 17
- *Designing Application Task Flows* on page 15
- *Activating Applications* on page 22
- *Assigning Events to Applications* on page 20
- *Activating Events* on page 23
- *Testing Event Applications* on page 26

## Creating One-Time Event Windows

Event windows define event start and stop times, and events invoke event applications. At the event-window start time, the event starts its corresponding event application; the event application stops either when it has finished processing its data, or at the event-window stop time, whichever comes first.

1. On the Events screen, select the **Event Windows** tab.
2. Click **Add New Window**, and enter:
   - Start date and time – time and date at which to start the event application.

- Stop date and time – time and date at which to stop the event application.
- Limit – maximum number of loopbacks to process. When used with a throttle, specify as a multiple of throttle. For example, if throttle = 60 messages per minute, specify a limit of 60, 120, or 180.
- Throttle – maximum processing rate: number of messages per minute.
- Resume – select to resume from the last processed item; leave unselected to restart from the beginning of the list. This is useful for states that process lists, such as the Process Subscriber state.

3. Save your settings.
4. (Optional) Create another event window, if necessary.

## Creating Recurring Event Windows

Create recurring event windows to start event applications at the same time every day, week, or month.

1. On the Events screen, select the **Event Windows** tab.
2. Click **Add New Window**.
3. Select **Switch to Recurring Mode**, and select:
   - Recurring Start Date – the date at which to start the event application.
   - Recurring Interval – the frequency at which to start the application: Daily, Weekly, or Monthly.
4. Click **Add New Window**, and enter:
   - Start time – time at which to start the event application.
   - Stop time – time at which to stop the event application.
   - Limit – maximum number of loopbacks to process. When used with a throttle, specify as a multiple of throttle. For example, if throttle = 60 messages per minute, specify a limit of 60, 120, or 180.
   - Throttle – maximum processing rate: number of messages per minute.
   - Resume – select to resume from the last processed item; leave unselected to restart from the beginning of the list.
5. Save your settings.
6. (Optional) Define additional recurring event windows, if required.

## Assigning Events to Applications

Assign an event to an event application. The event invokes the event application.

### Prerequisites

Activate the event application.

### Task

1. In the main Web UI window, select **Events**.
2. Select the event, then select either the **Event Applications** tab or the **Interactive Applications** tab.

   **Note:** You can assign an event to only one event application. If an assignment already exists, you can remove it. If you assign an event to an interactive application, it prevents it from being inadvertently deleted. You can assign an event to an unlimited number of interactive applications.
3. Click **Assign Applications**.
4. To narrow the list of applications that appear, do one of the following, and click **Search**:
   - Select **Event Applications** or **Interactive Applications**.
   - Enter the application name.
   - Expand the **Advanced** list, and select a category.
5. Select the application to assign to the event.
6. To save the assignment, select:
   - **Add to Event** – remains on the current screen.
   - **Add and Return to Event** – returns to the Events screen, and displays the Event Applications tab.

### See also

- *Developing Event Applications* on page 17
- *Designing Application Task Flows* on page 15
- *Activating Applications* on page 22
- *Creating Events* on page 19
- *Activating Events* on page 23
- *Testing Event Applications* on page 26

## Activation

Before you can test and run applications and events, you must activate them.

The processing engine executes applications and events when they are in active mode. If you edit the active version of an application or an event in the Web UI, changes are saved to an in-review version. Changing an in-review version does not impact the active version, until you reactivate the application or event.

Initially, the mode of activated applications and events is on-deck, and changes to active when the active from-date and time are the same as the current-date and time. Artifacts in active mode are rolled back to on-deck mode, if the active from-date and time are moved into the future.

To run some newly created artifacts—default menus, applications, and events—you must activate them. If you make any changes to one of these artifacts, you must reactivate them.

Once artifacts are activated, changes are committed and cannot be rolled back. If applications or events contain mistakes, deactivate them.

## Application Mode Transitions

After you create an application, it transitions through a series of modes. A running application is in active mode.

| Initial Mode | Event/Condition | New Mode |
|---|---|---|
| None | Create an application | Draft |
| Draft | • Activate the application<br>• Start date is earlier than current date | Active |
| Draft | • Activate the application<br>• Start date is later than current date | On-deck |
| On-deck | Start date is earlier than current date | Active |
| On-deck | Modify the application | On-deck |
| On-deck | • Modify the application<br>• Start date is earlier than current date | Active in-review |
| Active | Modify the application | Active in-review |
| Active | End date is earlier than current date | Ended |
| Active In-Review | End date is earlier than current date | Ended |

## Activating Applications

You must activate applications before you can test or run them. If you modify an active application and save changes, you must reactivate the application before changes are applied to the active version.

Applications that are currently running are in active mode. If you activate an application, but its active start time is in the future, the application mode is on-deck, and cannot be tested.

1. On the Web UI navigation bar, select **Assets**.
2. On the Assets screen, select **Activate Applications**.
3. Click **Load Applications for Activation**.

   Applications that are in-review appear.

4. Choose either:
    - To activate a single application, select **Actions** > **Activate**.
    - To activate all in-review applications, select **Activate All**.

### See also

- *Developing Event Applications* on page 17
- *Designing Application Task Flows* on page 15
- *Creating Events* on page 19
- *Assigning Events to Applications* on page 20
- *Activating Events* on page 23
- *Testing Event Applications* on page 26
- *Developing Interactive Applications* on page 13
- *Testing Interactive Applications* on page 24

## Activating Events

Activate an event to invoke an event application.

### Prerequisites

Assign the event to an active event application.

### Task

1. In the Web UI navigation bar, select **Events**.
2. For the event you want to activate, select **Actions** > **Activate**.

### See also

- *Developing Event Applications* on page 17
- *Designing Application Task Flows* on page 15
- *Activating Applications* on page 22
- *Creating Events* on page 19
- *Assigning Events to Applications* on page 20
- *Testing Event Applications* on page 26

## Deactivating Applications

If necessary, you can deactivate or delete an application.

- To deactivate the application until a specified future date, change the active from-date to a future date, and reactivate.
- (Interactive applications only) To prevent an application from being invoked, remove the keywords, and reactivate.

- To delete an application:
    a) Export the application.
    b) Delete the application.

### Deactivating Events

If necessary, you can deactivate an event. If the event has a current event window, change the window start date to a future date, before deactivating the event.

1. In the Web UI navigation bar, select **Events**.
2. Select the event you want to deactivate.
3. On the **Event Details** tab, change the active from-date to a future date.
4. (If necessary) Reset the event window start date and time.
5. Save your changes and reactivate the event.

    The event remains inactive until the specified future date.

## Testing Applications

Test applications using the built-in application simulator.

To access the Simulation page, expand the Actions list on the right side of the navigation bar, and select **Simulate Application**. You can test interactive applications and event applications. Select the tab that corresponds to the application type you want to test.

You can also test applications using either a Short Message Peer-to-Peer (SMPP) test harness or a Java Message Service (JMS) test harness; these methods are typically used by custom-state developers and advanced system administrators.

### Testing Interactive Applications

Test an interactive application in the current workspace by simulating incoming and outgoing messages.

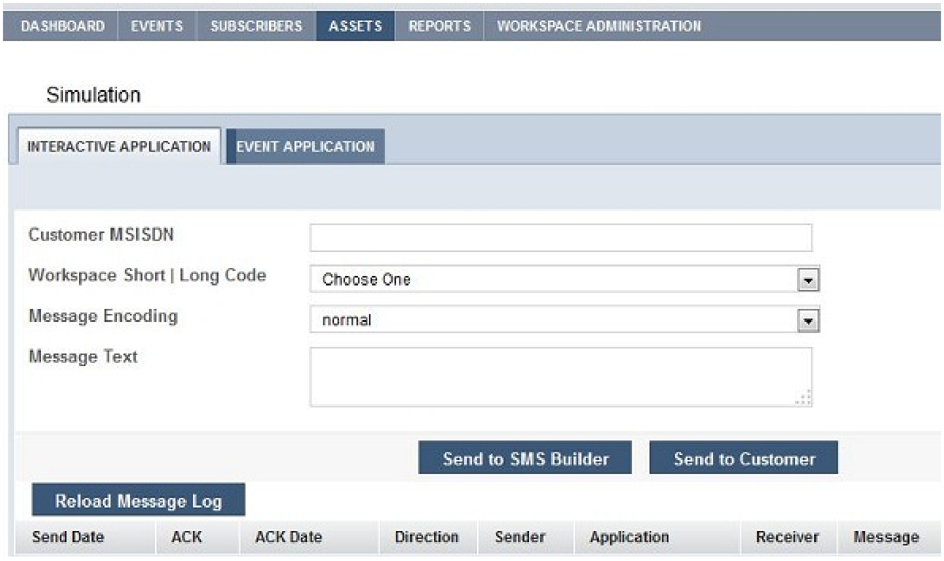
#### Prerequisites

Activate the application.

#### Task

1. On the **Interactive Application** tab of the Simulation page, enter:
    - Customer MSISDN – numeric value. SMS Builder uses the MSISDN to either create a new session or find the existing session. If the application being tested has states that interface with a back-end system, enter an MSISDN that identifies a customer in that system.
    - Workspace Short | Long Code – select from the list.
    - Message Encoding – accept the default, or select Unicode.

- Message Text – a valid keyword for the application.



|  |  |
|---|---|
| Customer MSISDN |  |
| Workspace Short \| Long Code | Choose One |
| Message Encoding | normal |
| Message Text |  |

| Send Date | ACK | ACK Date | Direction | Sender | Application | Receiver | Message |
|---|---|---|---|---|---|---|---|

2. Click **Send to SMS Builder**.
3. To see responses, click **Reload Message Log**.

   If the application calls an external Web service, responses may take longer than the page-refresh time.

## See also

- *Developing Interactive Applications* on page 13
- *Activating Applications* on page 22

### *Sample Interactive Message Log*

An interactive-application message log shows a sequence of consumer interactions with SMS Builder.

For each message, the logs displays:

- Send Date – the date and time the message was sent.
- ACK and ACK Date – whether an acknowledgment is requested from the short message service center (SMSC) or the SMS gateway, and the date and time the acknowledgment was received.
- Direction – message direction, IN or OUT; IN messages come from customers; OUT messages are SMS Builder responses.
- Sender – sender's identification number. For IN messages, the number is the customer's MSISDN; for OUT messages, it is the workspace short or long code.

- Application – name of the application that processed the message. A SMS Builder application can call other applications, which are identified in the log.
- Receiver – receiver's identification number. For IN messages, the number is the workspace short or long code; for OUT messages, it is the customer's MSISDN.

## Testing Event Applications

To test event applications, invoke the triggering event. Event applications are linked to events that occur at times defined by their event windows.

1. On the Simulation page, select the **Event Application** tab:
   - Event Name – select from the list.
   - Resume From Last – accept the default value, false. If set to true, and if the previous test did not exhaust the subscriber list, the application resumes from the last subscriber.
   - Throttle – enter the maximum processing rate: number of messages per minute.
   - Limit – enter the maximum number of loopbacks to process. When used with a throttle, specify as a multiple of throttle. For example, if throttle = 60 messages per minute, specify a limit of 60, 120, or 180.
   - Event Threads – specify the number of threads to use to run the simulation. Change this value to test performance with different numbers of threads.
   - End Date – specify to keep the application from overrunning.
2. Click **Simulate Event**.
3. To see messages, click **Reload Message Log**.

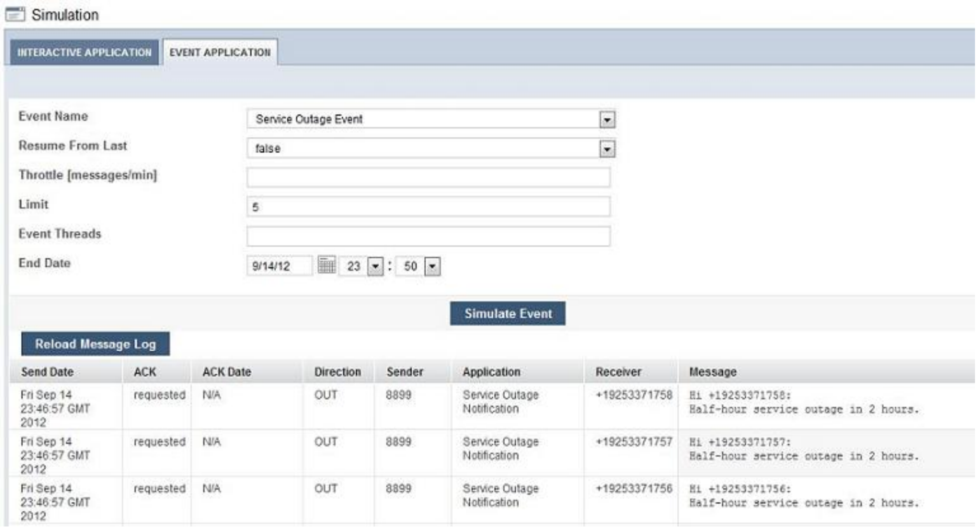
   Depending on the number of subscribers, you may need to reload the log multiple times to see all the messages.

### See also

- *Developing Event Applications* on page 17
- *Designing Application Task Flows* on page 15
- *Activating Applications* on page 22
- *Creating Events* on page 19
- *Assigning Events to Applications* on page 20
- *Activating Events* on page 23

### *Sample Event Message Log*

The Utility Notification event application generates messages that appear in the message log.



| | |
|---|---|
| Event Name | Service Outage Event |
| Resume From Last | false |
| Throttle [messages/min] | |
| Limit | 5 |
| Event Threads | |
| End Date | 9/14/12 23 : 50 |

| Send Date | ACK | ACK Date | Direction | Sender | Application | Receiver | Message |
|---|---|---|---|---|---|---|---|
| Fri Sep 14 23:46:57 GMT 2012 | requested | N/A | OUT | 8899 | Service Outage Notification | +19253371758 | Hi +19253371758: Half-hour service outage in 2 hours. |
| Fri Sep 14 23:46:57 GMT 2012 | requested | N/A | OUT | 8899 | Service Outage Notification | +19253371757 | Hi +19253371757: Half-hour service outage in 2 hours. |
| Fri Sep 14 23:46:57 GMT 2012 | requested | N/A | OUT | 8899 | Service Outage Notification | +19253371756 | Hi +19253371756: Half-hour service outage in 2 hours. |

**See also**

- *Utility Notification Event Application* on page 32

## Importing Applications

You can import application XML files that were previously exported from SMS Builder, and you can create applications from Quick-Start template files that are installed with SMS Builder.

**See also**

- *Exporting Applications* on page 28

### Importing Application XML Files

Import an application by uploading the XML file that contains the application configuration. XML configuration files are created when you export applications from SMS Builder.

If you import a single application that links to other applications, create the linked-to applications before you import. If you import a single application that contains circular references, which are common in menu-based systems, you must manually relink applications before you can run them.

To import a group of dependent applications, first export them as a group, so all the dependent applications are in one export file. When you import a group of applications from a single export file, all interdependent links and references are maintained.

1. In the Web UI, select **Assets**, then select **Create Asset**.
2. Under Upload Applications From Existing Files, click **Browse**, and select the application file.
3. Enter a name for the application.
   - If the file contains a single application, the application name is replaced.
   - If the file contains more than one application, the new application name is prepended to all applications. For example, if the file contains two applications, Test1 and Test2, and you enter `NewName` as the new application name, the uploaded applications are named NewName-Test1 and NewName-Test2.
4. Click **Upload**.
5. To edit application details, select **View Application Details**.

### Creating Applications from Templates

SMS Builder includes a set of application templates that you can upload and run.

1. In the Web UI navigation bar, select **Assets**.
2. Select **Create Asset**.
3. Choose a template from the list, and click **Create**.
   The template is installed, and names of the template applications appear.
4. Select **Application Details**.

After you create an application, you can run it or modify its details.

#### See also

- *Developing Quick-Start Templates* on page 76

## Exporting Applications

You can export applications to make backup copies, or to move applications to other SMS Builder installations. If you export an application, it is saved in an XML file.

#### See also

- *Importing Applications* on page 27

### Exporting a Single Application

Exporting a single application creates an XML file that contains the application configuration.

1. In the Web UI, navigate to the **Application Details** tab for the application you want to export.
2. Click **Export**.
   The application is exported to a file called `appFlow.xml` in the `Downloads` directory.

If the application you export contains references to other applications through either the Goto Application state or the Application Call state, details of the called applications are included in `appFlow.xml`; however, interapplication links may not be reestablished when you import the file. To maintain links and dependencies between applications, export them as a group.

### Exporting a Group of Applications

Exporting a group of applications maintains links and dependencies between applications.

1. In the Web UI, navigate to the **Assets** page.
2. Select the check box to the left of each application you want to export.
3. Click **Group Export Applications**.
   A file called `groupedFlow.xml`, which contains all the exported application configurations is created in the `Downloads` directory.

## Sample Applications

SAP Mobile Platform offers a customizable way to more efficiently manage financial services. It allows customers to redeem vouchers on any phone, remit money domestically, pay bills automatically, and manage their accounts remotely.

### Cash-Out Interactive Application

Use SMS to interact with the Cash-Out application. SMS Builder manages a unique user session that maintains the context of the conversation between the user and the application.

The Cash-Out application comprises multiple interactive applications. The applications are linked by either Goto Application states, in which control is passed to referenced applications, or Application Call states, in which case control moves temporarily to the referenced application, before returning to the application that called it.

A complete mobile service is created from multiple interactive applications that are validated with a customer's MSISDN. Although there is no internal customer list, back-end systems can validate customers. The Cash-Out application assumes a valid customer session exists.

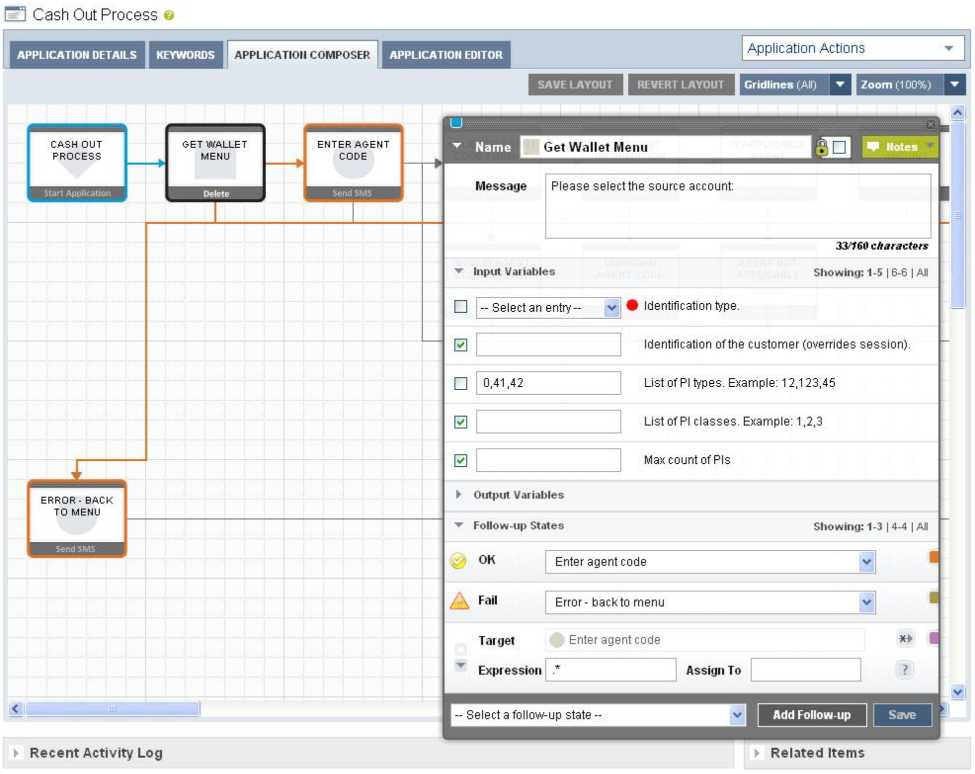
Once an application has validated a customer, it is typical to offer a series of SMS menus, from which customers can select. By default, the Cash-Out application contains one menu option that is related to the mobile financial services that are offered to customers.

The Cash-Out application:

1. Requests the account from which to withdraw cash.
2. Requests the code of the customer support agent with whom to perform the transaction.
3. Requests the transaction amount.
4. Validates and preauthorizes the transaction by verifying sufficient funds in the account, amount limits, and the agent's SVA.
5. Requests an account PIN, or transaction confirmation.
6. Sends money to the agent.
7. If a transaction fails, requests a solution to validation problems.

*Cash-Out Application State Editor*

In the Cash-Out application, the Get Wallet Menu state sends a menu to customers via SMS.



Name: Get Wallet Menu

Message: Please select the source account:

33/160 characters

Input Variables — Showing: 1-5 | 6-6 | All

| | | |
|---|---|---|
| ☐ | -- Select an entry -- | ● Identification type. |
| ✓ | | Identification of the customer (overrides session). |
| ☐ | 0,41,42 | List of PI types. Example: 12,123,45 |
| ✓ | | List of PI classes. Example: 1,2,3 |
| ✓ | | Max count of PIs |

Output Variables

Follow-up States — Showing: 1-3 | 4-4 | All

| | |
|---|---|
| OK | Enter agent code |
| Fail | Error - back to menu |

Target: Enter agent code

Expression: .*

Assign To:

-- Select a follow-up state --

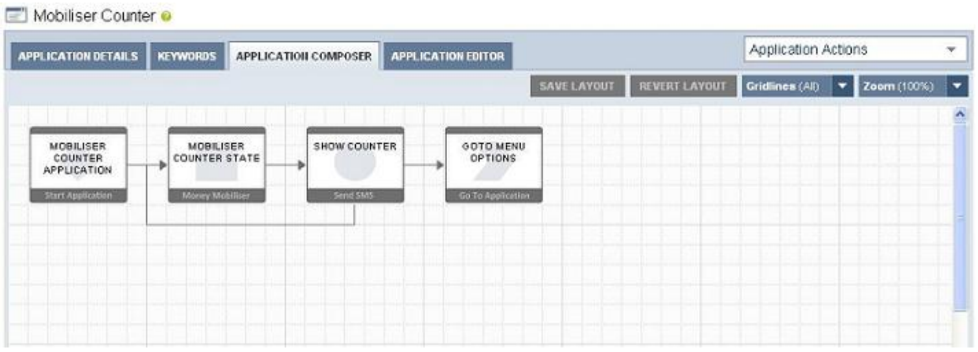
Add Follow-up

Save

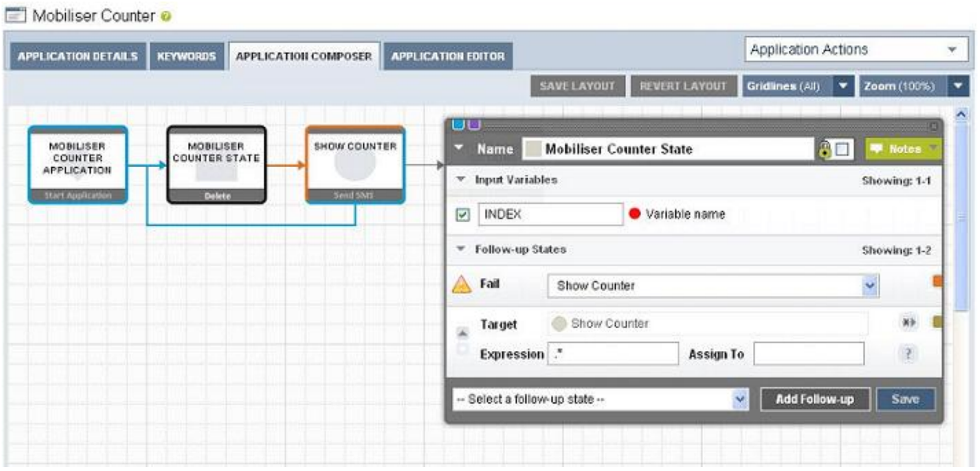
## Mobiliser Counter Interactive Application

The Mobiliser Counter sample application increments a session variable, displays the value, then either increments the value again, or exits.

You can develop the Mobiliser Counter application in the Application Composer.



The session variable **INDEX** is used as the counter variable. This variable is dynamically substituted into the text sent to mobile consumers.



Name: Mobiliser Counter State

Input Variables — Showing: 1-1

INDEX — Variable name

Follow-up States — Showing: 1-2

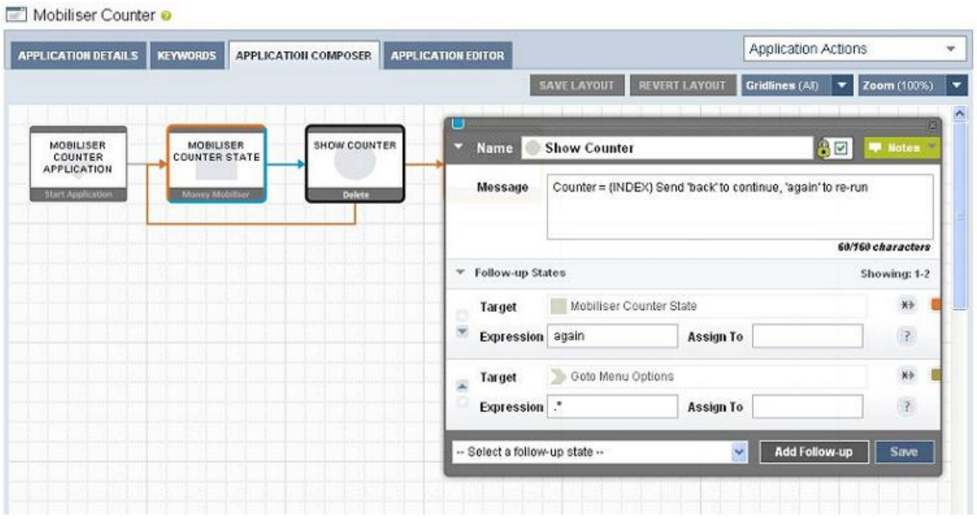
Fail: Show Counter

Target: Show Counter

Expression: .* Assign To:

-- Select a follow-up state --

If consumers respond with the keyword "again," the application loops back to the Mobiliser Counter state. Any other input causes the application to exit.



Name: Show Counter

Message: Counter = {INDEX} Send 'back' to continue, 'again' to re-run

60/160 characters

Target: Mobiliser Counter State
Expression: again
Assign To:

Target: Goto Menu Options
Expression: .*
Assign To:

## Utility Notification Event Application

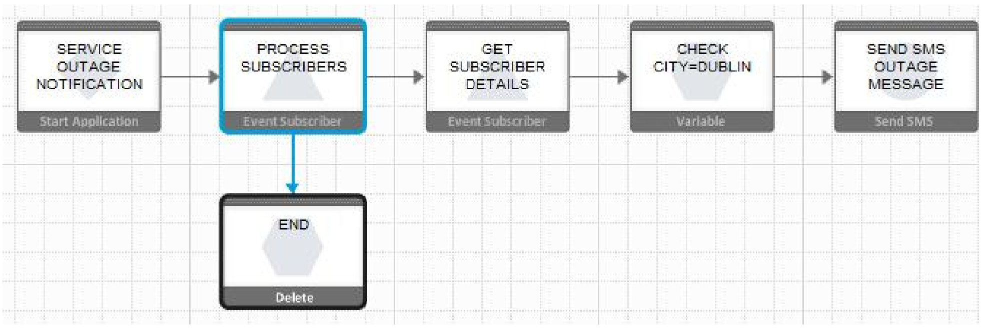
Event applications are designed for task flow or batch processing, and are typically invoked by events, such as scheduled times, system triggers, or external triggers.

For example, event applications can provide end-to-end solutions for utility companies. A common use case includes:

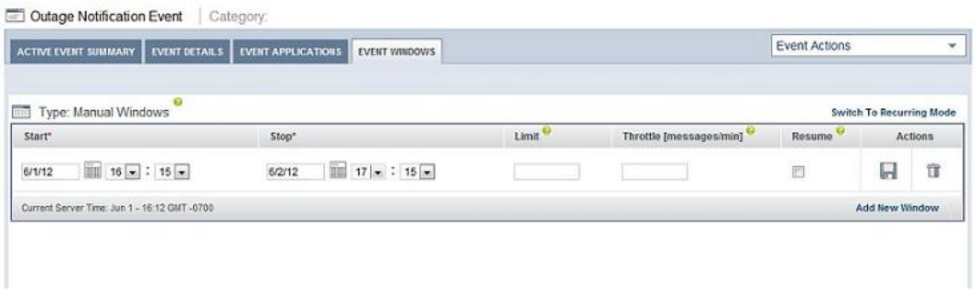
- Self-registration – register telephone numbers using SMS; for customers who did not provide their number when signing up with the company.
- Self-services – such as looking up usage history and status of move-in activation, reporting issues, and finding offices.
- Notifications – set up notifications for overdue payments, high usage, service-outage alerts, summer-savings awareness, and so on.
- Engagement – enables customers who receive notifications to reply. For example, if customers respond to overdue-payment notifications, they automatically receive 1–2 days extension; they can also authorize automatic payments.

In this example, the company's customer relationship management (CRM) system generates a list of subscribers who have opted to receive outage notifications. The list contains customer telephone numbers (MSISDNs) and cities for which an outage-notification service is provided. This list is uploaded to subscriber storage. When a service outage is planned for the city of Dublin, the Process Subscriber state retrieves subscribers from the list. For each subscriber:

1. Get Subscriber Details retrieves subscriber attributes (city).
2. Check City=Dublin filters out customers who are not in Dublin.
3. Send SMS Outage Message sends a message to Dublin customers.



Invoke the application, by assigning it to an active event, and creating an event window. Event windows can be one-time or recurring. This application has a one-time event window.


Outage Notification Event | Category:

ACTIVE EVENT SUMMARY | EVENT DETAILS | EVENT APPLICATIONS | EVENT WINDOWS

Event Actions

Type: Manual Windows

Switch To Recurring Mode

| Start* | Stop* | Limit | Throttle [messages/min] | Resume | Actions |
|---|---|---|---|---|---|
| 6/1/12 16 : 15 | 6/2/12 17 : 15 | | | | |

Current Server Time: Jun 1 - 16:12 GMT -0700

Add New Window

An alternative to manually uploading subscribers to the database is to use an event application to fetch subscribers from the system of record, and use batch processing to upload and store them in the database.

### See also

- *Sample Event Message Log* on page 27

## Developing Custom Application States

Custom state development using the State SDK is a Java development task you can perform with or without a development IDE, such as Eclipse or NetBeans. After you develop and deploy custom states, you can use them to develop applications.

Before proceeding with custom state development, verify that:

- The development environment meets system requirements—see *http://service.sap.com/pam*.
- SMS Builder is installed on the development machine. SMS Builder is required to access State SDK bundles for custom state development, and to deploy and test custom states through the development process.

Third-party software mechanisms that custom states can use include:

- Spring Framework – for application context and dependency injection.
- Spring Dynamic Modules (Spring DM) – for abstracting OSGi mechanisms.
- OSGi Services – for software-service publication and consumption.
- OSGi Configuration Admin – for container-based configuration of services and components.

## Application Life Cycle

Applications run in the processing-engine runtime container and are managed by the processing engine. Once deployed to the runtime container, applications can be invoked by either incoming messages or events. Events can be generated by the system, a scheduled time, or a call from an external Web service.

### *Starting or Restarting an Application*

For a newly started application, a new session is created, and the Application Start state is executed. Sessions are based on a consumer's MSISDN, which is typically the mobile telephone number from which the message is sent. The Application Start state is created automatically for new applications, and cannot be removed. This state performs initialization prior to executing the application. The Application Start state is typically followed by at least one state. For example, if an interactive application is invoked by an incoming message, the Application Start state processes the incoming message, and routes it to the appropriate follow-up state, based on the message value. The Application Start state can also filter messages, and save incoming message values in session attributes.

If you restart an application, the existing session is reactivated, and all session attributes are available to the application. The application continues from the last active state.

### *Executing the Current Application State*

The processing engine executes the current application state, calling either `processMessage` or `processState`; these methods contain state-specific logic.

The processing engine calls:

- `processMessage` to reactivate a state, when an external event occurs for which the state is waiting.
- `processState` when another state activates the current state through a follow-up transition.

### *Processing an Incoming Message*

If a state is reactivated by a call to its `processMessage` method, the state processes the incoming message.

For example, State 1 —> Send SMS state —> State 3. When the flow reaches the Send SMS state, a message is sent out and the flow waits for a response. When the response arrives, the processing engine calls the Send SMS state's `processMessage` method to reactivate the state. The state processes the message, finds the follow-up transition that matches the incoming message, and returns the follow-up transition state. For example, if the follow-up state is State 3, the processing engine sets the current state to State 3, and begins executing it.

### *Processing State Logic*

When a state is activated by a follow-up transition, the processing engine calls the `processState` method, which contains the core logic of the state. If the state needs to call an external Web service, you implement the call in the `processState` method.

States do not return objects from the `processState` method. Instead, they set flags using the helper object `SmappStateProcessingAction`, which is an input parameter to the method. For example, if the state-logic processing is successful, the state calls `continueProcessing(followUpState)`, passing the name of the follow-up state as `followUpState`.

The processing engine sets the current state to the value of **followUpState**, and executes the current state.

To determine the follow-up state, you can call either of two methods provided by the utility class `StateUtils`, which is included in the State SDK:

- `determineFollowingSmappStateFromPattern`
- `determineFollowingSmappStateFromTransitionType`

In addition to calling `continueProcessing`, states can call:

- `terminateProcessing` – if a severe error occurs and the application must be terminated.
- `waitForMessage` – if the state sends a message and must wait for the response.

### *Terminating Conditions*

The processing engine continues through the application flow until it meets one of these terminating conditions:

- No follow-up transition
- Call to `terminateProcessing`
- Call to `waitForMessage`

The first two conditions terminate the application. A call to `waitForMessage` pauses the application until a response is received, and the session hibernates. When the response message arrives, the life cycle restarts.

For event applications, if the processing engine encounters no follow-up transition, it checks the preconfigured terminating criteria to determine whether to stop, or keep the session alive and generate a callback to repeat from the Application Start state.

## Developing and Deploying Custom States

Develop and deploy custom states to extend the functionality of SMS Builder, and to meet client-specific requirements.

1. Develop a custom state by extending either:
   - `SmappStatePlugin` class – for most states.
   - `AbstractDynamicMenu` class – for menu states.
2. (Classes that extend `SmappStatePlugin` only) Implement the state logic.
3. Add custom state information.
4. Define custom state variables.
5. Set up Apache Maven.
6. Build and deploy a custom state bundle.

### Extending the SmappStatePlugin Class

You can simplify custom-state development by extending the `SmappStatePlugin` class.

If you develop a custom state by extending the `SmappStatePlugin` class, you must:

- Implement the state logic.
- Provide the state information: ID, name, revision number, and usage notes.
- Specify the input attributes.
- Specify the output attributes.
- Customize the state follow-up transitions, if they are different from the default transitions.

**See also**

- *Sample Custom State* on page 81
- *Sample GetMyWeather State* on page 79
- *Implementing State Logic* on page 42

#### *StatePlugin Interface*

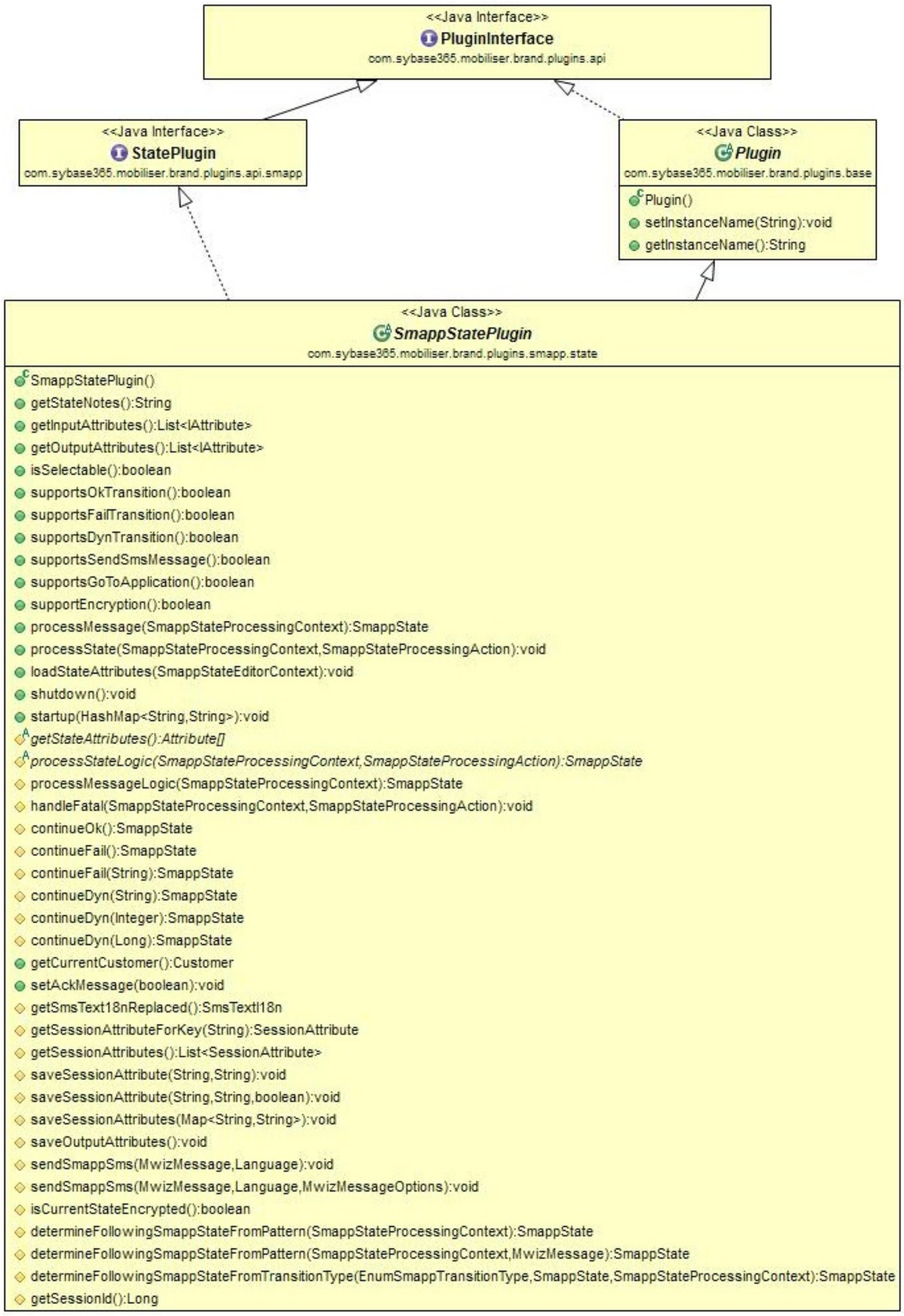
You can use the `StatePlugin` interface to develop application states.

The `SmappStatePlugin` class is a base abstract class that implements the `StatePlugin` interface. Most custom states should extend `SmappStatePlugin`, which provides basic implementations that are common to most custom states, as well as helper methods that are commonly used in state implementations.

Two important methods in the `StatePlugin` interface are `processMessage` and `processState`, which are integral parts of application life cycles. Some of the methods in the `StatePlugin` interface customize the state editor, for example, `supportsOkTransition` and `getStateNotes`.

If a custom state extends the `SmappStatePlugin` class, implementing the class is simplified significantly. Instead of implementing both `processMessage` and `processState` methods, you can focus on adding business logic to the `processStateLogic` method. This is sufficient in most custom-state implementations.

**Note:** Do not extend the abstract class `Plugin`. Instead, extend `SmappStatePlugin`.


<<Java Interface>>
PluginInterface
com.sybase365.mobiliser.brand.plugins.api
<<Java Interface>>
StatePlugin
com.sybase365.mobiliser.brand.plugins.api.smapp
<<Java Class>>
Plugin
com.sybase365.mobiliser.brand.plugins.base
Plugin()
setInstanceName(String):void
getInstanceName():String
<<Java Class>>
SmappStatePlugin
com.sybase365.mobiliser.brand.plugins.smapp.state
SmappStatePlugin()
getStateNotes():String
getInputAttributes():List<IAttribute>
getOutputAttributes():List<IAttribute>
isSelectable():boolean
supportsOkTransition():boolean
supportsFailTransition():boolean
supportsDynTransition():boolean
supportsSendSmsMessage():boolean
supportsGoToApplication():boolean
supportEncryption():boolean
processMessage(SmappStateProcessingContext):SmappState
processState(SmappStateProcessingContext,SmappStateProcessingAction):void
loadStateAttributes(SmappStateEditorContext):void
shutdown():void
startup(HashMap<String,String>):void
getStateAttributes():Attribute[]
processStateLogic(SmappStateProcessingContext,SmappStateProcessingAction):SmappState
processMessageLogic(SmappStateProcessingContext):SmappState
handleFatal(SmappStateProcessingContext,SmappStateProcessingAction):void
continueOk():SmappState
continueFail():SmappState
continueFail(String):SmappState
continueDyn(String):SmappState
continueDyn(Integer):SmappState
continueDyn(Long):SmappState
getCurrentCustomer():Customer
setAckMessage(boolean):void
getSmsText18nReplaced():SmsTextI18n
getSessionAttributeForKey(String):SessionAttribute
getSessionAttributes():List<SessionAttribute>
saveSessionAttribute(String,String):void
saveSessionAttribute(String,String,boolean):void
saveSessionAttributes(Map<String,String>):void
saveOutputAttributes():void
sendSmappSms(MwizMessage,Language):void
sendSmappSms(MwizMessage,Language,MwizMessageOptions):void
isCurrentStateEncrypted():boolean
determineFollowingSmappStateFromPattern(SmappStateProcessingContext):SmappState
determineFollowingSmappStateFromPattern(SmappStateProcessingContext,MwizMessage):SmappState
determineFollowingSmappStateFromTransitionType(EnumSmappTransitionType,SmappState,SmappStateProcessingContext):SmappState
getSessionId():Long

### *PluginInterface Interface*

If you develop a custom state by extending the `SmappStatePlugin` class, it implements the `PluginInterface` interface.

Plug-in components must have at least one class that implements the `PluginInterface`. Components that implement `PluginInterface` are automatically loaded into the messaging server and started. During start-up, the server calls the `startup` method of the implementing class, which allows the class to perform any necessary setup.

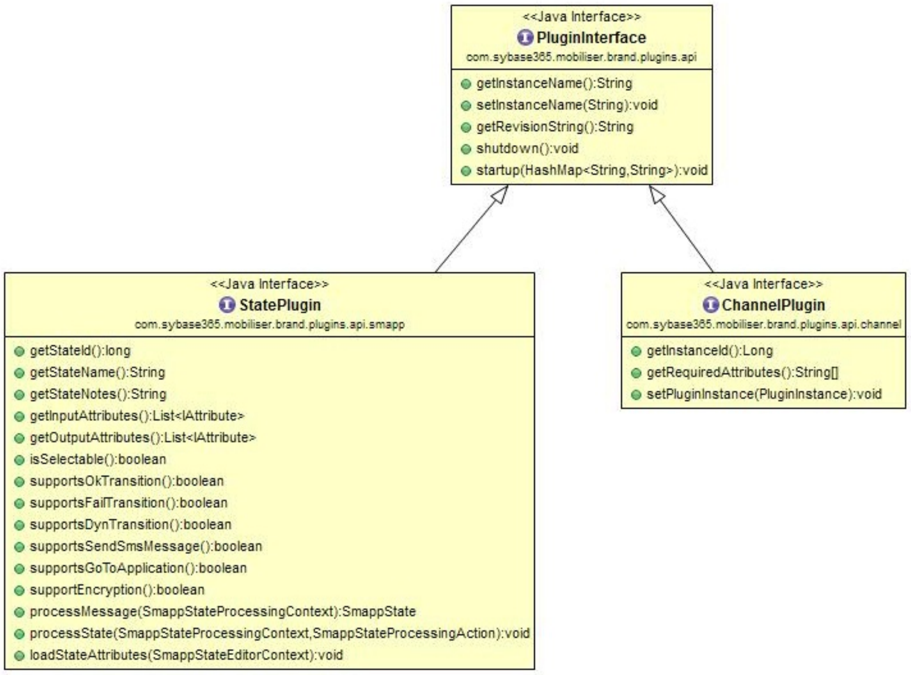
`PluginInterface` methods are:

- `getInstanceName():String`
- `setInstanceName(String):void`
- `getRevisedString():String`
- `shutdown():void`
- `startup(HashMap<String,String>):void`

The `shutdown` method is called when the server is shutting down, giving the implementation a chance to perform housecleaning, such as persisting cache data.

`getInstanceName`, `setInstanceName`, and `getRevisedString` are placeholders only. The component must implement the appropriate functionality.

`StatePlugin` and `ChannelPlugin` implementations extend `PluginInterface` and define their specific interfaces. You can use `StatePlugin` APIs to develop custom states. The `ChannelPlugin` interface is reserved for SAP internal development only.


<<Java Interface>>
PluginInterface
com.sybase365.mobiliser.brand.plugins.api
getInstanceName():String
setInstanceName(String):void
getRevisionString():String
shutdown():void
startup(HashMap<String,String>):void
<<Java Interface>>
StatePlugin
com.sybase365.mobiliser.brand.plugins.api.smapp
getStateId():long
getStateName():String
getStateNotes():String
getInputAttributes():List<IAttribute>
getOutputAttributes():List<IAttribute>
isSelectable():boolean
supportsOkTransition():boolean
supportsFailTransition():boolean
supportsDynTransition():boolean
supportsSendSmsMessage():boolean
supportsGoToApplication():boolean
supportEncryption():boolean
processMessage(SmappStateProcessingContext):SmappState
processState(SmappStateProcessingContext,SmappStateProcessingAction):void
loadStateAttributes(SmappStateEditorContext):void
<<Java Interface>>
ChannelPlugin
com.sybase365.mobiliser.brand.plugins.api.channel
getInstanceId():Long
getRequiredAttributes():String[]
setPluginInstance(PluginInstance):void

## Extending the AbstractDynamicMenu Class

Many SMS and Unstructured Supplementary Service Data (USSD) applications rely on menus to receive consumer responses. Menus reduce the potential for response errors, because they are numbered lists.

The `AbstractDynamicMenu` class simplifies the development of custom-menu states that extend the class, because they inherit:

- A list of menu items
- Menus and indexes that are generated automatically and recalculated on each page
- Methods to send menus as SMS messages
- These variables:
  - **Show Exit Menu** – an input variable that specifies whether to allow recipients to exit the menu.
  - **Variable Name of the Selected Key** – an output variable representing the menu selection, which is stored as a key-value pair object. Key is the unique key of the menu item, which may be used later in the application.
  - **Variable Name of the Selected Value** – an output variable that represents the value of the selected key.

Custom states that extend the `AbstractDynamicMenu` class must implement these methods:

- `constructMenuList()` – gets the menu list.
- `init()` – initializes the state.
- `getStateAttributeList()` – gets the list of attributes.
- `saveSessionVariables()` – explicitly saves session variables.

Message recipients can select from lists, and reply using index numbers. If a menu has more than four items, it includes a pagination option, which displays the next four items in the list. On the last page, selecting the pagination option returns to the first page. Selecting the exit option abandons a list without a response; the application task flow determines the follow-up transition. To force recipients to choose an item from the list, you can disable the exit option.

In a typical custom-state implementation that extends the `SmappStatePlugin` class, you implement state logic in the `processStateLogic` method. However, when you extend the `AbstractDynamicMenu` class, both `processStateLogic` and `processMessageLogic` methods are implemented by the abstract class. These methods contain the menu processing logic, and are declared as `final`, so they cannot be overridden.

### See also

- *Sample Custom-Menu State* on page 83

#### *AbstractDynamicMenu Life Cycle*

The life cycle of the `AbstractDynamicMenu` class is based on the life cycle of the `SmappStatePlugin` class; however, there are slight differences in menu functionality.

If you extend the `AbstractDynamicMenu` class, it implements the `processMessageLogic` method and the `processStateLogic` method.

1. The `processStateLogic` method calls the `init` method.
2. `processStateLogic` calls both the `constructMenuList` and `saveSessionVariables` methods.
3. The `SmappStatePlugin::getStateAttributes` method calls `getStateAttributeList`, which aggregates the attributes returned by the method with attributes defined in the `AbstractDynamicMenu` class, such as the input exit-menu item and the output key-value pair.
4. An `AbstractDynamicMenu` state is initially activated as a follow-up transition from a previous state, so the processing engine calls its `processStateLogic` method. The `init` and `constructMenuList` methods are called sequentially to initialize and construct the menu. Eventually, the menu is sent as an SMS message, and the processing engine waits for the response. The consumer selects a menu item.
5. If `constructMenuList` returns only a single item, the state immediately calls `saveSessionVariables`, and proceeds with the default dynamic follow-up

transition. You can customize the state's default behavior by overriding the `continueWhenSingleEntry` method.

6. When a response arrives, the processing engine calls the state's `processMessageLogic` method, which calls `constructMenuList` to assemble the menu and interpret the selected menu item. If the selection is a valid menu item, `saveSessionVariables` is called. The state prepares the selected-item details for output, and proceeds with the follow-up transition, as returned by the `saveSessionVariables` method. If null is returned, the default OK follow-up transition is used.

## Implementing State Logic

If you extend the `SmappStatePlugin` class, implement state logic in the `processStateLogic` method. If you extend the `AbstractDynamicMenu` class, the abstract class implements the state logic.

At runtime, the processing engine calls a state's `processState` method, which in turn calls `processStateLogic`. The `processState` method is implemented by the `SmappStatePlugin` abstract class.

The `processStateLogic` method signature is:

```
protected SmappState processStateLogic(
                      SmappStateProcessingContext context,
                      SmappStateProcessingAction action)
  throws MwizProcessingException, DBException;
```

The `processStateLogic` input parameters are:

- `SmappStateProcessingContext` – provides access to resources, such as data-access objects for session variables.
- `SmappStateProcessingAction` – signals to the processing engine that there is to be additional processing.

### See also

- *Extending the SmappStatePlugin Class* on page 36

### *SmappStateProcessingContext*

The processing engine `SmappStateProcessingContext` object provides access to resources, such as session variables and the subscribers data store.

You can use the `SmappStateProcessingContext` object to share resources between the processing engine and the state; however, in most state implementations, this is unnecessary.

**Note:** Do not alter `SmappStateProcessingContext`.

You can use these `SmappStateProcessingContext` methods:

- `getStateDao` – inserts, updates, or deletes session variables.
- `getSubscriberDao` – accesses the subscribers data store. Also used by some built-in states.
- `isAckMessageRequested` – queries whether an acknowledgment is requested.
- `setAckMessageRequest` – specifies whether an acknowledgment is requested.
- `isCurrentStateEncrypted` – queries whether state data is encrypted.

The following resources are available for read-only access, and include no API support. Do not access these resources directly, or make any changes. If you have special requirements, consult with SAP support services.

- `client`
- `session`
- `clientMsisdn`
- `currentState`
- `customer`
- `langDefault`
- `matchingPattern`
- `mr`
- `msg`
- `newSession`

Do not use the following methods or resources; doing so may result in errors or unexpected application behavior:

- `getlangRequest`
- `updateSession`
- `cacheMgr`
- `outgoingQueue`

### *SmappStateProcessingAction*

The `SmappStateProcessingAction` class controls state and application processing. Use it to signal the processing engine that further processing is intended.

The processing engine recognizes three signaling actions: continue, wait, and terminate, which you can send by calling:

- `continueProcessing (SmappState)` – continues execution to the specified follow-up state. Causes an infinite loop if the follow-up state is the same as the calling state. Termination must be handled within the state.
- `waitForMessage()` – pauses execution and waits for a response, then continues execution to the specified follow-up state.
- `terminateProcessing ()` – terminates the application.

States that extend the `SmappStatePlugin` class, implementing logic inside the `processStateLogic` method need not explicitly call `continueProcessing` or `terminateProcessing`. The same functionality is accomplished by returning the follow-up state from the `processStateLogic` method. For example, instead of calling `continueProcessing`, return the follow-up state using one of the helper methods:

- `continueOk()`
- `continueFail()`
- `continueDyn()`

To terminate processing, states should call `continueFail`, and let the state-editor configuration determine what to do. If the state is not configured to forward `continueFail` calls to a follow-up state, the application automatically terminates.

**Note:** If a state calls `waitForMessage` before it returns null from the `processStateLogic` method, the application does not terminate, because the state is waiting for a response. For this reason, SAP recommends that you do not let states return null.

To enable states to send messages and wait for replies before they continue processing, call `waitForMessage`.

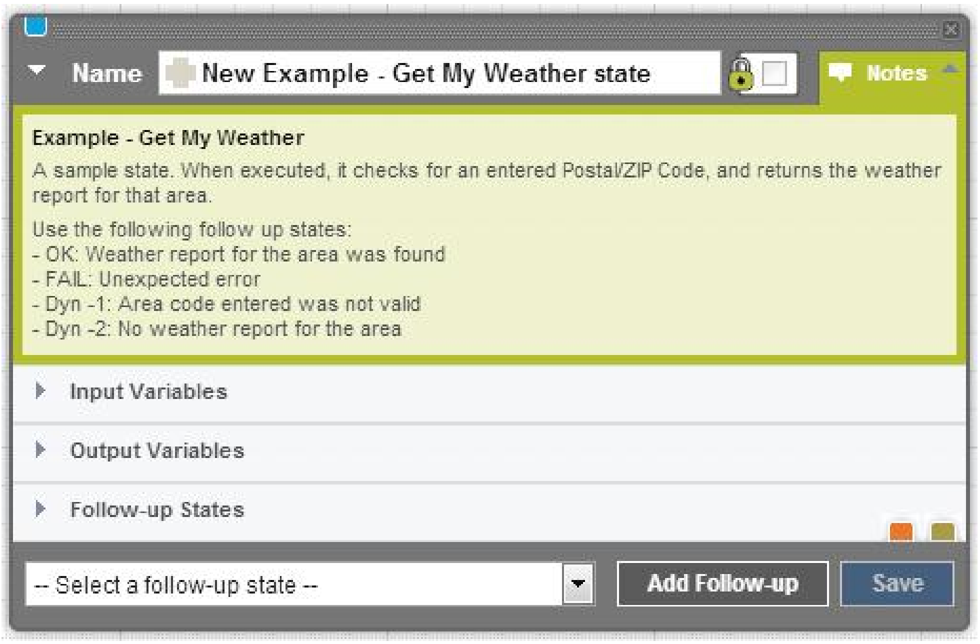
To display a message control in the state editor, call `supportsSendSmsMessage`.

## Custom State Information

State information includes an ID, a name, a revision number, and usage notes. The name and usage notes are metadata that the state editor shows in the Application Composer.

For a custom state, you can explain its purpose and functionality as state notes, which appear in the state editor.

```
@Override
public String getStateNotes() {
  StringBuilder sb = new StringBuilder();
  sb.append("A sample state. When executed, it checks for ");
  sb.append("an entered Postal/ZIP Code, and returns the ");
  sb.append("weather report for that area.\n\n);
  sb.append("Use the following follow up states:\n ");
  sb.append("- OK: Weather report for the area was found\n ");
  sb.append("- FAIL: Unexpected error\n ");
  sb.append("- Dyn -1: Area code entered was not valid\n ");
  sb.append("- Dyn -2: No weather report for the area\n ");
  return sb.toString();
}
```



Example - Get My Weather

A sample state. When executed, it checks for an entered Postal/ZIP Code, and returns the weather report for that area.

Use the following follow up states:
- OK: Weather report for the area was found
- FAIL: Unexpected error
- Dyn -1: Area code entered was not valid
- Dyn -2: No weather report for the area

The revision number is a prerequisite for any plug-in component, as specified in the `PluginInterface` class. It identifies a version, and sets the plug-in number. `getRevisionString()` can return any String value.

```
@Override
public String getRevisionString() {
  return "1.0.0";
}
```

The state ID is a unique identifier for the state. Each state must have a unique ID stored in the database for each installation in which the state is used. This unique value allows the state to be resolved to the same type across installations.

```
private static long STATE_ID = 600000L;

@Override
public long getStateId() {
  return STATE_ID;
}
```

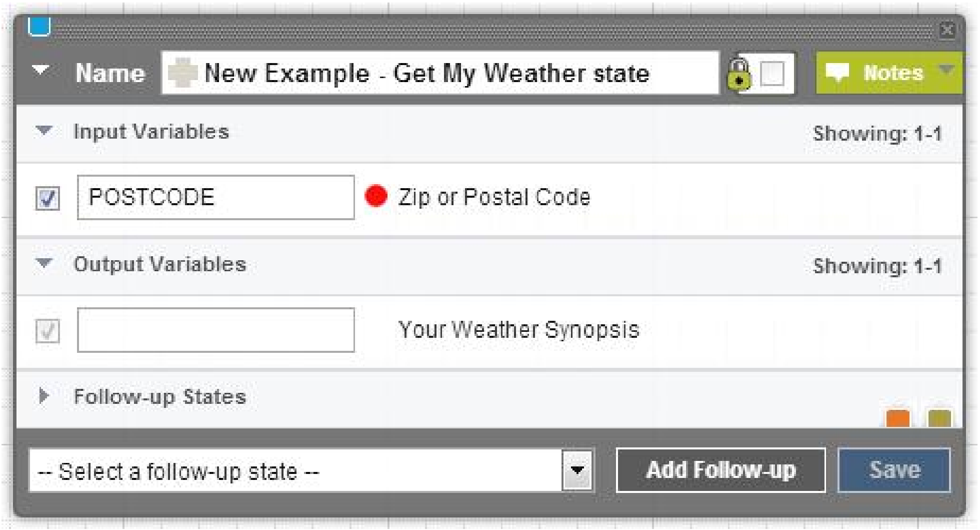
For custom states, assign unique ID values between 600,000 and 999,999. Values between 0 and 599,999 are reserved.

## Custom State Variables

You can define input and output variables for custom states. Variables are used as both metadata in the state editor, and as runtime objects for storing session variables.

In the `GetMyWeather` sample custom state, one input variable (Zip or Postal Code) and one output variable (Your Weather Synopsis) are defined in the code, and appear in the state editor view.

```
// Define input variable

private static final TextBoxAttribute inPostCode =
       new TextBoxAttribute("POSTCODE", "Zip or Postal Code", false);


// Define output variable

private static final OutputAttribute outWeather =
        new OutputAttribute("WEATHER", "Your Weather Synopsis");

private static Attribute[] stateAttr;

static {
   stateAttr = new Attribute[] {inPostCode, outWeather};
}

@Override
protected Attribute[] getStateAttributes() {
   return stateAttr.clone();
}
```



Name: New Example - Get My Weather state

Input Variables — Showing: 1-1

✓ POSTCODE ● Zip or Postal Code

Output Variables — Showing: 1-1

✓ Your Weather Synopsis

Follow-up States

-- Select a follow-up state --

`getStateAttributes` is an abstract helper method that the `SmappStatePlugin` class implements. It aggregates both input and output variables. The base class derives the required `getInputAttributes` and `getOutputAttributes` methods from `getStateAttributes`, based on the attribute-type class. The state editor uses the attribute array that the `getStateAttributes` method returns to render input and output variables. The `saveOutputAttributes` method saves output attributes from the attribute array.

All variables (input and output) have input controls that appear on the state editor. The `public String getText()` method returns the text from input controls.

**See also**

- *Sample GetMyWeather State* on page 79
- *Input and Output Parameters* on page 6
- *Defining Input Variables* on page 48
- *Defining Output Variables* on page 50
- *Accessing Input Variables* on page 51
- *List Variables* on page 52

### *Variables for Troubleshooting*

When you develop custom states, include error output variables that can help you troubleshoot problems in the production environment.

To facilitate debugging, include output variables in the state code for an error message, a unique error ID, and a service code. If the state calls an external Web service, for example, the Web service can return a code in the service-code output variable.

```
// Define output variables

private static final OutputAttribute outErrMsg =
                  new OutputAttribute("ERR_MSG", "Error Message");
private static final OutputAttribute outErrUUID =
                   new OutputAttribute("ERR_UUID", "Error Unique ID");

private static final OutputAttribute outSvcCode =
                  new OutputAttribute("SVC_CODE", "Service Code");

// some code omitted here…

@Override
protected SmappState processStateLogic(…)
{
  // Logic implementation

  try {
    // Reset the error output variable
    outErrMsg.setHoldValue("");
    outErrUUID.setHoldValue("");
    saveOutputAttributes();
    return continueOk();
  }
  catch (Exception ex) {
    String uuid = UUID.randomUUID().toString();
    log.error(ex.getMessage()+ " [UUID={}]", uuid);
    outErrMsg.setHoldValue(message);
    outErrUUID.setHoldValue(uuid);
    saveOutputAttributes();
    return continueFail();
  }
}
```

UUID is a unique user ID that you can use to report errors. For example, if an error occurs, an SMS message can be sent to the consumer, who is identified by the UUID. Consumers can call customer support to report issues, using their UUID. UUIDs are logged so they can be correlated with reported issues.

### *Defining Input Variables*

States use input variables to get input values, either from a session variable or as a constant. You can configure the behavior in the state editor. The `InputAttribute` class manages input variables.

In addition to the basic properties, input variables have an **isOptional** property. If set to true, the input variable is optional; false indicates it is mandatory.

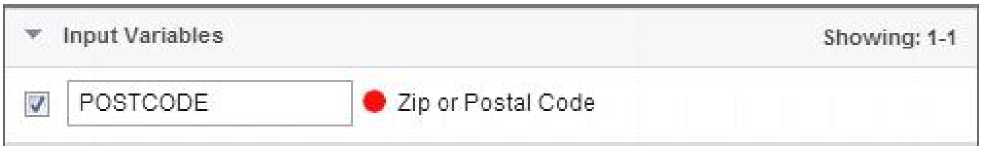
The input variable constructor is:

```
InputAttribute (String  id,  String  description, boolean isOptional)
```

Two types of input variables exist, text box input controls and selection input controls.

### *Text Box Input Controls*

Text boxes manage either a single constant value or a value that is accessed from a session variable.



You can create the input variable in the example above using this constructor:

```
TextBoxAttribute( " POSTCODE " , " Zip or Postal Code " , false);
```

By default, the variable ID is automatically assigned to the `TextBoxAttribute` control. In this case, the ID is `POSTCODE`. The description, `Zip or Postal Code`, appears to the right. The red dot indicates that the input variable is mandatory.

**Note:** If input is mandatory and a session variable name is specified, a runtime error is thrown if the session variable does not exist. The processing engine terminates the application, unless the state implementation handles `RequiredParameterMissingException`, with either `continueFail` or `continueDyn` follow-up transitions.

The state of the check box tells the processing engine how to process an input variable:

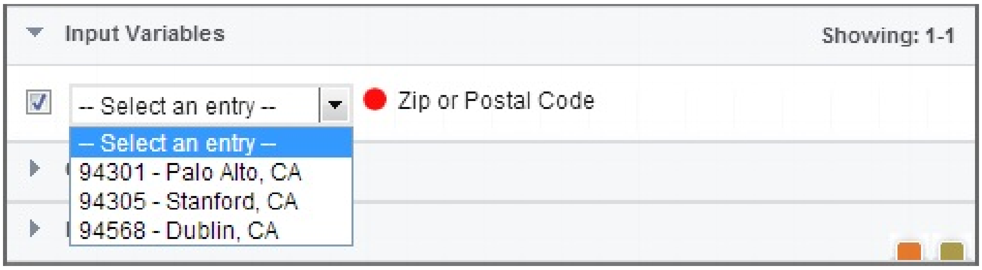
- Selected – retrieve the value from the named session variable.
- Not selected – use the constant value.

If you use a state twice in the same application, and if the state saves a value in a session variable, change the session-variable name in the second instance, so it does not overwrite the value.

To find the session-variable name, hover the mouse over the description text; pop-up text includes the variable description and the variable name.

### *Selection Input Controls*

Selection input controls manage constant values that are selected from a list of options. Lists are populated in the state code.



Unique IDs are automatically assigned as the session-variable name; you cannot change them, and they do not appear in the state editor. To find the session-variable name, hover the mouse over the description text; pop-up text includes the variable description and the variable name.

To use a state twice in the same application, and save the value of the session variable, you can call the `Copy Variables` state to copy the session variable to another variable.

The check box performs the same function as it does for text box controls. The red dot indicates that an input selection is mandatory.

**See also**

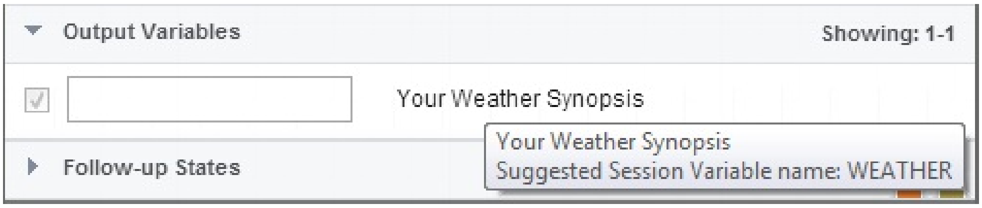
- *Input and Output Parameters* on page 6
- *Custom State Variables* on page 46
- *Defining Output Variables* on page 50
- *Accessing Input Variables* on page 51
- *List Variables* on page 52

### *Defining Output Variables*

States return results as output variables, which are always session variables. Only states can set output variables, and only at runtime. Output-variable check boxes are always selected and cannot be modified.

To create an output variable, use the `OutputAttribute` constructor:

```
OutputAttribute("WEATHER", "Your Weather Synopsis")
```



By default, output session-variable names are not set, so text boxes are empty. You can set values by calling either of these two methods:

- `setValue` – creates a session variable (if none exists), and saves the value immediately in the database, or,
- `setHoldValue` – temporarily holds the value in the cache, until you explicitly call the `SmappStatePlugin::saveOutputAttributes` method.

The `saveOutputAttributes` method saves multiple session variables with a single database connection. If the state has only a few output variables, call the `setValue` method. If there are many output variables, call `setHoldValue`; this may impact the efficiency of the state at runtime.

To set output variables, call one of the methods in the `OutputAttribute` class:

- `public void setValue (String val)`
- `public void setValue (Long val)`
- `public void setValue (Integer val)`
- `public void setValue (Boolean val)`
- `public void setHoldValue (String val)`
- `public void setHoldValue (Long val)`
- `public void setHoldValue (Integer val)`
- `public void setHoldValue (Boolean val)`

**See also**

- *Input and Output Parameters* on page 6
- *Custom State Variables* on page 46
- *Defining Input Variables* on page 48
- *Accessing Input Variables* on page 51

- *List Variables* on page 52

### *Accessing Input Variables*

You can access input variables that are in a custom state using either the `getInputValue` method or the `getInputValueWithWarning` method.

The signatures of the methods you can call to access input variables are:

```
public InputValue getInputValue()
   throws DBException;
public InputValue getInputValueWithWarning()
   throws DBException, RequiredParameterMissingException;
```

To retrieve optional input variables, call `getInputValue`. A null value is returned if either an input variable is not provided, or if the session variable that the input variable is assigned to does not exist.

```
InputValue iv = optionalVar.getInputValue();

if (iv != null) {
  retrieve the value
}
```

To retrieve mandatory input variables, call `getInputValueWithWarning`. The exception `RequiredParameterMissingException` is raised if either an input variable is not provided, or if the session variable that the input variable is assigned to does not exist. You can retrieve all mandatory input variables in the same `try/catch` block.

```
try {
  Long id = mandatoryIdVar.getInputValueWithWarning().getLong();
  Integer count =
mandatoryCountVar.getInputValueWithWarning().getInt();
}
catch (RequiredParameterMissingException rex) {
  log.error(rex.getMessage());
  return continueFail();
}
```

**Note:** The `RequiredParameterMissingException::getMessage` method indicates the mandatory variable that is missing.

Both methods that access input variables return the `InputValue` class. `InputValue` methods return values that you enter in the state editor when you configure an input attribute; return values can be either constants or session-variable names:

- `InputValue.getString();`
- `InputValue.getString(int size);`
- `InputValue.getLong();`
- `InputValue.getInt();`
- `InputValue.getBoolean();`
- `InputValue.getDouble();`

- `InputValue.getMsisdn();`

**See also**

- *Input and Output Parameters* on page 6
- *Custom State Variables* on page 46
- *Defining Input Variables* on page 48
- *Defining Output Variables* on page 50
- *List Variables* on page 52

## List Variables

List variables do not appear in the state editor. You can use list variables to save lists of the `BeanConverterInterface` type to session variables.

As an example, the `AbstractDynamicMenu` class uses a list variable to persist an SMS menu. The `BeanConverterInterface` specifies that a bean must provide string serialization and deserialization logic. Each `BeanConverterInterface` item is saved as a session variable with a unique name.

```
package com.sybase365.mobiliser.brand.plugins.smapp.beans;
public interface BeanConverterInterface<T> {
  T convert(String value);
  String convert(T object);
}
```

**Note:** Strings returned by the `convert(T object)` method must be less than 1000 characters.

The `SessionVariableAttribute` class has two methods: `getList` and `setList`. The `getList` method retrieves a list from the database. When `setList` is called, the list is saved to a session variable, which requires a database connection.

**Note:** Lists are saved outside of transactions. Therefore, if an exception occurs, the method throws a `DBException`, and a partial list may be saved. It is up to the state implementation that uses this attribute to retry.

Most state implementations do not need list variables. They are needed only if a state can transition into an internal waiting condition by calling `waitForMessage`. For example, list variables are most commonly used when sending SMS messages. Calling `waitForMessage` causes the application to hibernate until the response arrives. The list variable is saved to a session variable, so it is available when the application is reactivated.

**See also**

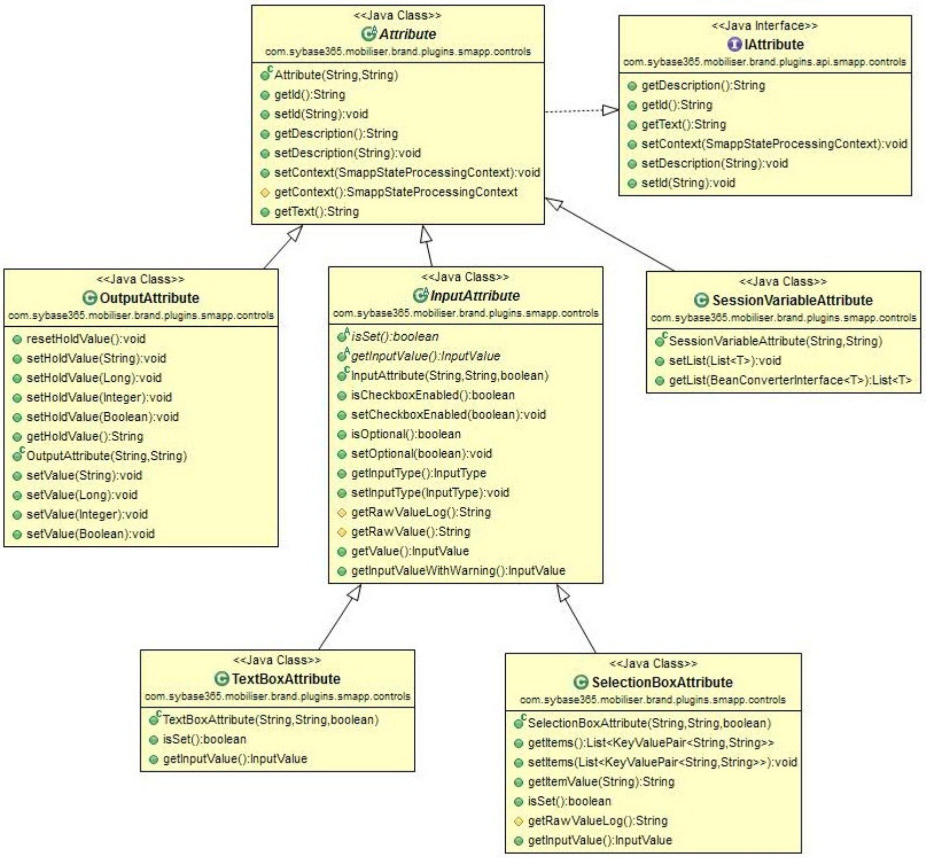
- *Input and Output Parameters* on page 6
- *Custom State Variables* on page 46
- *Defining Input Variables* on page 48
- *Defining Output Variables* on page 50

- *Accessing Input Variables* on page 51

### *State Attributes Class Hierarchy*

All state variables that are derived from the `Attribute` class are identified by an ID and a description, which are defined in the constructor `Attribute(String ID, String Description)`. ID is a unique identifier of the attribute; for `InputAttribute`, ID defaults to the session variable name. The value of the Description variable appears in the Application Composer.

The diagram below illustrates the attribute class hierarchy.



<<Java Class>>
Attribute
com.sybase365.mobiliser.brand.plugins.smapp.controls
Attribute(String,String)
getId():String
setId(String):void
getDescription():String
setDescription(String):void
setContext(SmappStateProcessingContext):void
getContext():SmappStateProcessingContext
getText():String
<<Java Interface>>
IAttribute
com.sybase365.mobiliser.brand.plugins.api.smapp.controls
getDescription():String
getId():String
getText():String
setContext(SmappStateProcessingContext):void
setDescription(String):void
setId(String):void
<<Java Class>>
OutputAttribute
com.sybase365.mobiliser.brand.plugins.smapp.controls
resetHoldValue():void
setHoldValue(String):void
setHoldValue(Long):void
setHoldValue(Integer):void
setHoldValue(Boolean):void
getHoldValue():String
OutputAttribute(String,String)
setValue(String):void
setValue(Long):void
setValue(Integer):void
setValue(Boolean):void
<<Java Class>>
InputAttribute
com.sybase365.mobiliser.brand.plugins.smapp.controls
isSet():boolean
getInputValue():InputValue
InputAttribute(String,String,boolean)
isCheckboxEnabled():boolean
setCheckboxEnabled(boolean):void
isOptional():boolean
setOptional(boolean):void
getInputType():InputType
setInputType(InputType):void
getRawValueLog():String
getRawValue():String
getValue():InputValue
getInputValueWithWarning():InputValue
<<Java Class>>
SessionVariableAttribute
com.sybase365.mobiliser.brand.plugins.smapp.controls
SessionVariableAttribute(String,String)
setList(List<T>):void
getList(BeanConverterInterface<T>):List<T>
<<Java Class>>
TextBoxAttribute
com.sybase365.mobiliser.brand.plugins.smapp.controls
TextBoxAttribute(String,String,boolean)
isSet():boolean
getInputValue():InputValue
<<Java Class>>
SelectionBoxAttribute
com.sybase365.mobiliser.brand.plugins.smapp.controls
SelectionBoxAttribute(String,String,boolean)
getItems():List<KeyValuePair<String,String>>
setItems(List<KeyValuePair<String,String>>):void
getItemValue(String):String
isSet():boolean
getRawValueLog():String
getInputValue():InputValue

These methods are reserved for use by the processing engine:

- `public void setContext(SmappStateProcessingContext context)`
- `protected SmappStateProcessingContext getContext()`

`SmappStateProcessingContext` is the running context of the application, set by the processing engine using the `setContext` method. The

`SmappStateProcessingContext` object provides access to the data source that stores session variables.

## Setting Up Apache Maven

Apache Maven is a software project management tool that is based on a project object model (POM). You can use Maven to manage a project's build, reporting, and documentation from a central piece of information.

Install and configure Apache Maven, and deploy the State SDK bundles, so you can build custom-state bundles and deploy them to the server.

### *Installing Apache Maven*

You can download Apache Maven from the Apache Maven Project Web site. Apache Maven version 3.0.4 has been tested and certified with SMS Builder.

1. Navigate to *http://maven.apache.org/download.cgi*, and download Apache Maven.
2. To verify that your Apache Maven installation is successful, on the command line, run:

   ```
   mvn -version
   ```

   The output looks similar to:

```
Apache Maven 3.0.4 (r1232337; 2012-01-17 00:44:56-0800)
Maven home: C:\ZPrograms\apache-maven-3.0.4 Java version: 1.6.0_35,
vendor: Sun Microsystems Inc.
Java home: C:\Program Files\Java\jdk1.6.0_35\jre
Default locale: en_US, platform encoding: Cp1252
OS name: "windows 7", version: "6.1", arch: "amd64", family: "windows"
```

**Next**

Configure Apache Maven.

### *Configuring Apache Maven*

You can customize where Maven looks for dependencies by editing the Maven configuration file.

**Prerequisites**

Install Apache Maven.

**Task**

By default, Maven looks for dependencies in its central repository; however, in some cases, it may need additional repositories. For example, some companies have their own internal Maven repositories, and you, as a developer, must find these dependencies. The central Maven repository is open to the public, and its libraries are either open source or available for public use. SMS Builder SDK libraries are not hosted in the central Maven repository, nor in any publicly accessible Maven repository.

1. Navigate to your Apache Maven installation directory, and open the `conf\setting.xml` file.
2. Enter these lines:

```
<settings>

 <profiles>
  <profile>
   <id>brand_state_development</id>
   <repositories>
    <repository>
     <id>EclipseLink</id>
     <name>Eclipse Link</name>
     <url>http://download.eclipse.org/rt/eclipselink/maven.repo</url>
    </repository>
   </repositories>
  </profile>
 </profiles>

  <activeProfiles>
    <activeProfile>brand_state_development</activeProfile>
  </activeProfiles>

</settings>
```

3. To add a Maven dependency location, between the `<repositories></repositories>` elements, add a `<repository></repository>` element pair.
4. For the new repository, define:
   - `id` – repository ID.
   - `name` – name of the repository.
   - `url` – Internet location of the repository.

Maven creates a default-user local cache repository in `${user.home}\.m2\repository`, where *user.home* depends on the operating system. For example, on a Windows 7 machine, the *user.home* location is `C:\Users\`**userName**. During the build process, this is the first location Maven searches for dependency libraries. Initially, the local repository is empty. During the first build, Maven does not find libraries in the local repository, so it looks in the Maven central repository, which is, by default, *http://search.maven.org/#browse*. Maven downloads any dependency libraries to the local repository, then uses them in the build. Subsequent builds are faster, because dependency libraries have been downloaded to the local repository.

**Next**
Deploy State SDK bundles to Maven repositories.

### *Deploying State SDK Bundles to a Maven Repository*

You can deploy State SDK bundles to the local Maven repository (also known as the `.m2`). Deploy bundles to local repositories on each development machine.

#### Prerequisites

Install and configure Apache Maven.

#### Task

The State SDK consists of five bundles:

- `mobiliser-brandplugin-api-1.3.1.jar`
- `mobiliser-brandstate-sdk-1.3.1.jar`
- `mobiliser-brandplugin-security-1.3.1.jar`
- `mobiliser-brandplugin-core-1.3.1.jar`
- `mobiliser-brandplugin-jpa-1.3.1.jar`

Deploy these bundles to the Maven repository so they are accessible as dependencies to state-development projects. Bundles are in the *`SMSBUILDER_HOME`*`\bundle\application` directory. To deploy the bundles, run a script for each bundle, or copy all five scripts to a single script file, and run it once.

**Note:** Scripts are for Windows only; to run on Linux, modify the `-Dfile` path.

1. Change to the *`SMSBUILDER_HOME`* directory.
2. Run:

```
mvn install:install-file -Dfile=bundle\application\mobiliser-brand-
plugin-api-1.3.1.jar
-DgroupId=com.sybase365.mobiliser.brand.plugins -DartifactId=mobiliser-
brand-plugin-api
-Dversion=1.3.1 -Dpackaging=jar

mvn install:install-file -Dfile=bundle\application\mobiliser-brand-
state-sdk-1.3.1.jar
 -DgroupId=com.sybase365.mobiliser.brand.plugins -DartifactId=mobiliser-
brand-state-sdk
 -Dversion=1.3.1 -Dpackaging=jar

mvn install:install-file -Dfile=bundle\application\mobiliser-brand-
security-1.3.1.jar
 -DgroupId=com.sybase365.mobiliser.brand.security -
DartifactId=mobiliser-brand-security
 -Dversion=1.3.1 -Dpackaging=jar

mvn install:install-file -Dfile=bundle\application\mobiliser-brand-
core-1.3.1.jar
 -DgroupId=com.sybase365.mobiliser.brand.core -DartifactId=mobiliser-
brand-core -Dversion=1.3.1
 -Dpackaging=jar
```

```
mvn install:install-file -Dfile=bundle\application\mobiliser-brand-
jpa-1.3.1.jar
 -DgroupId=com.sybase365.mobiliser.brand.database -
DartifactId=mobiliser-brand-jpa
 -Dversion=1.3.1 -Dpackaging=jar
```

## Custom State Bundles

Package custom states as OSGi bundles, so you can deploy them.

An OSGi bundle is a JAR file with extra manifest headers that can be deployed in the OSGi container. A custom-state bundle can contain one or more custom states, and it must be packaged as an OSGi bundle before you can deploy it to SMS Builder.

### *Building Custom State Bundles*

After you develop custom states, and set up Apache Maven, build OSGi bundles that you can deploy to SMS Builder.

1. *Creating Maven Projects*

   The main artifacts of a Maven project are the project object model (POM) file, and folders that contain source-code files.
2. *Customizing Maven POM Files*

   Customize a Maven project object model (POM) file to create and build custom-state OSGi bundles to deploy to SMS Builder.
3. *Creating Maven Project Artifacts*

   After you create a Maven project, create project artifacts to use in a custom-state bundle.
4. *Building Maven Projects*

   You can build Maven projects on the command line, or you can use Maven build and unit test projects in an IDE.
5. *Declaring States as Spring Beans*

   Developing a custom-state bundle requires that you declare each state as a Spring Framework bean in the beans-context.xml file. A state is any Java class that either directly or indirectly extends the SmappStatePlugin abstract class.
6. *Configuring Bean Properties*

   The bean properties file, properties-context.xml, declares all properties that must be retrieved from the OSGi configuration administration service during runtime; properties are stored in the service so they can be configured dynamically at runtime.
7. *Registering States as OSGi Services*

   To enable SMS Builder to discover states at runtime, register them as OSGi services, by declaring them in the services-context.xml file.

### Creating Maven Projects

The main artifacts of a Maven project are the project object model (POM) file, and folders that contain source-code files.

You can create a new Maven project on the command line, or in any IDE that supports Maven. To create a Maven project on the command line:

```
mvn archetype:create -DgroupId=com.sap.example -DartifactId=customState
```

where:

- *groupId* – names the package.
- *artifactId* – names the project and the project folder.

As the project is created, you see progress messages. For example:

```
[INFO] Scanning for projects...
Downloading: http://repo.maven.apache.org/maven2/org/apache/maven/
plugins/
maven-clean-plugin/2.4.1/maven-clean-plugin-2.4.1.pom
Downloaded: http://repo.maven.apache.org/maven2/org/apache/maven/
plugins/
maven-clean-plugin/2.4.1/maven-clean-plugin-2.4.1.pom (5 KB at 6.8 KB/
sec)
[...]
[INFO]
[INFO]------------------------------------------------------------------
------
[INFO] Building Maven Stub Project (No POM) 1
[INFO]------------------------------------------------------------------
------
[INFO]
[INFO] --- maven-archetype-plugin:2.2:create (default-cli) @ standalone-
pom ---
[...]
[INFO]------------------------------------------------------------------
------
[INFO] BUILD SUCCESS
[INFO]------------------------------------------------------------------
------
[INFO] Total time: 41.155s
[INFO] Finished at: Mon Oct 22 17:00:49 PDT 2012
[INFO] Final Memory: 8M/245M
```

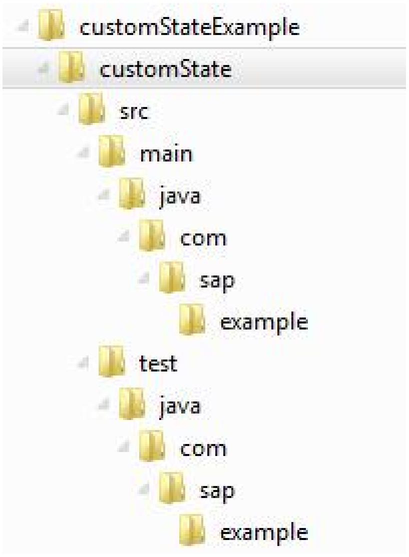
#### See also

- *Customizing Maven POM Files* on page 60
- *Creating Maven Project Artifacts* on page 64
- *Sample Maven POM File* on page 61
- *Maven Project Structure* on page 59

### *Maven Project Structure*

When you create a Maven project, the directory structure that is created includes the project object model (POM) file.

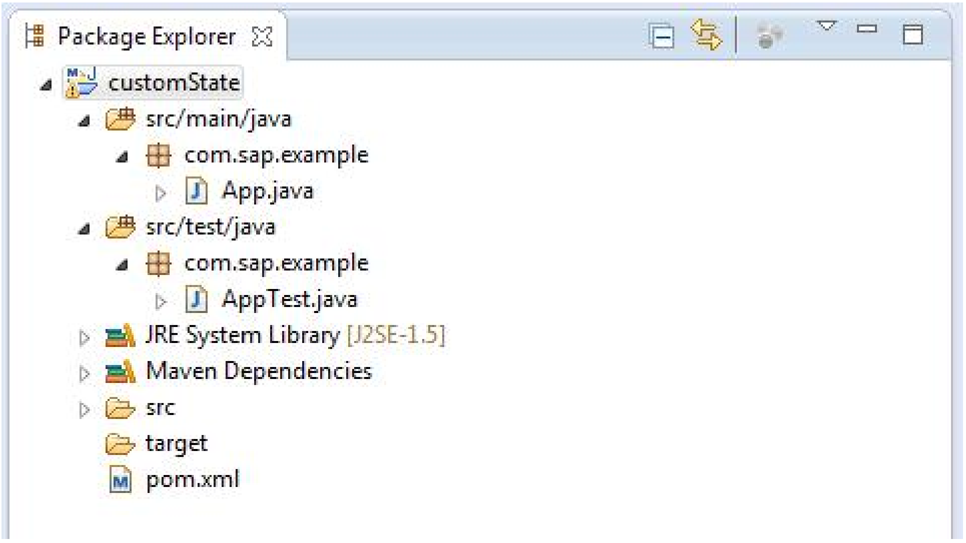
In this sample project, the *groupId* is set to `com.sap.example`. This directory structure is created automatically for a new project:



Two Java files, `App.java` and `AppTest.java`, are created in the `example` folders, under `main` and `test`, respectively. The POM file, which contains the initial project configuration, is created in the `customState` folder. You can use this POM file as a starting point for custom-state development.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.sap.example</groupId>
  <artifactId>customState</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>customState</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

You can open or import a newly created Maven project into your IDE. Eclipse and NetBeans both support Maven. The image below shows the sample project structure in Eclipse.



Once you are familiar with the structure and the content of POM files, you can create them manually. You can also create a new project in any IDE that supports Maven.

**See also**

- *Sample Maven POM File* on page 61
- *Creating Maven Projects* on page 58
- *Creating Maven Project Artifacts* on page 64

## *Customizing Maven POM Files*

Customize a Maven project object model (POM) file to create and build custom-state OSGi bundles to deploy to SMS Builder.
Edit the Maven `pom.xml` file for your project to define:

- `groupId` – package name.

- `artifactId` – name of the project.
- `version` – version number of the project.
- `packaging` – `bundle`.
- `name` – name of the state.

For example:

```
<groupId>com.sap.example</groupId>
<artifactId>customState</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>bundle</packaging>
<name>Custom State</name>
```

**See also**

- *Creating Maven Projects* on page 58
- *Creating Maven Project Artifacts* on page 64

### *Sample Maven POM File*

A Maven project object model (POM) file contains all the required information for Maven to create and build OSGi bundles that you can deploy to SMS Builder.

This POM file (`pom.xml`) illustrates the basic configuration for a custom-state bundle. The state implementation does not need libraries other than those provided by the SDK. The SDK libraries are shown as dependencies. The contents of the original POM are shown in **bold**:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.sap.example</groupId>
<artifactId>customState</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>bundle</packaging>
<name>Custom State</name>
<url>http://www.sap.com</url>

<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <bundle.namespace>${project.groupId}</bundle.namespace>
  <bundle.symbolicName>${bundle.namespace}.${project.artifactId}</
bundle.symbolicName>
  <brand.version>1.3.1</brand.version>
</properties>

<build>
 <defaultGoal>install</defaultGoal>
 <plugins>
  <plugin>
   <artifactId>maven-compiler-plugin</artifactId>
   <version>2.3.2</version>
   <configuration>
     <source>1.6</source>
     <target>1.6</target>
   </configuration>
  </plugin>

  <!-- Create an OSGi Bundle Manifest -->
  <plugin>
    <groupId>org.apache.felix</groupId>
    <artifactId>maven-bundle-plugin</artifactId>
    <version>2.3.7</version>
    <extensions>true</extensions>
    <configuration>
     <instructions>
     <manifestLocation>META-INF</manifestLocation>
     <Bundle-Category>object</Bundle-Category>
    <Bundle-SymbolicName>${bundle.symbolicName}</Bundle-SymbolicName>

     <Bundle-Version>${project.version}</Bundle-Version>
     <Embed-Dependency></Embed-Dependency>

     <!--
     Note: When you develop additional classes within this object
     bundle, include the package names of the classes in either the
     Export-Package, or the Private-Package, otherwise it will not
     be included in the bundle.
     -->

     <Export-Package>
     </Export-Package>

     <Private-Package>
       com.sap.example
     </Private-Package>

     <DynamicImport-Package>
     </DynamicImport-Package>

     <!--
     Note: If you use other only referenced from spring context then
     include them in the Import-Package instruction here. The *
     instruction ensures that any directly imported packages in
     supporting classes are included automatically, but the Spring
     context referenced ones need explicit reference.
     -->
     <Import-Package>
       *
     </Import-Package>

     <!--
     Each module can override these defaults in an
     optional osgi.bnd file
     -->
     <_include>-osgi.bnd</_include>

    <!--
    Enable viewing of the properties file content from telnet console
    -->
    <ARF-Bundle-Template>/META-INF/config</ARF-Bundle-Template>


    </instructions>
    <obrRepository>NONE</obrRepository>
   </configuration>
  </plugin>
 </plugins>
</build>

<dependencies>
 <dependency>
  <groupId>com.sybase365.mobiliser.brand.plugins</groupId>
  <artifactId>mobiliser-brand-plugin-api</artifactId>
  <version>${brand.version}</version>
 </dependency>
 <dependency>
  <groupId>com.sybase365.mobiliser.brand.plugins</groupId>
  <artifactId>mobiliser-brand-state-sdk</artifactId>
  <version>${brand.version}</version>
 </dependency>
 <dependency>
  <groupId>com.sybase365.mobiliser.brand.security</groupId>
  <artifactId>mobiliser-brand-security</artifactId>
  <version>1.3.1</version>
 </dependency>
 <dependency>
  <groupId>com.sybase365.mobiliser.brand.core</groupId>
  <artifactId>mobiliser-brand-core</artifactId>
  <version>${brand.version}</version>
 </dependency>
 <dependency>
  <groupId>com.sybase365.mobiliser.brand.database</groupId>
  <artifactId>mobiliser-brand-jpa</artifactId>
  <version>${brand.version}</version>
 </dependency>

 <!-- Logging -->
 <dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
  <version>1.6.6</version>
 </dependency>

 <!-- Optional for Unit Test -->
 <dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>3.8.1</version>
  <scope>test</scope>
 </dependency>
</dependencies>

 <!--
 Required Javax Persistence dependencies not available
 from Maven central repository
 -->
 <profiles>
  <profile>
   <activation>
    <jdk>[1.5, 1.7)</jdk>
   </activation>
   <dependencies>
    <dependency>
     <groupId>org.eclipse.persistence</groupId>
     <artifactId>javax.persistence</artifactId>
     <version>2.0.4.v201112161009</version>
     <scope>provided</scope>
    </dependency>
   </dependencies>
   <repositories>
    <repository>
     <id>EclipseLink</id>
     <url>http://download.eclipse.org/rt/eclipselink/maven.repo</url>
    </repository>
   </repositories>
  </profile>
 </profiles>
</project>
```

**See also**

- *Maven Project Structure* on page 59
- *Creating Maven Projects* on page 58
- *Creating Maven Project Artifacts* on page 64

## *Creating Maven Project Artifacts*

After you create a Maven project, create project artifacts to use in a custom-state bundle.

### Prerequisites

Create a Maven project.

### Task

1. In the `example` subdirectory under `main`, delete the `App.java` file.
2. In the `example` subdirectory, under `test`, delete the `AppTest.java` file.
3. In the `main` directory, create a subdirectory called `resources`.

   The `resources` directory stores configuration files that SMS Builder needs when it loads state bundles.
4. In the `resources` directory, create these subdirectories:
   - `META-INF` – contents are packaged in the state bundle.

- `META-INF\spring` – stores a configuration file that the Spring Framework uses.
- `META-INF\sample\conf` – stores sample configuration property files; if you copy these files to *`SMSBUILDER_HOME`*`\conf\cfgload`, SMS Builder can load them dynamically.

  Configuration files are specific to a bundle. They identify what states and configurations to load, and how to link them together.

5. In the `test` directory, create these subdirectories:
   - `java`
   - `resources`

**See also**

- *Customizing Maven POM Files* on page 60
- *Creating Maven Projects* on page 58
- *Sample Maven POM File* on page 61
- *Maven Project Structure* on page 59

### *Building Maven Projects*

You can build Maven projects on the command line, or you can use Maven build and unit test projects in an IDE.

For information about building projects using Maven in the Eclipse IDE, see *http://maven.apache.org/eclipse-plugin.html*.

On the command line, run:

```
mvn clean install
```

As the project builds, you see progress messages:

```
[INFO] Scanning for projects...
Downloading: http://repo.maven.apache.org/maven2/org/apache/felix/maven-
bundle-plugin/
2.3.7/maven-bundle-plugin-2.3.7.pom
Downloaded: http://repo.maven.apache.org/maven2/org/apache/felix/maven-
bundle-plugin/
2.3.7/maven-bundle-plugin-2.3.7.pom
(4 KB at 15.0 KB/sec)
[…]
 [INFO] Installing C:\ZMobiliser\customStateExample\customState\target
\customState-1.0-SNAPSHOT.jar
to C:\Users\I824993\.\m2\repository\com\sap\example\customState\1.0-
SNAPSHOT\customState-1.0-SNAPSHOT.jar
[INFO] Installing C:\ZMobiliser\customStateExample\customState\pom.xml
to
C:\Users\I824993\.m2\repository\com\sap\example\customState\1.0-SNAPSHOT
\customState-1.0-SNAPSHOT.pom
[INFO]
[INFO] --- maven-bundle-plugin:2.3.7:install (default-install) @
customState ---
[INFO] Local OBR update disabled (enable with -DobrRepository)
```

```
[INFO]
------------------------------------------------------------------------

[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------------

[INFO] Total time: 36.332s
[INFO] Finished at: Mon Oct 29 10:48:50 PDT 2012
[INFO] Final Memory: 11M/242M
[INFO]
------------------------------------------------------------------------
```

The bundle JAR file is saved in the `\`**className**`\target` directory; its name is derived from the Maven project *artifactId* and version. For this example, the filename is `customState-1.0-SNAPSHOT.jar`.

### Declaring States as Spring Beans

Developing a custom-state bundle requires that you declare each state as a Spring Framework bean in the `beans-context.xml` file. A state is any Java class that either directly or indirectly extends the `SmappStatePlugin` abstract class.

You can configure Spring beans by setting properties, or by creating other beans that support state operations.

1. Edit the `beans-context.xml` file to add a `<bean>` element for each state. Define:
   - `id` – name of the state.
   - `class` – name of the Java class that implements the state.

   For example:

```
 <bean id="SampleState" class="com.sap.example.SampleState">
   <property name="country" value="${sample.country}"/>
 </bean>
...
```

2. (Optional) Declare state properties, and assign either constant values or references to the values that are defined in the `properties-context.xml` file.

   The value of the *country* property is a reference to the *sample.country* property defined in `properties-context.xml`.

### Configuring Bean Properties

The bean properties file, `properties-context.xml`, declares all properties that must be retrieved from the OSGi configuration administration service during runtime; properties are stored in the service so they can be configured dynamically at runtime.

You can reconfigure states at runtime, without reloading state bundles or restarting the server. However, state developers must implement dynamic reconfiguration, by defining state properties in the code.

Edit the `properties-context.xml` file to configure bean properties:

a) Set `osgix:cm-properties id` to the name of the OSGi configuration administration service property that is identified by the value of `persistent-id`.

   SMS Builder initializes the property, and loads the property file identified by the value of `persistent-id`.

b) For each property, enter a `<prop key>` element and default value.

   Properties are initialized with values from the OSGi configuration administration service. If a property does not exist in the service, the default value is used.

c) Set the value of `ctx:property-placeholder properties-ref` to the value of `osgix:cm-properties id`.

   The value identifies a list of properties that are available for the Spring Framework to use during state initialization.

For example:

```
<osgix:cm-properties id="sampleState-cfg" persistent-
id="service.sampleState">
  <prop key="sample.country">US</prop>
</osgix:cm-properties>

<ctx:property-placeholder properties-ref="sampleState-cfg"/>
```

**Note:** SAP recommends that you store a copy of the `properties-context.xml` file in the `META-INF\sample\conf` directory.

### *Registering States as OSGi Services*

To enable SMS Builder to discover states at runtime, register them as OSGi services, by declaring them in the `services-context.xml` file.

Registered states are discoverable by the `StatePlugin` interface:

```
com.sybase365.mobiliser.brand.plugins.api.smapp.StatePlugin
```

Edit `services-context.xml`, and set OSGi service properties:

- `id` – name of the service.
- `ref` – name of the state.
- `interface` – name of the class that implements the `StatePlugin` interface.

For example:

```

<osgi:service id="SampleStateService" ref="SampleState"

interface="com.sybase365.mobiliser.brand.plugins.api.smapp.StatePlugin"/
>
```

### *Deploying State Bundles*

To deploy custom-state bundles, make the files available to SMS Builder at runtime, and configure the states to start automatically.

1. Copy the bundle `.jar` files to `SMSBUILDER_HOME\bundle\application`.

   This directory contains all the bundles that are deployed to the runtime environment.

   **Note:** System bundles are installed in `SMSBUILDER_HOME\bundle`.

2. Edit the `SMSBUILDER_HOME\conf\config.properties` file to add the new custom-state file to the list of bundles that are started automatically.

```
felix.auto.start.15 = ${aims.app.dir}\customState-1.0-SNAPSHOT.jar
```

   All state bundles are listed in the `config.properties` file. SMS Builder reinitializes its bundle cache each time it starts.

3. Restart the server.

   To verify that no errors occurred, check these log files:
   - `brand.log`
   - `felix.log`
   - `spring.log`
   - `persist.log`

   If there are errors, check the Spring configuration and the `\import\private \dynamic` package specifications.

**Next**

To verify that bundles resolve and start, use either Telnet or the AIMS System Web console (both require access to localhost).

### *Verifying Deployment Using Telnet*

Use Telnet to verify that custom-state bundles resolve and start. The Telnet interface listens only on the localhost port, which ensures runtime environment security.

1. On the command line, run:

   ```
   telnet localhost 5365
   ```

2. At the Telnet prompt, run:

   ```
   felix:lb
   ```

   You see output similar to the following; the state of the bundle, in this case `customState`, is Active:

```
START LEVEL 20
ID|State      |     Level|Name
 0|Active     |         0|System Bundle (4.0.3)
 1|Active     |        14|activemq-core (5.5.1)
 2|Active     |        14|activemq-pool (5.5.1)
```

```
 3|Active     |        14|activemq-ra (5.5.1)
 4|Active     |        14|activemq-spring (5.5.1)
 5|Active     |        14|ARF :: System :: arf-sys (0.3.4)
 6|Active     |        14|ARF :: System :: arf-util-commands (0.3.2)
 7|Active     |        14|ARF :: System :: cm-bridge (0.3.4)
 8|Active     |        14|Java Activation API (1.1.1)
 9|Active     |        14|Java Messaging System API (1.1.0)
10|Active     |        14|CGLIB Code Generation Library (2.2.0)
11|Active     |        14|AOP Alliance API (1.0.0)
12|Active     |        14|Commons Pool (1.5.6)
 ...
108|Active    |         1|ARF :: System :: cm-loader (0.3.4)
109|Resolved  |         1|AIMS :: Object :: SMS Builder Felix JRE System
Package Support (1.3.1)
110|Installed |        10|AIMS :: Object :: SMS Builder Quartz OSGi Support
(1.3.1)
111|Active    |        17|Restlet API (2.0.13.0)
112|Active    |        17|Restlet Extension - Servlet (2.0.13.0)
113|Active    |        17|Restlet Extension - Spring Framework (2.0.13.0)

114|Active    |        17|Restlet Extension - JSON (2.0.13.0)
115|Active    |        17|AIMS :: Service :: SMS Builder Core REST Services
(1.3.1)
116|Active    |        16|AIMS :: Object :: Web Core (0.1.9)
117|Active    |        16|AIMS :: Object :: Web API and Model (0.1.9)
118|Active    |        16|AIMS :: Process :: SMS Builder Webadmin UI
(1.3.1)
119|Active    |        15|customState (1.0.0.SNAPSHOT)
```

*Verifying Deployment Using the AIMS Web Console*

In a development environment, you can use the AIMS System Web console to verify that custom-state bundles resolve and start. To ensure runtime environment security, the console restricts access, based on a list of allowable IP addresses. By default, only localhost is accessible.

### Prerequisites

Enable the AIMS System Web console.

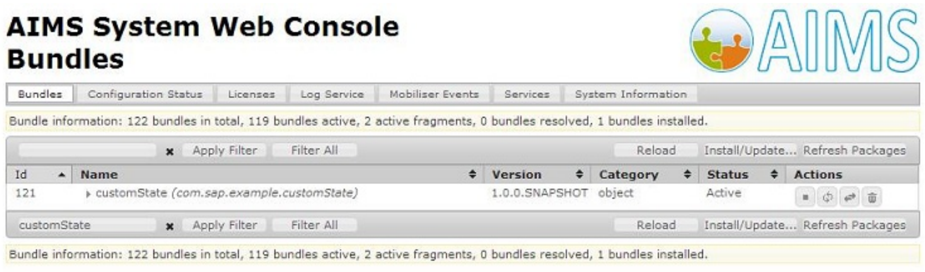
### Task

1. (Optional) To add IP addresses that the console can access:
   a) Edit the `org.apache.felix.webconsole.internal.servlet.OsgiManager.properties` file.
   b) Add IP addresses to the `allowed.ip.list`, as a comma-separated list.
2. In a Web browser, connect to `http://localhost:8080/system/console`.

   If you added other IP addresses, you can connect using one of them.
3. In the AIMS System Web console, enter these credentials:

- User name – `sybase365`
- Password – `fr4nt1c`

The **Bundles** tab lists all installed bundles. The Status of the `customState` bundle is Active.



**AIMS System Web Console**
**Bundles**

Bundle information: 122 bundles in total, 119 bundles active, 2 active fragments, 0 bundles resolved, 1 bundles installed.

| Id | Name | Version | Category | Status | Actions |
|---|---|---|---|---|---|
| 121 | customState *(com.sap.example.customState)* | 1.0.0.SNAPSHOT | object | Active | |

customState

Bundle information: 122 bundles in total, 119 bundles active, 2 active fragments, 0 bundles resolved, 1 bundles installed.

4. To view details about a bundle, click the bundle name.

   The console displays metadata, created by the Maven Bundle Plug-in (from the bundle's manifest file), package wiring, and services information.

### *Enabling the AIMS System Web Console*

During development, you can use the AIMS System Web console to inspect deployed bundles, registered configurations, and the OSGi container. By default, the Web console is disabled.

1. Edit the *`SMSBUILDER_HOME`*`\conf\config.properties` file, and uncomment these lines:

```
# Uncomment to aid in debugging container issues.
#felix.auto.start.6 = \
#${aims.app.dir}/aims-felix-webconsole-1.0.2.jar \
#${aims.app.dir}/event-webconsole-1.0.3-SNAPSHOT.jar
```

2. Copy the `org.apache.felix.webconsole.internal.servlet.OsgiManager.properties` file to the `conf/cfgbackup` folder.

#### Next

See *http://felix.apache.org/site/apache-felix-web-console.html*.

## *Configuring State Bundles*

You can configure state bundles in the `service.`**bundle**`.properties` file, where **bundle** is the name of the state bundle.

#### Prerequisites

Deploy the state bundle.

### Task

1. Edit the `service.`**bundle**`.properties` file.
2. Copy the file to the *`SMSBUILDER_HOME`*`\conf\cfgload` directory.
   When the SMS Builder server restarts, the files in the `\conf\cfgload` directory are moved to `\conf\cfgbackup`, and all properties are reconfigured.

### Next

Verify the new configuration using either Telnet or the AIMS Web System console.

#### *Verifying Bundle Configuration Using Telnet*

You can use Telnet to verify that state bundle configuration changes are in effect.

1. On the command line, run:
   ```
   telnet localhost 5365
   ```
2. At the Telnet prompt, run:
   ```
   aims:cmlist
   ```
   You see:

```
Configuration list:
org.apache.felix.webconsole.internal.servlet.OsgiManager
   file:bundle/application/aims-felix-webconsole-1.0.2.jar
service.event.quartz
   file:bundle/application/event-scheduler-quartz-1.0.3.jar
org.ops4j.pax.logging
   file:bin/pax-logging-service-1.6.9.jar
service.webui.security
   file:bundle/application/web-core-0.1.9.jar
service.sampleState
   file:bundle/application/customState-1.0-SNAPSHOT.jar
service.brand_webapp
   file:bundle/application/mobiliser-brand-webadmin-ui-1.3.1.war
service.mobiliserCustomer.states.plugin  null
service.mobiliserCustomer.client.plugin  null
service.dsprovider
   file:bundle/application/dbcp-osgi-service-1.3.1.jar
service.coreprocessing
   file:bundle/application/mobiliser-brand-processing-1.3.1.jar
org.ops4j.pax.web
   file:bundle/application/pax-web-jetty-bundle-1.1.4.jar
service.event.core
   file:bundle/application/event-core-1.0.3.jar
```

   In the output above, the service process ID (PID) for the `customState-1.0-SNAPSHOT.jar` is `service.sampleState`.

3. To see the `customState-1.0-SNAPSHOT.jar` configuration, run:
   ```
   aims:cmget service.sampleState
   ```
   You see:

```
Configuration for service (pid) "service.sampleState"
(bundle location = file:bundle/application/customState-1.0-
SNAPSHOT.jar)

key              value
------           ------
service.pid      service.sampleState
sample.country   US
arf.filename     service.sampleState.properties
```

If you set the *<ARF-Bundle-Template>* property in the Maven POM file, you can view the sample properties file that is packaged in the state bundle. Sample property files generally contain documentation for each property.

4. To find all state bundles that have sample property templates, run:

   ```
   aims:template
   ```

   You see:

```
Bundles with configuration templates:
ID: 39  Bundle:com.sybase365.mobiliser.thirdparty.smppapi
ID: 49  Bundle:com.sybase365.mobiliser.brand.processing.mobiliser-brand-
processing
ID: 51  Bundle:com.sybase365.mobiliser.brand.database.mobiliser-brand-
jpa
ID: 52  Bundle:com.sybase365.mobiliser.brand.database.mobiliser-brand-
jpa-eclipselink
ID: 56  Bundle:com.sybase365.mobiliser.framework.event-store-db-
provider
ID: 57  Bundle:com.sybase365.mobiliser.framework.event-store-jpa
ID: 58  Bundle:com.sybase365.mobiliser.framework.event-store-
eclipselink
ID: 60  Bundle:com.sybase365.mobiliser.brand.osgi.dbcp-osgi-service
ID: 117 Bundle:com.sybase365.mobiliser.brand.service.mobiliser-brand-
rest-core
ID: 118 Bundle:com.sybase365.aims.webui.web-core
ID: 120 Bundle:com.sybase365.mobiliser.brand.webadmin.mobiliser-brand-
webadmin-ui
ID: 121 Bundle:com.sap.example.customState
```

5. To see more information about the `com.sap.example.customState` bundle, run:

   ```
   aims:template 121
   ```

### *Verifying Bundle Configuration Using the AIMS Web Console*

You can use the AIMS System Web console to verify that state bundle configuration changes are in effect.

#### Prerequisites

Enable the AIMS System Web console.

### Task

1. In a Web browser, connect to `http://localhost:8080/system/console`.
2. In the AIMS System Web console, enter these credentials:
   - User name – `sybase365`
   - Password – `fr4nt1c`
3. Select the **Configuration Status** tab, then select the **Configuration** tab.

   You see all state-bundle configurations.

## *Custom State Bundle Samples*

Many custom-state implementations are based on a service-oriented architecture, in which the custom states consume existing Web services, either SOAP or Representational State Transfer (REST)ful types. States can either get results from one Web service, or they can aggregate results from multiple Web service calls.

### *Consuming SOAP Web Service Sample*

A custom state can consume an external SOAP Web service.

The Web service provider in this sample is the United States Consumer Product Safety Commission. The WSDL file (`CPSCUpcSvc.wsdl`) is embedded with the bundle. Alternately, you can retrieve the WSDL file in real time using the `<wsdlUrls>` configuration. The JAX-WS Maven plug-in reads the WSDL file and generates all the required artifacts for Web service development, deployment, and invocation.

#### *pom.xml*

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">

<build>
[…]
 <!-- Create an OSGi Bundle Manifest -->
 <plugins>
  <plugin>
  […]
    <configuration>
    […]
      <Private-Package>
       com.sap.example
       ,org.tempuri
      </Private-Package>
    […]
    </configuration>
  </plugin>
 </plugins>
</build>

<profiles>
```

```
<!-- Required Javax Persistence dependency -->
 <profile>
    <activation>
     <jdk>[1.5, 1.7)</jdk>
    </activation>
  <dependencies>
    <dependency>
      <groupId>org.eclipse.persistence</groupId>
      <artifactId>javax.persistence</artifactId>
      <version>2.0.4.v201112161009</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  <repositories>
    <repository>
      <id>EclipseLink</id>
     <url>http://download.eclipse.org/rt/eclipselink/maven.repo</url>

    </repository>
  </repositories>
 </profile>

<!-- Required SOAP Web Service JAX-WS only on JDK 6 -->
 <profile>
  <id>jdk6</id>
  <activation>
    <jdk>1.6</jdk>
  </activation>
  <build>
   <plugins>
    <plugin>
      <groupId>org.jvnet.jax-ws-commons</groupId>
      <artifactId>jaxws-maven-plugin</artifactId>
      <version>2.1</version>
      <executions>
        <execution>
         <id>import-wsdld</id>
         <phase>generate-sources</phase>
         <goals>
          <goal>wsimport</goal>
         </goals>
        <configuration>
          <wsdlFiles>
           <wsdlFile>CPSCUpcSvc.wsdl</wsdlFile>
          </wsdlFiles>
         <extension>true</extension>
         <xdebug>true</xdebug>
        </configuration>
       </execution>
      </executions>
     </plugin>
    </plugins>
   </build>
  </profile>
```

```
<!-- Required SOAP Web Service JAX-WS only on JDK 7 -->
  <profile>
   <id>jdk7</id>
   <activation>
     <jdk>1.7</jdk>
   </activation>
   <build>
     <plugins>
      <plugin>
       <groupId>org.jvnet.jax-ws-commons</groupId>
       <artifactId>jaxws-maven-plugin</artifactId>
       <version>2.2</version>
       <executions>
        <execution>
         <id>import-wsdld</id>
         <phase>generate-sources</phase>
         <goals>
          <goal>wsimport</goal>
         </goals>
         <configuration>
           <wsdlFiles>
             <wsdlFile>CPSCUpcSvc.wsdl</wsdlFile>
           </wsdlFiles>
           <extension>true</extension>
           <xdebug>true</xdebug>
         </configuration>
        </execution>
       </executions>
     </plugin>
    </plugins>
   </build>
  </profile>
 </profiles>
</project>
```

*SampleSOAPState.java*

```
package com.sap.example;
[…]
import org.tempuri.CPSCUpcSvc;
import org.tempuri.GetRecallByWordResponse.GetRecallByWordResult;

public class SampleSOAPState extends SmappStatePlugin {

  @Override
  protected SmappState processStateLogic(SmappStateProcessingContext
context,
                                 SmappStateProcessingAction action)

    throws MwizProcessingException, DBException {
    CPSCUpcSvc recallService = null;
    String serviceUrl = "http://www.cpsc.gov/cgibin/CPSCUpcWS/
CPSCUpcSvc.asmx?WSDL";
    try {
      recallService = new CPSCUpcSvc(new URL(serviceUrl),
      new QName("http://tempuri.org/", "CPSCUpcSvc"));
    } catch (MalformedURLException mfue) {
        […]
    }
    if (null == recallService) {
       return continueFail();
    }
    String keyword = "booster";
    GetRecallByWordResult recallServiceResult =

recallService.getCPSCUpcSvcSoap12().getRecallByWord(keyword, "", "");


    if (null == recallServiceResult) {
      return continueDyn(1);
    }
    return continueOk();
  }
}
```

### *Consuming RESTful Services*

Custom states that consume external RESTful Web services can use the Restlet API.

These Restlet bundles are included with SMS Builder, so you need not copy them when you install bundles. For information about using the Restlet API, see *www.restlet.org*.

#### *org.restlet-2.10.13.jar*

```
<groupId>org.restlet.jee</groupId>
<artifactId>org.restlet</artifactId>
<version>2.0.13</version>
```

#### *org.restlet.ext.servlet-2.0.13.jar*

```
<groupId>org.restlet.jee</groupId>
<artifactId>org.restlet.ext.servlet</artifactId>
<version>2.0.13</version>
```

#### *org.restlet.ext.spring-2.0.13.jar*

```
<groupId>org.restlet.jee</groupId>
<artifactId>org.restlet.ext.spring</artifactId>
<version>2.0.13</version>
```

#### *org.restlet.ext.json-2.0.13.jar*

```
<groupId>org.restlet.jee</groupId>
<artifactId>org.restlet.ext.json</artifactId>
<version>2.0.13</version>
```

### *Developing Quick-Start Templates*

You can develop custom states to enhance application capabilities, such as integration with existing enterprise systems or cloud services. To demonstrate functionality, include sample applications in state bundles, which appear in the Web UI as quick-start templates that you can import.

### Prerequisites

1. Develop custom states and deploy them to SMS Builder.
2. Develop one or more sample applications that use the custom states.
3. Export applications to an XML file. An XML file can contain multiple applications.

**Note:** Each XML file creates one quick-start template. Each custom-state bundle can contain multiple quick-start templates.

### Task

Quick-start templates provide commonly used applications that you can customize to meet specific customer needs. You can also create a quick-start template that includes a group of applications to meet a specific functionality, for example, a mobile wallet.

1. Copy application XML files to `META-INF/sample/template`.
2. For each XML file, create a dynamic template plug-in.
3. Redeploy the custom-states bundle to SMS Builder.

The Quick-Start Templates component appears on the Web UI Dashboard.

### See also

- *Creating Applications from Templates* on page 28

#### *Creating Dynamic Template Plug-Ins*

To create a dynamic template that you can plug in to a custom-state bundle, configure the State SDK `SmappTemplateProvider` class as a Spring bean.

This example configures the `SmappTemplateProvider` class for the `GetDate.xml` file, which contains an application that demonstrates how to use the custom state Get Date. To configure the `SmappTemplateProvider` class, edit both the `beans-context.xml` and the `services-context.xml` files.

#### *beans-context.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans

   http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">


<!--
***********************************
```

```
       Beans Configuration
***********************************
-->
 <bean id="SampleState" class="com.sap.example.SampleState">
   <property name="country" value="${sample.country}"/>
 </bean>

 <!-- Template -->
 <bean id="SampleApplication" class=

"com.sybase365.mobiliser.brand.template.SmappTemplateProvider">
   <property name="name" value="Sample Get Date Application" />
   <property name="description" value="Type: Training.
                   A sample application to demonstrate the Get Date
state." />
   <property name="resource" value="classpath:META-INF/template/
GetDate.xml" />
 </bean>
</beans>
```

*services-context.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns=http://www.springframework.org/schema/beans
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:osgi="http://www.eclipse.org/gemini/blueprint/schema/
blueprint"
   xsi:schemaLocation="http://www.springframework.org/schema/beans

   http://www.springframework.org/schema/beans/spring-beans-3.1.xsd

   http://www.eclipse.org/gemini/blueprint/schema/blueprint
   http://www.eclipse.org/gemini/blueprint/schema/blueprint/gemini-
blueprint-1.0.xsd">

   <!--
   ***********************************
     Register state as OSGi Service
   ***********************************
   -->
   <osgi:service id="SampleStateService"
                 ref="SampleState"
                 interface=

"com.sybase365.mobiliser.brand.plugins.api.smapp.StatePlugin"/>

   <!--
   Template Service
   -->
   <osgi:service id="SampleApplicationService"
                 ref="SampleApplication"
                 interface=

"com.sybase365.mobiliser.brand.plugins.api.smapp.SmappTemplate"
                 context-class-loader="service-provider"/>
</beans>
```

## Custom State Samples

Custom state samples illustrate how to implement a service state, a standalone state, and a menu state.

### Sample GetMyWeather State

The `GetMyWeather` sample illustrates a typical custom-state implementation. This type of state is called a service state, because its function is to call a specific Web service (in this case a weather service), and store the results for the application to use. This type of state is commonly integrated with enterprise systems.

```
public class GetMyWeather extends SmappStatePlugin {
  private static final Logger LOG =
                       LoggerFactory.getLogger(GetMyWeather.class);


  // Define Input attributes

  private static final TextBoxAttribute inPostCode =
        new TextBoxAttribute("POSTCODE", "Zip or Postal Code", false);


  // Define Output attributes

  private static final OutputAttribute outWeather =
         new OutputAttribute("WEATHER", "Your Weather Synopsis");

  private static Attribute[] stateAttr;

  static {
    stateAttr = new Attribute[] {inPostCode, outWeather};
  }
  private static long STATE_ID = 600000L;

  @Override
  public long getStateId() {
    return STATE_ID;
  }

  @Override
  public String getStateName() {
    return "Example - Get My Weather";
  }

  @Override
  public String getRevisionString() {
    return "1.0.0";
  }

  @Override
  public String getStateNotes() {
    StringBuilder sb = new StringBuilder();
    sb.append("A sample state. When executed, it checks for a ");
    sb.append("Postal/ZIP Code, and returns the weather report for ");
    sb.append(" that area.\n\n Use the following follow up states:\n ");
    sb.append("- OK: Weather report for the area was found\n ");
    sb.append("- FAIL: Unexpected error\n ");
    sb.append("- Dyn -1: Area code entered was not valid\n ");
    sb.append("- Dyn -2: No weather report for the area\n ");
    return sb.toString();
  }

  @Override
  public boolean supportsFailTransition() {
    return true;
  }

  @Override
  protected Attribute[] getStateAttributes() {
    return stateAttr.clone();
  }

  @Override
  protected SmappState processStateLogic(
                           SmappStateProcessingContext context,
                           SmappStateProcessingAction action)
           throws MwizProcessingException, DBException {

    WeatherResult result = null;

    try {
      // Call the weather Web service
      // Details are Web service specific and therefore
      // are encapsulated in the callWeatherService method

      result = callWeatherService();

      if (result == null)
        return continueFail();

      if (result.status == -1)
        return continueDyn(-1);

      if (result.status == -2)
        return continueDyn(-2);


      // Output attribute

      outWeather.setValue(result.text);
      return continueOk();
    }
    catch (DBException dbex) {
    // Database exception can occur while saving session attributes
      LOG.error("error");
      return continueFail();
    }
  }
}
```

**See also**

- *Custom State Variables* on page 46
- *Sample Custom State* on page 81
- *Extending the SmappStatePlugin Class* on page 36

## Sample Custom State

A simple custom state, named SampleState, formats the current date.

You can modify the date format in the `properties-context.xml` file. The formatted date is stored in an output variable.

### *SampleState.java*

```
package com.sap.example;
import java.text.Format;
import java.text.SimpleDateFormat;
import java.util.Date;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import com.sybase365.mobiliser.brand.dao.DBException;
import com.sybase365.mobiliser.brand.jpa.SmappState;
import
com.sybase365.mobiliser.brand.plugins.api.smapp.SmappStateProcessingActi
on;
import
com.sybase365.mobiliser.brand.plugins.api.smapp.SmappStateProcessingCont
ext;
import
com.sybase365.mobiliser.brand.plugins.smapp.controls.Attribute;
import
com.sybase365.mobiliser.brand.plugins.smapp.controls.OutputAttribute;
import
com.sybase365.mobiliser.brand.plugins.smapp.state.SmappStatePlugin;
import
com.sybase365.mobiliser.brand.processing.exceptions.MwizProcessingExcept
ion;

public class SampleState  extends SmappStatePlugin {
  private static final Logger LOG =
                    LoggerFactory.getLogger(SampleState.class);
  protected static final OutputAttribute outDate =
                       new OutputAttribute("DATE", "Current Date");
  private static Attribute[] stateAttr;
  private String country = "";

  public void setCountry(String value) {
    LOG.debug("Country = " + value);
    this.country = value;
  }
  static {
    stateAttr = new Attribute[] {outDate};
  }

  private static long STATE_ID = 600000L;

  @Override
  public String getStateNotes() {
    return "A sample state. When executed, it returns the current \n"

        + " date in the format of the configured country.\n\n"
         + "Use the following follow up states:\n"
         + "- OK: date and time in the output variable.\n"
         + "- FAIL: If an error occurs during processing.\n";
  }

  @Override
  public boolean supportsFailTransition() {
    return true;
  }

  @Override
  protected Attribute[] getStateAttributes() {
    return stateAttr.clone();
  }

  public String getRevisionString() {
    return "1.0.0";
  }

  public long getStateId() {
    return STATE_ID;
  }

  public String getStateName() {
    return "Example - Get Date";
  }

  @Override
  protected SmappState processStateLogic(
                            SmappStateProcessingContext context,
                            SmappStateProcessingAction action)
            throws MwizProcessingException, DBException {

    Format formatter = new SimpleDateFormat("MM dd yyyy");

    if (!country.equalsIgnoreCase("US"))
      formatter = new SimpleDateFormat("dd MM yyyy");

    outDate.setValue(formatter.format(new Date()));
    return continueOk();
  }
}
```

**See also**

- *Sample GetMyWeather State* on page 79
- *Extending the SmappStatePlugin Class* on page 36

## Sample Custom-Menu State

The contents of `SendSampleMenu.java` and `SampleBean.java` illustrate how to create a custom-menu state.

### *SendSampleMenu.java*

Some details from this sample have been omitted, because they are similar to those in nonmenu custom-state implementations.

```
// Package name and imports have been omitted for clarity

public class SendSampleMenu extends AbstractStateMenuImpl {

  // Other omissions include input and output variable declarations,
  // getRevisionString, getStateId, getStateName, and getStateNotes

  @Override
  protected int getMaxMenuItems () {
    return 4;
  }

  // Similar implementation as getStateAttributes

  @Override
  protected Attribute[] getStateAttributeList() {

    // Assume stateAttr has been defined
    return stateAttr.clone();
  }

  @Override
  protected SmappState init(SmappStateProcessingAction action)
           throws DBException {
    try {
      // Get the menu list from the source: database or service
      // Convert it to the SampleBean list
      // See SampleBean class below

      List<SampleBean> sampleList = getSampleMenuList();

      // Store the list in the session variable
      setMenuListToSession(sampleList);
    }
    catch (DBException dbex) {
      return continueFail();
    }
    catch (Exception ex) {
      return continueFail();
    }
    return null;
  }

  @Override
  protected List<KeyValuePair<String, String>> constructMenuList()
           throws DBException {
    List<KeyValuePair<String, String>> menuList =
           new ArrayList<KeyValuePair<String, String>>();

    for (SampleBean sb : getMenuListFromSession(new SampleBean()))
{
      keyValuePair = new KeyValuePair<String, String>();
      keyValuePair.setKey(sb.getId());
      keyValuePair.setValue(sb.getStatus());
      menuList.add(keyValuePair);
    }
    return menuList;
  }

  @Override
  protected SmappState saveSessionVariables(String key, String value)
           throws DBException {
    int selectedKey = Integer.parseInt(key);

  }
```

*SampleBean.java*

```
// Package name and imports have been omitted for clarity

public class SampleBean implements BeanConverterInterface<SampleBean> {

  protected String id;
  protected String status;

  public static SampleBean parse (String id, String status) {
    SampleBean sb = new SampleBean();
    sb.id = id;
    sb.status = status;
  }

  @Override
  public String convert(SampleBean sb) {
    StringBuilder sb = new StringBuilder();
    sb.append(sb.getId());
    sb.append("|");
    sb.append(sb.getStatus());
    return sb.toString();
  }

  @Override
  public SampleBean convert(String value) {
    String[] values = value.split("\\|");
    Return SampleBean.parse(values[0], values[1]);
  }

  public String getId() {
    return id;
  }

  public String getStatus() {
    return status;
  }
}
```

**See also**

- *Extending the AbstractDynamicMenu Class* on page 40

## State SDK Core Components

You can use State SDK core components when developing custom states. Each component is an OSGi bundle. These components are deployed with SMS Builder, so you need not redeploy them with custom-state components.

### *Plug-in APIs*

The Plug-in APIs include APIs for states, state attributes, and data access objects.

Apache Maven:

```
<groupId>com.sybase365.mobiliser.brand.plugins</groupId>
<artifactId> mobiliser-brand-plugin-api</artifactId>
<name>AIMS :: Object :: SMS Builder Plug-in - API</name>
```

File name: `mobiliser-brand-plugin-api-1.3.1.jar`

### *State SDK*

The State SDK contains state implementation base classes, state input and output controls, and helper classes.

Apache Maven:

```
<groupId>com.sybase365.mobiliser.brand.plugins</groupId>
<artifactId> mobiliser-brand-state-sdk</artifactId>
<name>AIMS :: Object :: SMS Builder Plug-in - State SDK</name>
```

File name: `mobiliser-brand-state-sdk-1.3.1.jar`

### *Security*

The Security APIs support encryption functionality that states use.

Apache Maven:

```
<groupId>com com.sybase365.mobiliser.brand.security</groupId>
<artifactId> mobiliser-brand-security</artifactId>
<name>AIMS :: Object :: SMS Builder Security</name>
```

File name: `mobiliser-brand-security-1.3.1.jar`

### *Core Objects*

Apache Maven:

```
<groupId>com.sybase365.mobiliser.brand.core</groupId>
<artifactId> mobiliser-brand-core</artifactId>
<name>AIMS :: Object :: SMS Builder Core Objects</name>
```

File name: `mobiliser-brand-core-1.3.1.jar`

*Persistence APIs and Models*

Apache Maven:

```
<groupId>com.sybase365.mobiliser.brand.database</groupId>
<artifactId> mobiliser-brand-jpa</artifactId>
<name>AIMS :: Object :: SMS Builder Persistence</name>
```

File name: `mobiliser-brand-jpa-1.3.1.jar`

## States Catalog

For each predefined application state, a catalog entry explains its purpose and how to use it. Use predefined states to build interactive and event applications.

Each state definition includes:

- Input variables – constant values, or values copied from a variable in the current user session.
- Output variables – allow states to return values.
- Follow-up state OK – the condition that constitutes success.
- Follow-up state OK – the condition that constitutes failure, and possible reasons for the failure.
- Follow-up state dynamic – dynamic conditions that transition to follow-up states.
- State editor – example of the state configuration.
- Notes – additional information about the state.
- Usage – Application Composer screen shot that contains the state.

### Add Subscriber State

Adds a subscriber and attributes to the selected subscriber list. You can retrieve a subscriber's MSISDN from a session variable, and set as many as 20 attributes.

*Input Variables*

- **Subscriber Set** – select a subscriber set from a list.
- **Subscriber MSISDN** – unique key for retrieving a subscriber's attributes.
- **Attribute 1**, **Attribute 2**, ... **Attribute 20** – subscriber attributes.

*Output Variables*

**SUBSCRIBER_COUNT** – total number of subscribers in the subscriber set, after adding the current one.

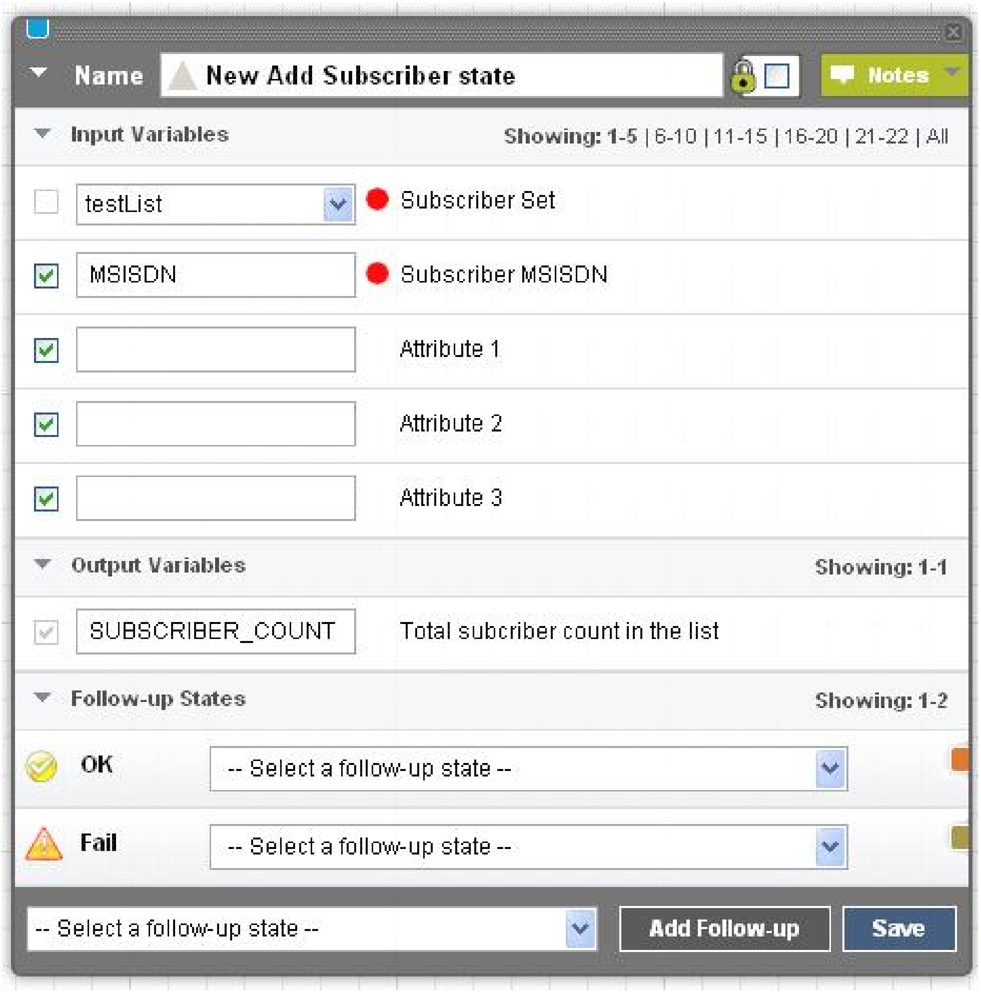
*Follow-up State – OK*
Subscriber was added successfully.

*Follow-up State – Fail*
Error while adding the subscriber, possibly because:

- MSISDN already exists
- Unrecoverable system error, such as a database-connection failure

*Follow-up State – Dynamic*
Not applicable.

*State Editor*
In this example, the New Add Subscriber state adds a subscriber to the testList subscriber set.



New Add Subscriber state

Input Variables — Showing: 1-5 | 6-10 | 11-15 | 16-20 | 21-22 | All

| | | |
|---|---|---|
| ☐ | testList | ● Subscriber Set |
| ✓ | MSISDN | ● Subscriber MSISDN |
| ✓ | | Attribute 1 |
| ✓ | | Attribute 2 |
| ✓ | | Attribute 3 |

Output Variables — Showing: 1-1

| | | |
|---|---|---|
| ✓ | SUBSCRIBER_COUNT | Total subcriber count in the list |

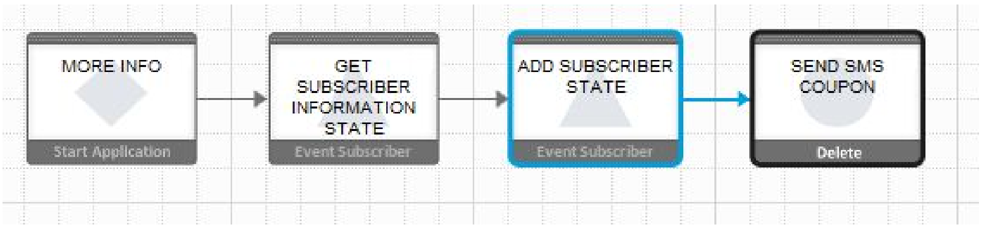
Follow-up States — Showing: 1-2

| | |
|---|---|
| OK | -- Select a follow-up state -- |
| Fail | -- Select a follow-up state -- |

-- Select a follow-up state --

*Usage*

A common use for the Add Subscriber state is to store subscribers who opt to receive messages or coupons. For example, in the More Info application, a message is sent to subscribers, and the message contains a reply keyword for interested subscribers. When a subscriber replies with the keyword, the application retrieves the subscriber's information from the list used in the campaign (Get Subscriber Information state), adds the subscriber to the Opt-In list (Add Subscriber state), and sends a discount coupon to the subscriber.



**See also**

- *Get Subscriber State* on page 101
- *Process Subscriber State* on page 106
- *Update Subscriber State* on page 123

## Application Call State

Calls another application as a subroutine. The called application has access to session variables, and returns control to the current (calling) application.

*Input Variables*

**Application** – select an application in the list. All applications in the list are active in the current workspace.

*Output Variables*

None.

*Follow-up State – OK*

Not applicable.

*Follow-up State – Fail*

Not applicable.

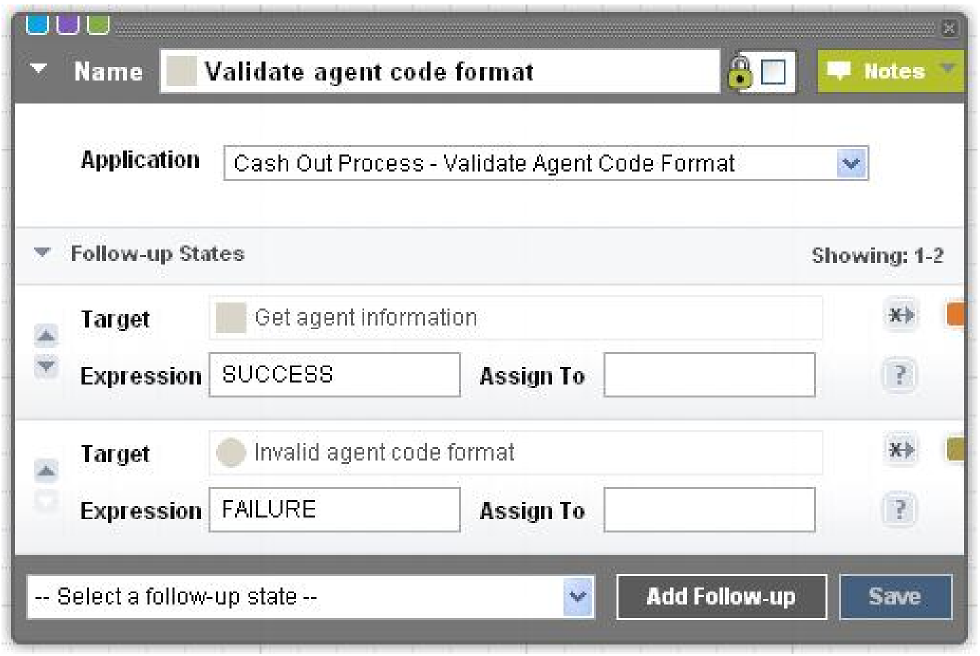
*Follow-up State – Dynamic*

Uses the return value from the Application Call Return state to select which transition to follow.

*State Editor*

The return value from the called application determines the follow-up state. In the example below:

- SUCCESS calls Get Agent Information.
- FAILURE calls Invalid Agent Code Format.



Name: Validate agent code format

Notes

Application: Cash Out Process - Validate Agent Code Format

Follow-up States — Showing: 1-2

| | | | |
|---|---|---|---|
| Target | Get agent information | | |
| Expression | SUCCESS | Assign To | |
| Target | Invalid agent code format | | |
| Expression | FAILURE | Assign To | |

-- Select a follow-up state --

Add Follow-up

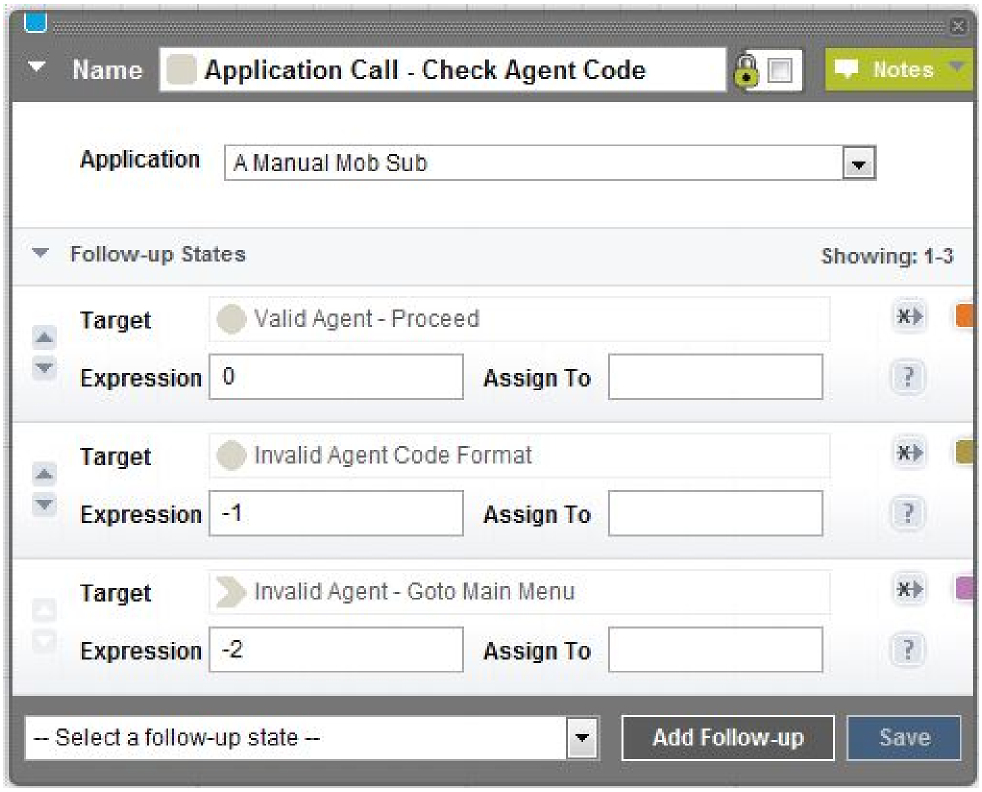
Save

*Notes*

Interactive applications only.

*Usage*

In this example, customers enter a 6-digit code that identifies an agent, and the code is validated. Because this is a common task, you may want to write the validation procedure as a separate application that returns a status code. Using multiple follow-up states, you can link the return value to the appropriate follow-up state.

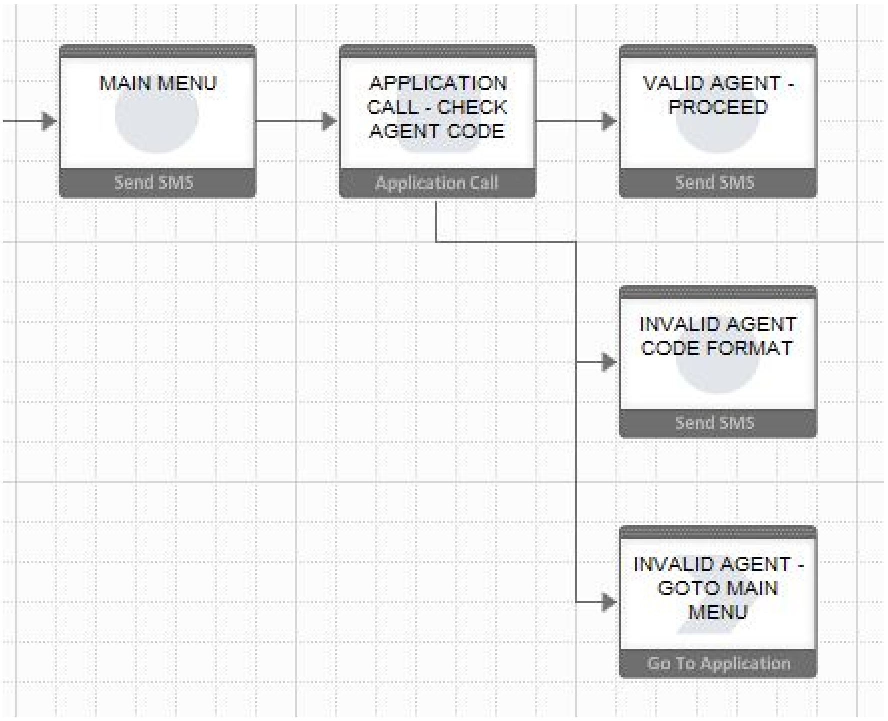


Name: Application Call - Check Agent Code

Application: A Manual Mob Sub

Follow-up States — Showing: 1-3

| | | | |
|---|---|---|---|
| Target | Valid Agent - Proceed | | |
| Expression | 0 | Assign To | |
| Target | Invalid Agent Code Format | | |
| Expression | -1 | Assign To | |
| Target | Invalid Agent - Goto Main Menu | | |
| Expression | -2 | Assign To | |


MAIN MENU
Send SMS
APPLICATION CALL - CHECK AGENT CODE
Application Call
VALID AGENT - PROCEED
Send SMS
INVALID AGENT CODE FORMAT
Send SMS
INVALID AGENT - GOTO MAIN MENU
Go To Application

**See also**

- *Application Call Return State* on page 92
- *Goto Application State* on page 104

## Application Call Return State

The final state of applications that are called by other applications. This state returns a value to the calling application.

*Input Variables*

**Return Value** – value returned to the calling application.

*Output Variables*

None.

*Follow-up State – OK*

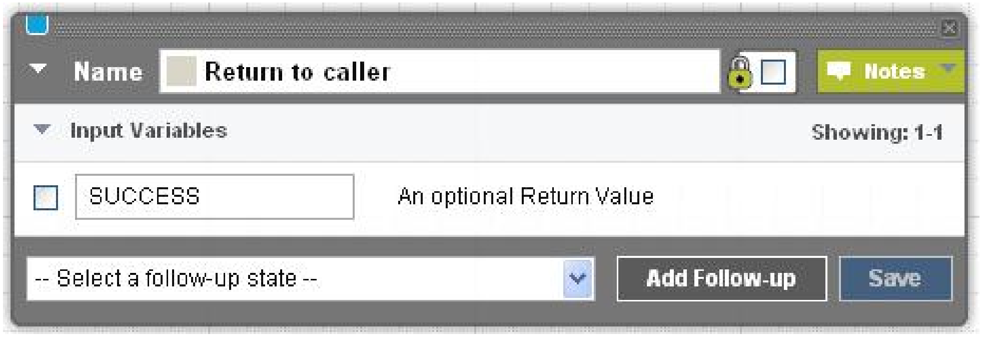
Not applicable.

*Follow-up State – Fail*

Not applicable.

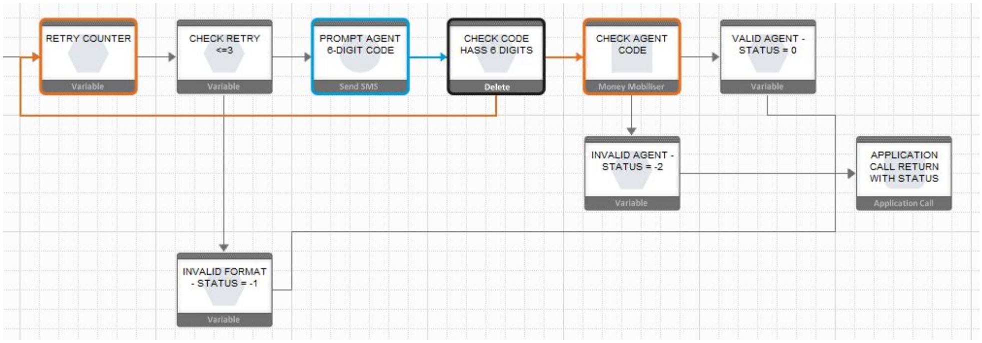
*Follow-up State – Dynamic*
Not applicable.

*State Editor*
This state returns the constant value SUCCESS to the calling application.



*Notes*
Interactive applications only.

*Usage*
This application attempts to validate an agent code, and returns three possible values to the calling application.



## See also

- *Application Call State* on page 89

## Compare Typed Variables State

Compares two variables of the same type: text, integer, double, or date.

*Input Variables*

- **Variable Type** – type to compare: text, integer, double, or date.
- **Text Case Sensitive** – whether text comparison is case-sensitive, yes or no; the default is no.
- **Left Variable** – name of the variable on left side of operator. If the corresponding check box is selected, the application assumes **Left Variable** is the name of a session variable; otherwise, the application assumes **Left Variable** is a constant.
- **Operator** – comparison operator; variable type determines valid operators:

| Variable Type | Valid Operators |
|---|---|
| text | =, !=, =REGEX<br><br>If =REGEX is selected, enter the regular expression as the **Right Variable**. |
| integer, double, or date | =, !=, <=. <, >=, > |

- **Right Variable** – name of variable on right side of operator (or regular expression). If the corresponding check box is selected, the application assumes **Right Variable** is the name of a session variable, otherwise, a constant.

**Note:** If you enter the name of a session variable that does not exist, the state fails.

*Output Variables*
None.

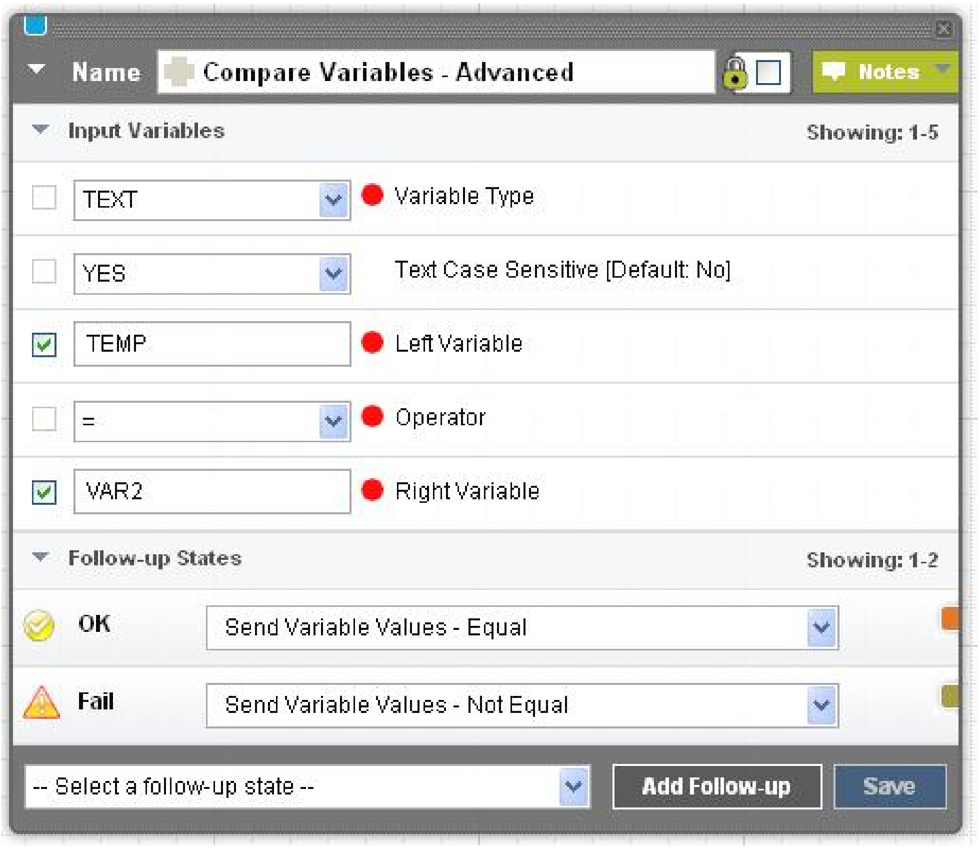
*Follow-up State – OK*
**Left Variable** equals **Right Variable**.

*Follow-up State – Fail*

- The values of **Left Variable** and **Right Variable** are not equal, or
- Either **Left Variable** or **Right Variable** is the name of a session variable that does not exist.

*Follow-up State – Dynamic*
Not applicable.

*State Editor*
In this example, a case-sensitive text comparison is performed for the session variables **TEMP** and **VAR2**. If equal, the follow-up state is Send Variable Values - Equal; if unequal, or either session variable does not exist, the follow-up state is Send Variable Values - Not Equal.



Name: Compare Variables - Advanced

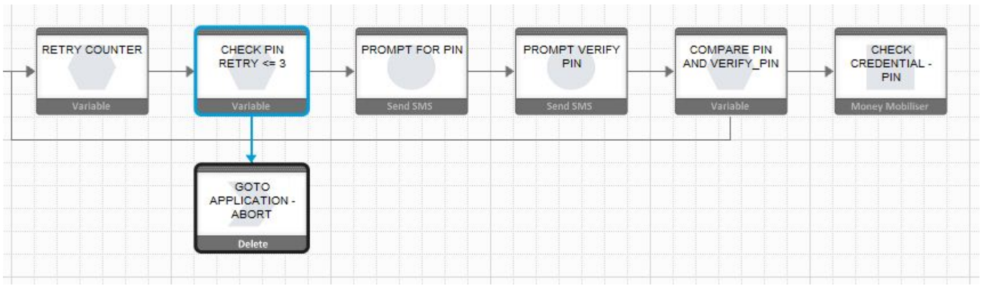
| | | |
|---|---|---|
| ☐ | TEXT | ● Variable Type |
| ☐ | YES | Text Case Sensitive [Default: No] |
| ✓ | TEMP | ● Left Variable |
| ☐ | = | ● Operator |
| ✓ | VAR2 | ● Right Variable |

| | |
|---|---|
| OK | Send Variable Values - Equal |
| Fail | Send Variable Values - Not Equal |

-- Select a follow-up state --

*Usage*

A common use of the Compare Typed Variables state is in an application that prompts for a PIN, and limits the number of incorrect entries.



## See also

- *Compare Variables State* on page 96

## Compare Variables State

Compares the values of two variables, for string equality.

*Input Variables*

For both input variables, if the corresponding check box is selected, the application assumes the value is the name of a session variable; otherwise, the value is treated as a constant.

- **Variable 1** – name of a session variable, or a constant value.
- **Variable 2** – name of a session variable, or a constant value.

*Output Variables*

None.

*Follow-up State – OK*

The values of **Variable 1** and **Variable 2** are equal.

*Follow-up State – Fail*

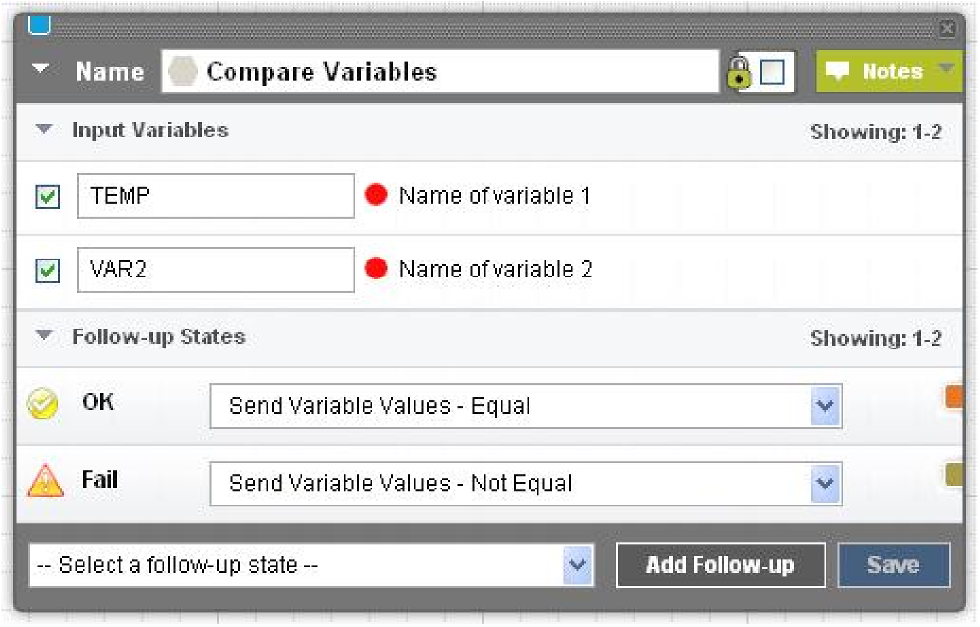
- The values of **Variable 1** and **Variable 2** are not equal, or
- Either **Variable 1** or **Variable 2** is the name of a session variable that does not exist.

*Follow-up State – Dynamic*

Not applicable.

*State Editor*

In this example, if the values of **TEMP** and **VAR2** are equal, the application proceeds to the Send Variable Values - Equal state; if unequal, or either session variable does not exist, proceeds to the Send Variable Values - Not Equal state.



Name: Compare Variables

| | | |
|---|---|---|
| ✓ | TEMP | ● Name of variable 1 |
| ✓ | VAR2 | ● Name of variable 2 |

| | |
|---|---|
| OK | Send Variable Values - Equal |
| Fail | Send Variable Values - Not Equal |

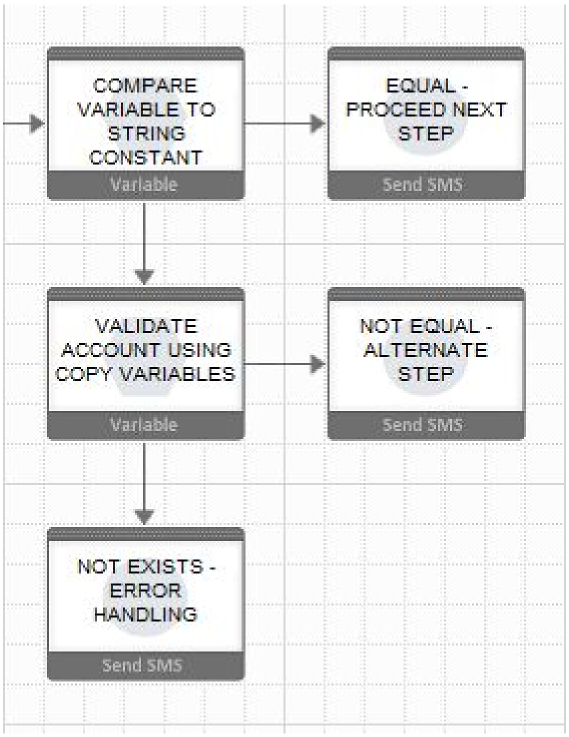
-- Select a follow-up state --

*Notes*

This state compares only for string equality. For comparing other types, use the Compare Typed Variables state.

*Usage*

The sample application below compares the session variable **ACCOUNT** to a constant value. If the two values are unequal, the Validate Account Using Copy Variable state is called to copy the **ACCOUNT** session variable to a dummy session variable. If copying fails, the **ACCOUNT** session variable does not exist.


COMPARE VARIABLE TO STRING CONSTANT
Variable
EQUAL - PROCEED NEXT STEP
Send SMS
VALIDATE ACCOUNT USING COPY VARIABLES
Variable
NOT EQUAL - ALTERNATE STEP
Send SMS
NOT EXISTS - ERROR HANDLING
Send SMS

**See also**

- *Compare Typed Variables State* on page 94

## Copy Variables State

Copies a constant or the value of a source variable to a session variable.

*Input Variables*

**Source** – the source from which to copy. If source is the name of a session variable, select the check box. Otherwise, the application assumes the value of source is a constant.

**Note:** If you specify a session variable that does not exist, the state fails.

*Output Variables*

**Destination** – name of the destination session variable. If the session variable does not already exist, it is created.

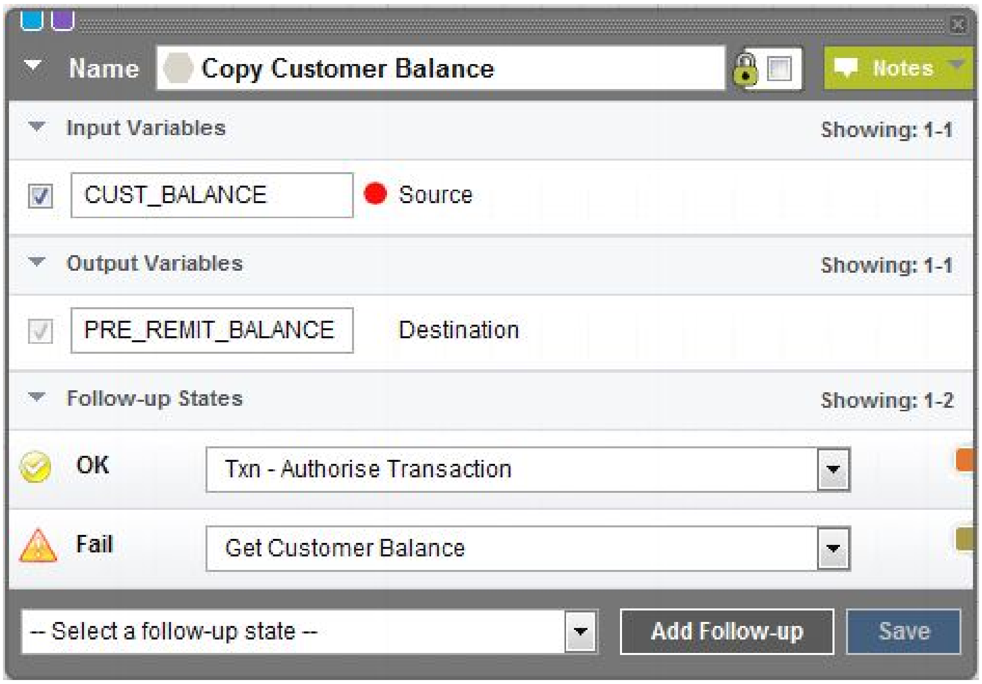
*Follow-up State – OK*

Successfully copied the source to the destination variable.

*Follow-up State – Fail*
Failed to copy the source to the destination variable, usually because the source variable does not exist.

*Follow-up State – Dynamic*
Not applicable.

*State Editor*
This example copies the value of the session variable **CUST_BALANCE** into the session variable **PRE_REMIT_BALANCE**.


Name: Copy Customer Balance

Input Variables — Showing: 1-1
CUST_BALANCE — Source

Output Variables — Showing: 1-1
PRE_REMIT_BALANCE — Destination

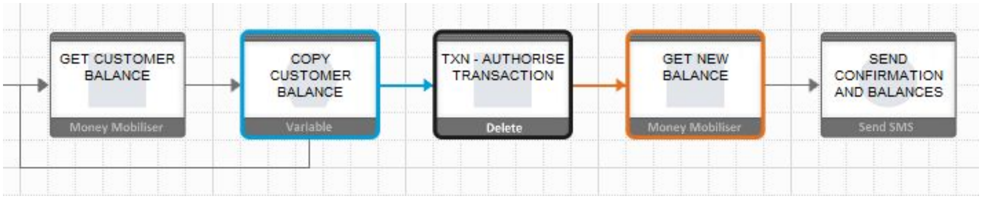
Follow-up States — Showing: 1-2
OK — Txn - Authorise Transaction
Fail — Get Customer Balance

*Notes*
Session variables are also set in these circumstances:

- If you specify a value surrounded by parentheses in the Expression field for a follow-up state, and specify the session variable name in the Assign To field.
- If a state returns values, they are copied to session variables, so they are accessible by follow-up states.

*Usage*
In the sample application below, the customer balance is retrieved twice, before and after calling the transaction. The customer balance is stored in a session variable called Balance. To

prevent overwriting the pretransaction balance with the posttransaction balance, the application copies the pre-transaction balance into another session variable before calling Get New Balance. If Copy Customer Balance fails, Get Customer Balance is called again.



**See also**

- *Set Variable State* on page 120

## Counter State

Creates a variable that is incremented by one each time the state is called.

*Input Variables*

**Variable Name** – name of the session variable to increment. You must select the corresponding check box, or the state fails.

*Output Variables*

None.

*Follow-up State – OK*

Not applicable.

*Follow-up State – Fail*

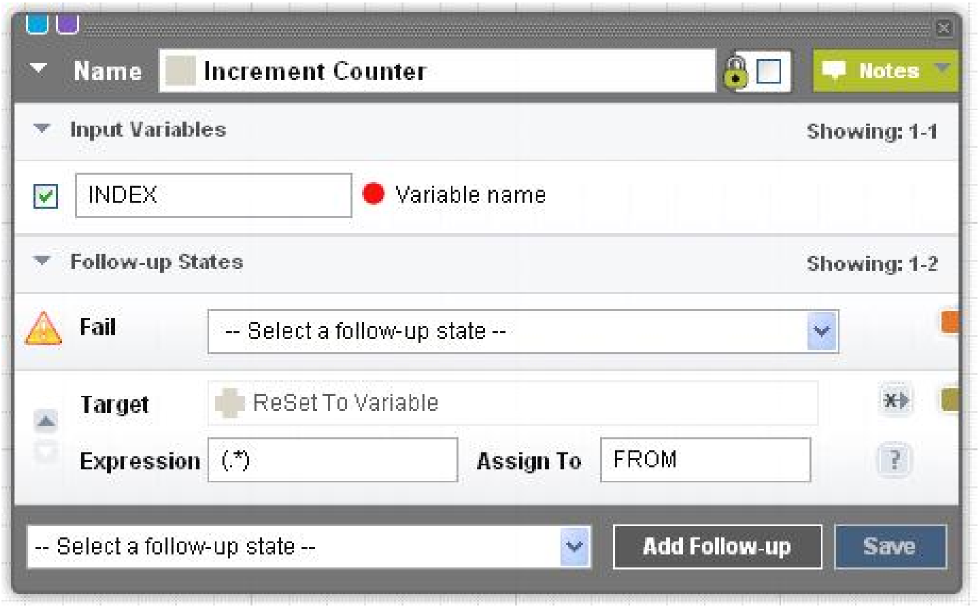
Fails if variable check box is not selected.

*Follow-up State – Dynamic*

Determined by the integer **N**, the updated counter.

*State Editor*

In this example, the Counter state increments the INDEX session variable.



Name: Increment Counter

INDEX — Variable name

Fail: -- Select a follow-up state --

Target: ReSet To Variable

Expression: (.*)   Assign To: FROM

-- Select a follow-up state --

*Notes*

The Counter state increments session variables only.

*Usage*

You can use the Counter state as an index in a loop; commonly used to allow customers a limited number of retry attempts.

## Get Subscriber State

Gets subscriber information from a selected subscriber list. The subscriber's MSISDN is retrieved from the session variable MSISDN. Up to 20 subscriber attributes can be retrieved and assigned to session variables.

*Input Variables*

- *Subscriber Set* – select a subscriber set from a list.
- *Subscriber MSISDN* – unique key for retrieving a subscriber's attributes.

*Output Variables*

*Attribute 1, Attribute 2, ... Attribute 20* – up to 20 subscriber attributes can be assigned to these session variables.

*Follow-up State – OK*

Subscriber attributes successfully retrieved.

*Follow-up State – Fail*

Error while retrieving attributes, possibly because:

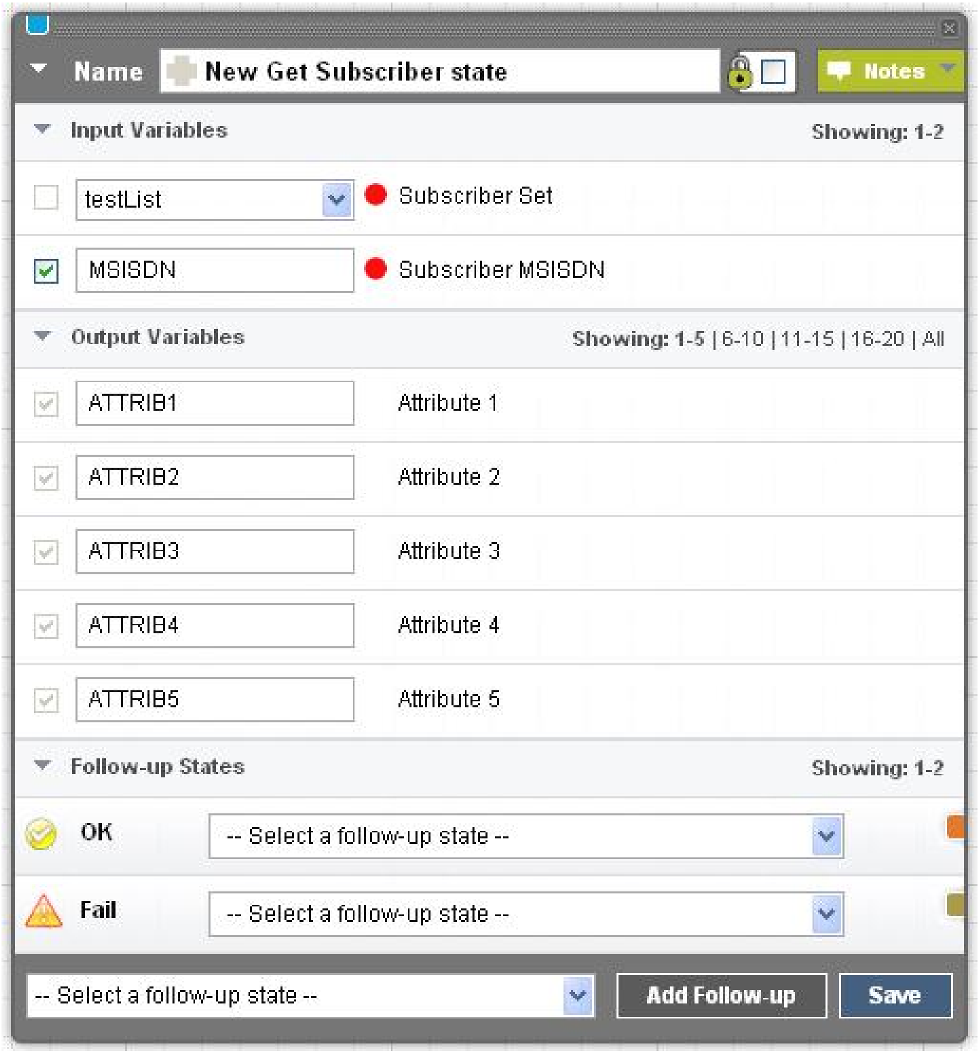
- MSISDN does not exist.
- Unrecoverable system error, such as database-connection failure.

*Follow-up State – Dynamic*

Not applicable.

*State Editor*

This Get Subscriber state retrieves the attributes for the subscriber identified by MSISDN, from the testList subscriber set, and saves attribute values in the output variables.



Name: New Get Subscriber state

Notes

Input Variables — Showing: 1-2

| | | |
|---|---|---|
| ☐ | testList | ● Subscriber Set |
| ✓ | MSISDN | ● Subscriber MSISDN |

Output Variables — Showing: 1-5 | 6-10 | 11-15 | 16-20 | All

| | | |
|---|---|---|
| ✓ | ATTRIB1 | Attribute 1 |
| ✓ | ATTRIB2 | Attribute 2 |
| ✓ | ATTRIB3 | Attribute 3 |
| ✓ | ATTRIB4 | Attribute 4 |
| ✓ | ATTRIB5 | Attribute 5 |

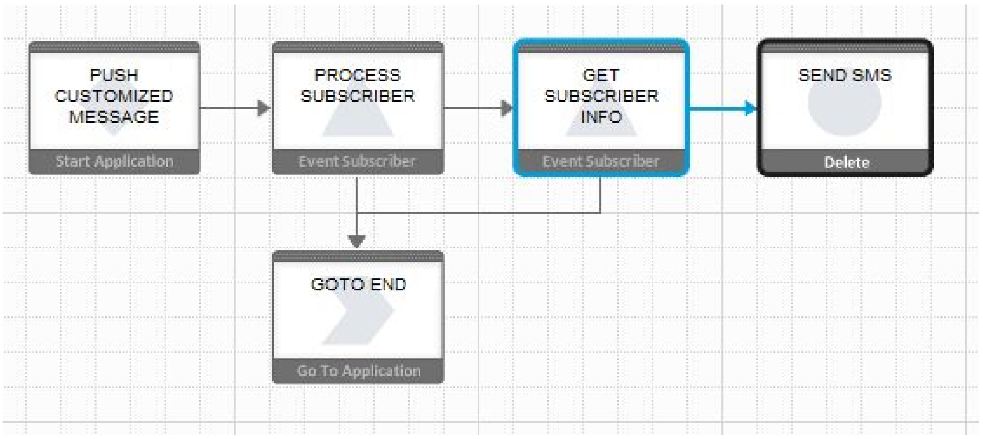
Follow-up States — Showing: 1-2

| | |
|---|---|
| OK | -- Select a follow-up state -- |
| Fail | -- Select a follow-up state -- |

-- Select a follow-up state --

*Usage*

The Get Subscriber state is typically used with the Process Subscriber state.



**See also**

- *Add Subscriber State* on page 86
- *Process Subscriber State* on page 106
- *Update Subscriber State* on page 123

## Goto Application State

The final state of an application that transfers control to another application. Session variables are available to the next application.

*Input Variables*

**Application** – select an application from the list. All applications in the list are active in the current workspace.

*Output Variables*

None.

*Follow-up State – OK*

Not applicable.

*Follow-up State – Fail*

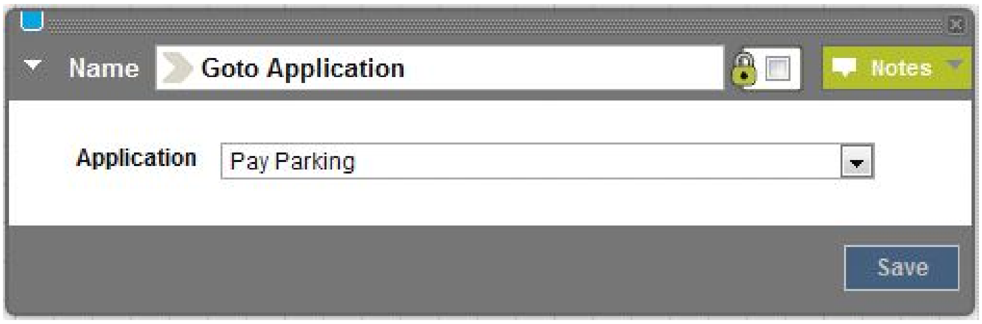
Not applicable.

*Follow-up State – Dynamic*

Not applicable.

*State Editor*

This Goto Application state calls the Pay Parking application.



Goto Application

Application: Pay Parking

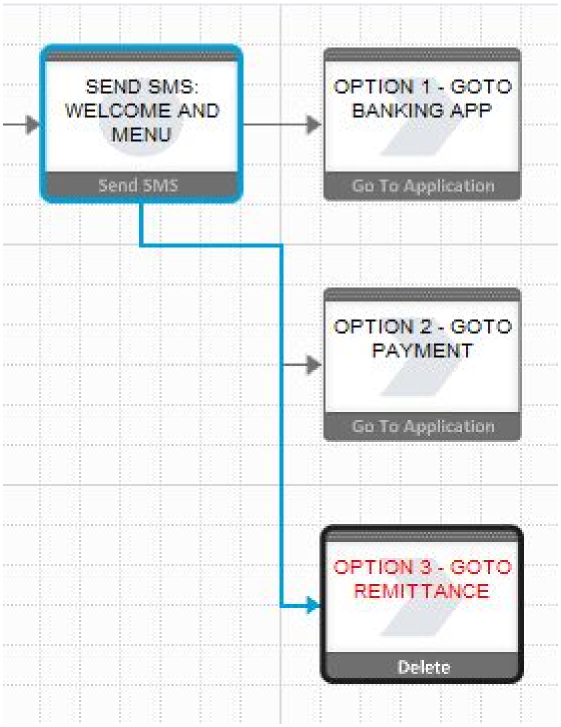
*Notes*

The called (Goto) application must be in the same workspace as the calling application.

In event applications, the Goto Application state cannot follow the Process Subscriber state, because the Goto Application state discontinues the loopback mechanism provided by the engine.

*Usage*

In this example, the Send SMS state sends a menu to customers, whose selections determine the next (Goto) application.


SEND SMS: WELCOME AND MENU
Send SMS
OPTION 1 - GOTO BANKING APP
Go To Application
OPTION 2 - GOTO PAYMENT
Go To Application
OPTION 3 - GOTO REMITTANCE
Delete

**See also**

- *Application Call State* on page 89

## Process Subscriber State

In event applications, the Process Subscriber state typically retrieves a subscriber from a subscriber set, passes the subscriber information to the Send SMS state, then either returns to get the next subscriber, or ends the application.

### *Input Variables*

**Subscriber Set** – select a subscriber set from the list.

### *Output Variables*

None.

### *Follow-up State – OK*

A subscriber is available to process.

### *Follow-up State – Fail*

The event-window processing terminates, because of database connection errors, or other unexpected errors.

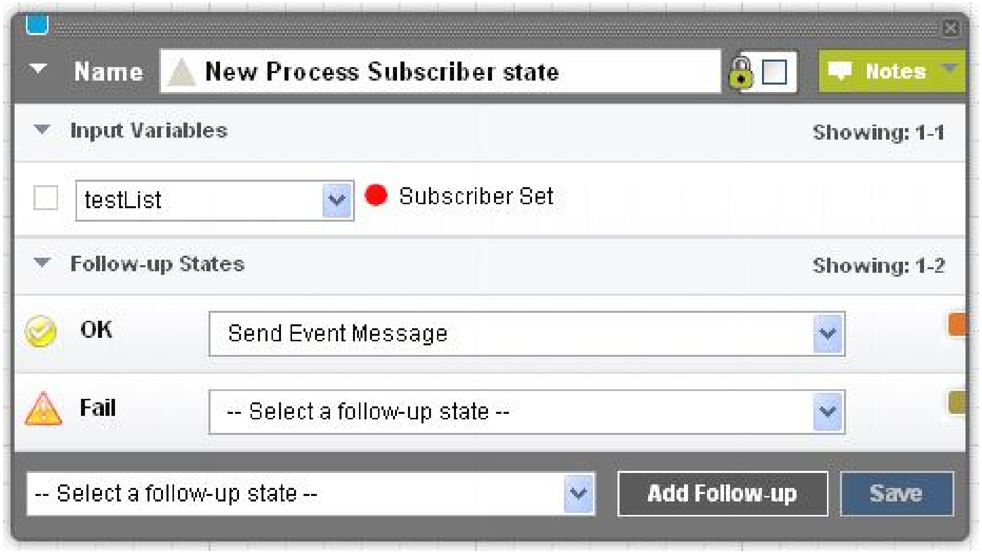
### *Follow-up State – Dynamic*

- END – the end date for the event window has been reached.
- FINISH – processing terminates because the event window ends.
- COMPLETE – no unprocessed subscribers remain in the list.

**Note:** If the state does not handle END, FINISH, and COMPLETE dynamic transitions, the follow-up state is the same as OK.

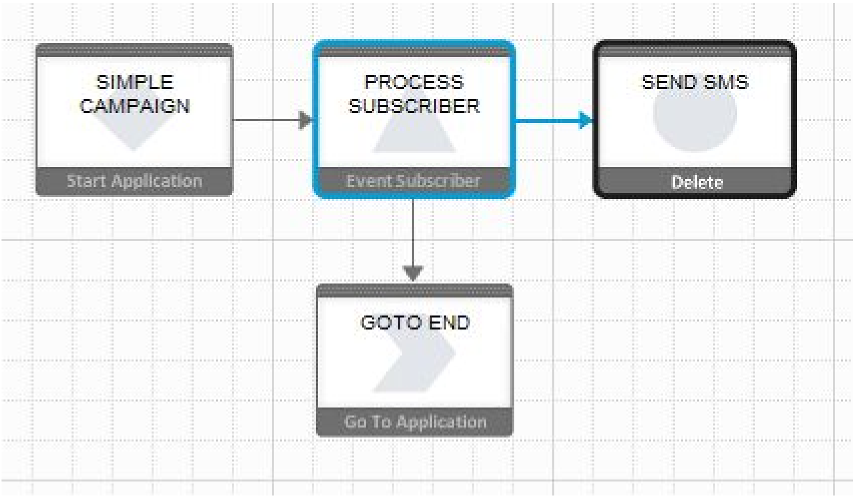
### *State Editor*

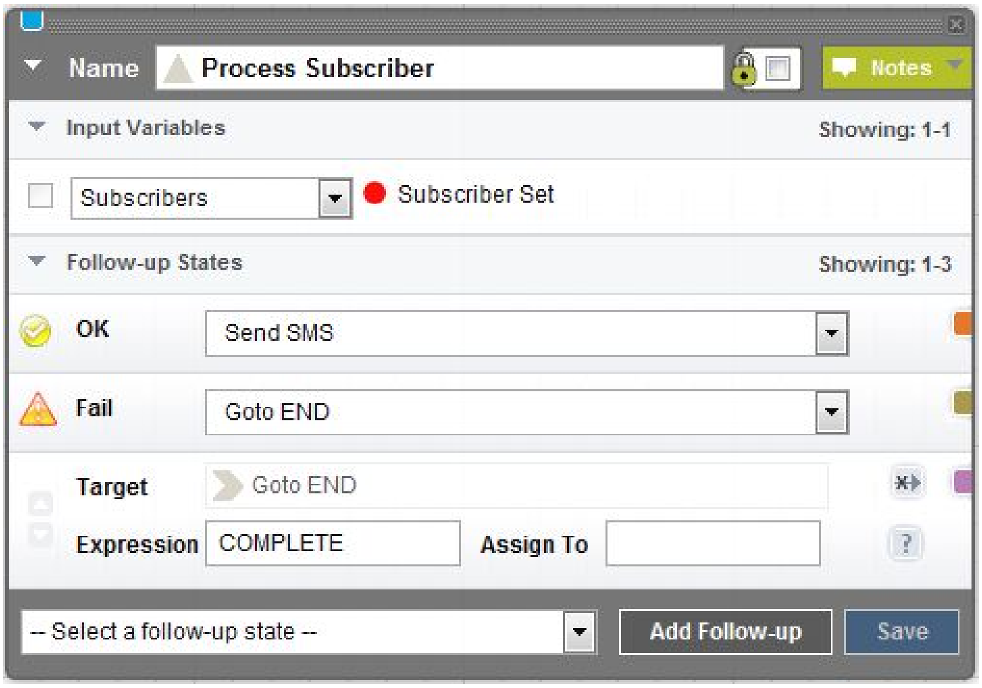
This sample state processes subscribers in the testList subscriber set. When it successfully retrieves a subscriber from the set, it calls Send Event Message.



*Notes*
Event applications only.

*Usage*
This example shows how a simple static-message push campaign gets a subscriber from a set, and sends a message.





Name: Process Subscriber

Input Variables: Subscribers — Subscriber Set

Follow-up States:

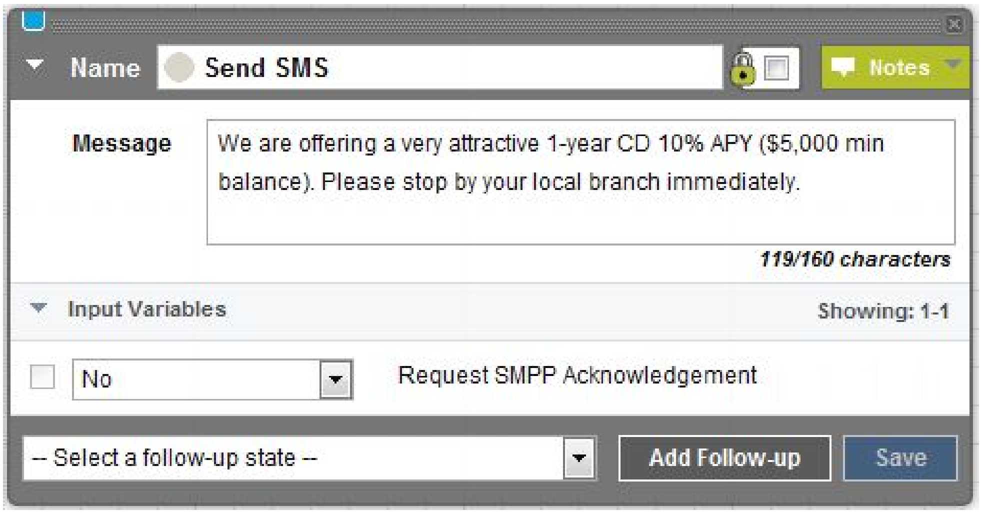
| | |
|---|---|
| OK | Send SMS |
| Fail | Goto END |

Target: Goto END

Expression: COMPLETE

Assign To:

-- Select a follow-up state --



Name: Send SMS

Message: We are offering a very attractive 1-year CD 10% APY ($5,000 min balance). Please stop by your local branch immediately.

*119/160 characters*

Input Variables: No — Request SMPP Acknowledgement

-- Select a follow-up state --

**See also**

- *Add Subscriber State* on page 86
- *Get Subscriber State* on page 101
- *Update Subscriber State* on page 123

## Send SMS State

Sends short message service (SMS) messages to mobile subscribers. If there is at least one follow-up state, the application waits for a subscriber response; otherwise, the application terminates.

*Input Variables*

**Message** – text to send via SMS. If the text is more than 160 characters, the text is divided and sent in multiple messages.

To embed the value of a session variable into the text, enter the name of the variable, surrounded by curly braces. For example, if you enter `{INDEX}`, it is replaced by the value of the session variable **INDEX**. If no such variable exists, `{INDEX}` is sent as a literal.

In event applications, the `Request SMPP Acknowledgement` flag appears in the message, requesting acknowledgement from the short message peer-to-peer (SMPP) gateway.

*Output Variables*

None.

*Follow-up State – OK*

Not applicable.

*Follow-up State – Fail*

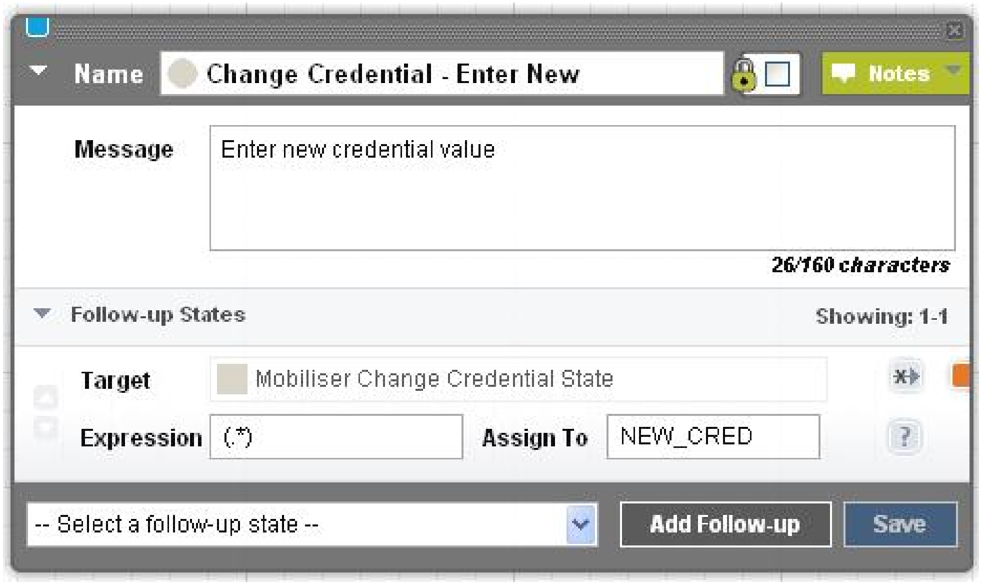
Not applicable.

*Follow-up State – Dynamic*

Continue the application when a response is received. To determine the follow-up state, compare the response to the values of Expression for follow-up states.

*State Editor*

This example specifies one follow-up state, the Mobiliser Change Credential state. The value of Expression matches any response, and assigns the response to the **NEW_CRED** session variable, which can be used later in the task flow.



Name: Change Credential - Enter New

Message: Enter new credential value

26/160 characters

Showing: 1-1

Target: Mobiliser Change Credential State

Expression: (.*)

Assign To: NEW_CRED

-- Select a follow-up state --

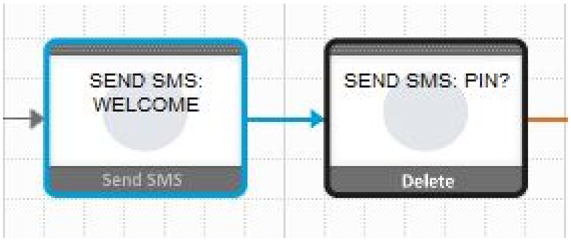
*Notes*

If session variables are embedded in a message, it may be impossible to determine the number of characters in the message prior to runtime.

At runtime, the Send SMS state temporarily suspends the application flow and waits for a response. By default, the wait (also known as session timeout) lasts 7.5 minutes (450 seconds). Once a session times out, responses are ignored. Depending on the setup, subscribers may receive a guidance message or a menu. You can alter the length of the session timeout for each application, on the Application Details screen.

*Usage*

In the scenario illustrated below, the Send SMS state sends a message asking for the subscriber's PIN.



## Send USSD Input State

Sends a prompt for input to subscribers using Unstructured Supplementary Service Data (USSD).

**Note:** By default, USSD states are disabled. USSD is a custom protocol that mobile operators can implement. To develop USSD applications, contact SAP® Professional Services.

*Input Variables*

All input variables are optional.

- **Input Validation String** – value that can validate expected response values.
- **Input Validation Handler URL** – URL to validate expected response values.
- **Mask the Response** – select Yes or No to mask input on the telephone.

*Output Variables*

None.

*Follow-up State – OK*

Not applicable.

*Follow-up State – Fail*

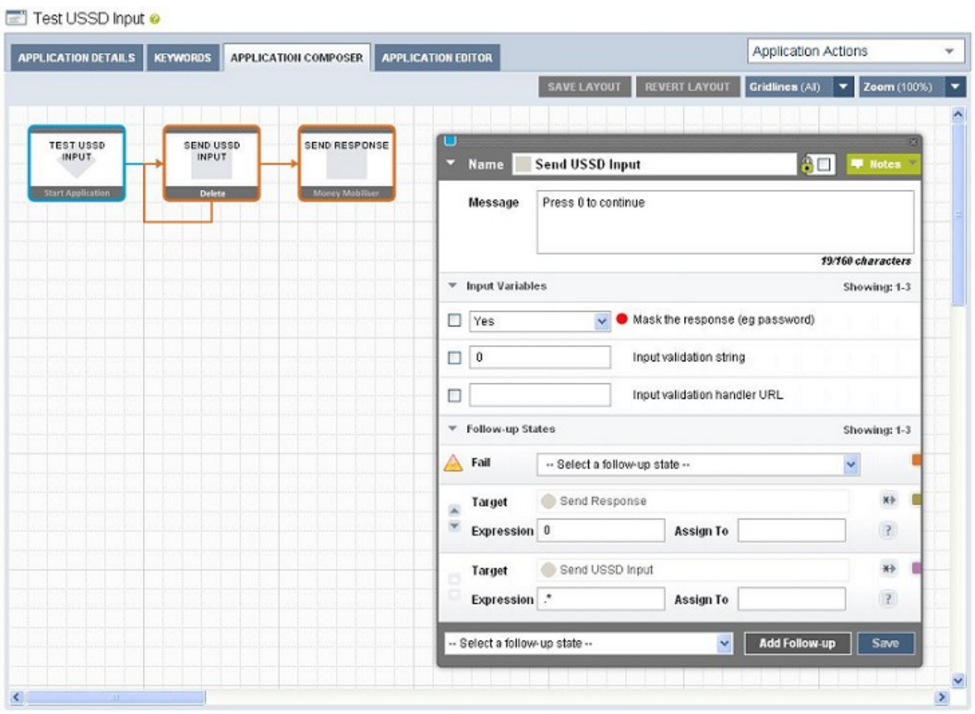
If an internal problem occurs formatting the state text.

*Follow-up State – Dynamic*

Continue the application when a response is received. To determine the follow-up state, compare the response to the values of Expression for follow-up states.

*State Editor*

This example specifies two follow-up states; if the input value is `0`, the Send Response state is called; if the input value is anything else, the Send USSD Input state is called again.



Name: Send USSD Input

Message: Press 0 to continue

19/160 characters

Input Variables — Showing: 1-3

| | |
|---|---|
| Yes | Mask the response (eg password) |
| 0 | Input validation string |
| | Input validation handler URL |

Follow-up States — Showing: 1-3

Fail: -- Select a follow-up state --

Target: Send Response — Expression: 0 — Assign To:

Target: Send USSD Input — Expression: .* — Assign To:

-- Select a follow-up state --

### See also

- *Send USSD Menu State* on page 112
- *Send USSD Text State* on page 118

## Send USSD Menu State

Sends a menu to subscribers via Unstructured Supplementary Service Data (USSD), and expects menu-option responses. This is an abstract state type, which you can extend to develop dynamic menus.

**Note:** By default, USSD states are disabled. USSD is a custom protocol that mobile operators can implement. To develop USSD applications, contact SAP® Professional Services.

### *Input Variables*

**Show Exit Menu Item** – enter:

- `1` for yes; this is the default.
- `0` for no.

*Output Variables*

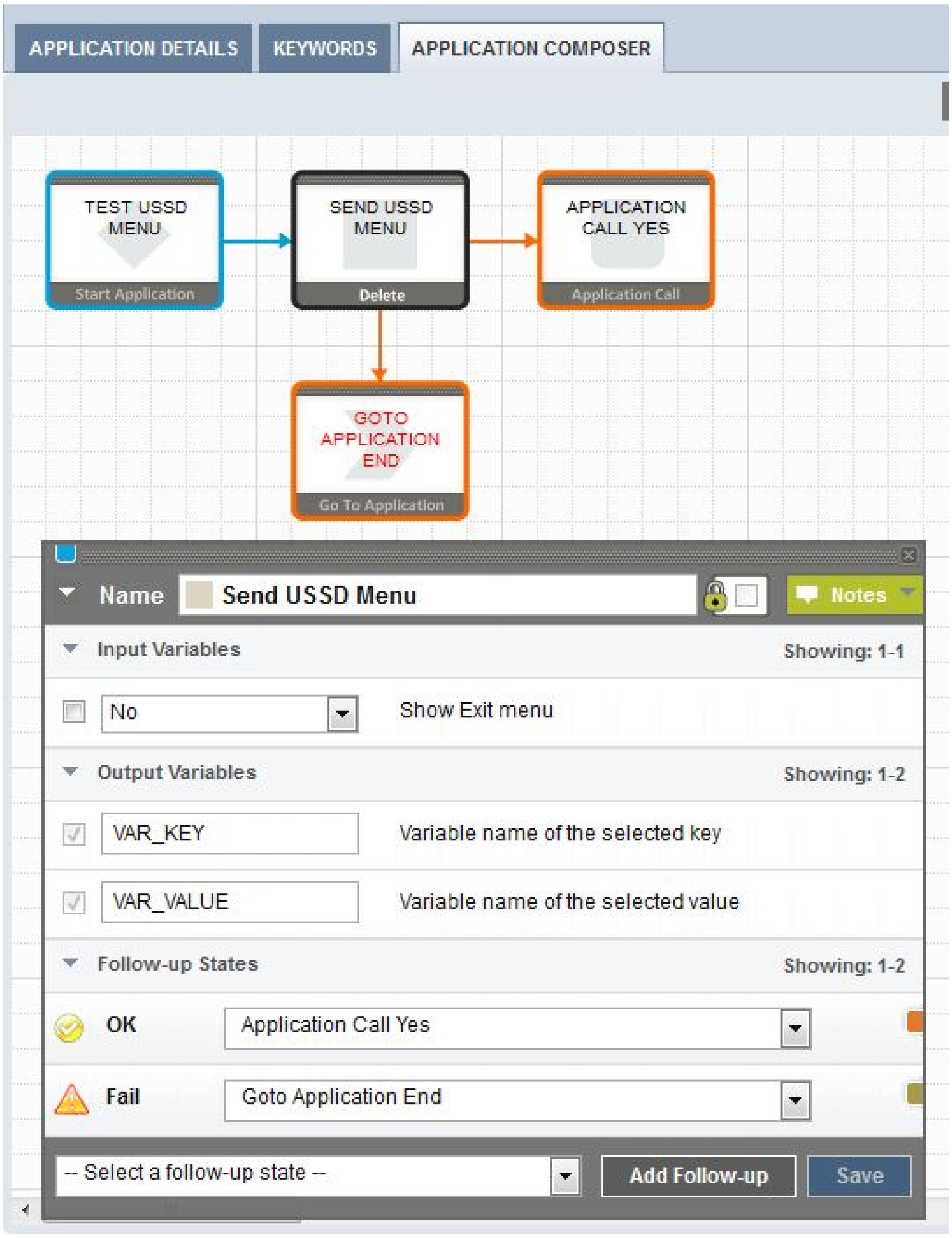
- Variable for selected key – name of the session variable in which to store the selected option key.
- Variable for selected value – name of the session variable in which to store the selected option value.

*Follow-up State – OK*
Typically used when the menu is created successfully, and the user sends a valid response.

*Follow-up State – Fail*
Used only if there is an internal error processing the dynamic menu.

*Follow-up State – Dynamic*
To process dynamic transitions, they must be implemented in the state's code.

*State Editor*
In this example, if users send a valid response, another application is called to process the response. If an error occurs, control is passed to an application that terminates processing. The selected option key is stored in the session variable **VAR_KEY**, and the selected option value is stored in the session variable **VAR_VALUE**.



Name: Send USSD Menu
Notes

Input Variables — Showing: 1-1

| | | |
|---|---|---|
| ☐ | No | Show Exit menu |

Output Variables — Showing: 1-2

| | | |
|---|---|---|
| ✓ | VAR_KEY | Variable name of the selected key |
| ✓ | VAR_VALUE | Variable name of the selected value |

Follow-up States — Showing: 1-2

| | |
|---|---|
| OK | Application Call Yes |
| Fail | Goto Application End |

-- Select a follow-up state --
Add Follow-up
Save

*Notes*

This state enables you to create a dynamic menu, and present the menu to subscribers as a series of options with relevant responses. The menu items are:

- Header text – enter in the Message input field, as the message header.
- Options – provided programmatically in instances of this state type, by a state developer.
- Paging Options – this state type automatically adds Next and Previous options to a menu list if there are more options than fit on a single page.
- End Option – an option that you can add to end or exit the menu.

*Usage*

To implement a dynamic menu, create a subclass that extends this abstract class:

```
com.sybase365.mobiliser.brand.plugins.ussd.impl.AbstractDynamicUssdMenu
```

This abstract superclass creates and structures messages. Subclasses must override and implement abstract methods to provide the required functionality.

```
/* The state attribute list is already set */
protected abstract Attribute[] getStateAttributeList();

/*
* Initialize the dynamic list, possibly based on subscriber information

*/
protected abstract SmappState init(SmappStateProcessingAction action)

   throws MwizProcessingException, DBException, JAXBException,
IOException,
          ServiceException, RequiredParameterMissingException;

/*
* Return the list of options in a format [[key,text],...]
*/
protected abstract List<KeyValuePair<String, String>> getMenuList()

 throws NumberFormatException, DBException,
RequiredParameterMissingException;

/*
* Allow the branching of processes based on selected key.
* If you want to use the configured dynamic follow-up
* transitions, override this method and return continueDyn(key);
* otherwise, override this method and return null to follow the
* OK transition when the user selects an option.
*/
protected abstract SmappState
saveSessionVariables(SmappStateProcessingContext context,
                              String key, String value)
 throws MwizProcessingException, DBException,
RequiredParameterMissingException;
...
```

**See also**

- *Send USSD Input State* on page 111
- *Send USSD Text State* on page 118

## Sample USSD Menu Code

The code for a sample implementation of the Send USSD Menu state produces a menu with four options: Option 1, Option 2, Option 3, and Option 4.

The `SmappStateSendUssdMenu` class implements the sample USSD menu. The fully qualified class name is:

```
com.sybase365.mobiliser.brand.plugins.ussd.impl.SmappStateSendUssdMenu
```

`SmappStateSendUssdMenu` is a subclass of the `AbstractDynamicUssdMenu` abstract class.

```
package com.sybase365.mobiliser.brand.plugins.ussd.impl;

import com.sybase365.mobiliser.brand.dao.DBException;
import com.sybase365.mobiliser.brand.jpa.SmappState;
import
com.sybase365.mobiliser.brand.plugins.api.smapp.SmappStateProcessingActi
on;
import
com.sybase365.mobiliser.brand.plugins.smapp.controls.Attribute;
import com.sybase365.mobiliser.brand.plugins.useful.KeyValuePair;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
* Loads all available languages and puts them into a menu
*
*/
public class SmappStateSendUssdMenu extends AbstractDynamicUssdMenu
{
  protected static final Logger LOG =
      LoggerFactory.getLogger(SmappStateSendUssdMenu.class);

  private static final String[] OPTIONS =
         { "Option 1", "Option 2", "Option 3", "Option 4" };

  private  List<String> listOfOptions = Arrays.asList(OPTIONS);

  private static Attribute[] stateAttr;

  static {
     stateAttr = new Attribute[]{};
  }

  @Override
  protected Attribute[] getStateAttributeList() {
     return stateAttr.clone();
  }

  @Override
  public long getStateId() {
     return 485002;
  }

  @Override
  public String getStateName() {
     return "Send USSD Menu";
   }

  @Override
  public String getStateNotes() {
     return "This state generates a sample USSD Menu.\n" +
            "Use these follow-up states:\n" +
            "- OK: If user selected a menu item.\n" +
            "- FAIL: If an error occurs.";
  }

  @Override
  public boolean supportsOkTransition() {
     return true;
  }

  @Override
  public String getRevisionString() {
     return "$Revision:28128 $";
  }

  @Override
  protected SmappState init(SmappStateProcessingAction action)
     throws DBException {

     if (listOfOptions == null) {
        return continueFail();
     }

     return null;
  }

  @Override
  protected int getMaxMenuItems() {
     return this.listOfOptions.size();
  }

  @Override
  protected List<KeyValuePair<String, String>> constructMenuList()
      throws DBException {

     List<KeyValuePair<String, String>> list =
           new ArrayList<KeyValuePair<String, String>>();

     int optionNumber = 1;

     for (String option : listOfOptions) {
        KeyValuePair<String, String> keyVal = new KeyValuePair<String,
String>();
        keyVal.setKey(Integer.toString(optionNumber));
        keyVal.setValue(option);
        list.add(keyVal);
        optionNumber++;
     }

     return list;
  }

  @Override
  protected SmappState saveSessionVariables(SmappStateProcessingContext
context,
                                        String key, String value)
      throws MwizProcessingException, DBException,
RequiredParameterMissingException {
     return null;
  }
}
```

## Send USSD Text State

Sends a text notification to subscribers via Unstructured Supplementary Service Data (USSD). When subscribers send confirmations, the channel manager passes the messages to the processing engine.

**Note:** By default, USSD states are disabled. USSD is a custom protocol that mobile operators can implement. To develop USSD applications, contact SAP® Professional Services.

*Input Variables*

**USSD Session Handling** – select how USSD sessions are managed by the channel manager.

**Note:** This option is relevant only when the channel manager is configured to manage USSD session information.

The session handling options are:

- None – used when no other option is selected; no specific handling is performed.
- Default – session handling is based on the follow-up state transitions.
- Continue – overrides the default behavior; the channel manager instructs the USSD Gateway with which it is interfacing to continue the USSD session for this user, regardless of whether there are follow-up transitions.
- End – overrides the default behavior; the channel manager instructs the USSD Gateway with which it is interfacing to terminate the USSD session for this user, regardless of whether there are follow-up transitions.

*Output Variables*

None.

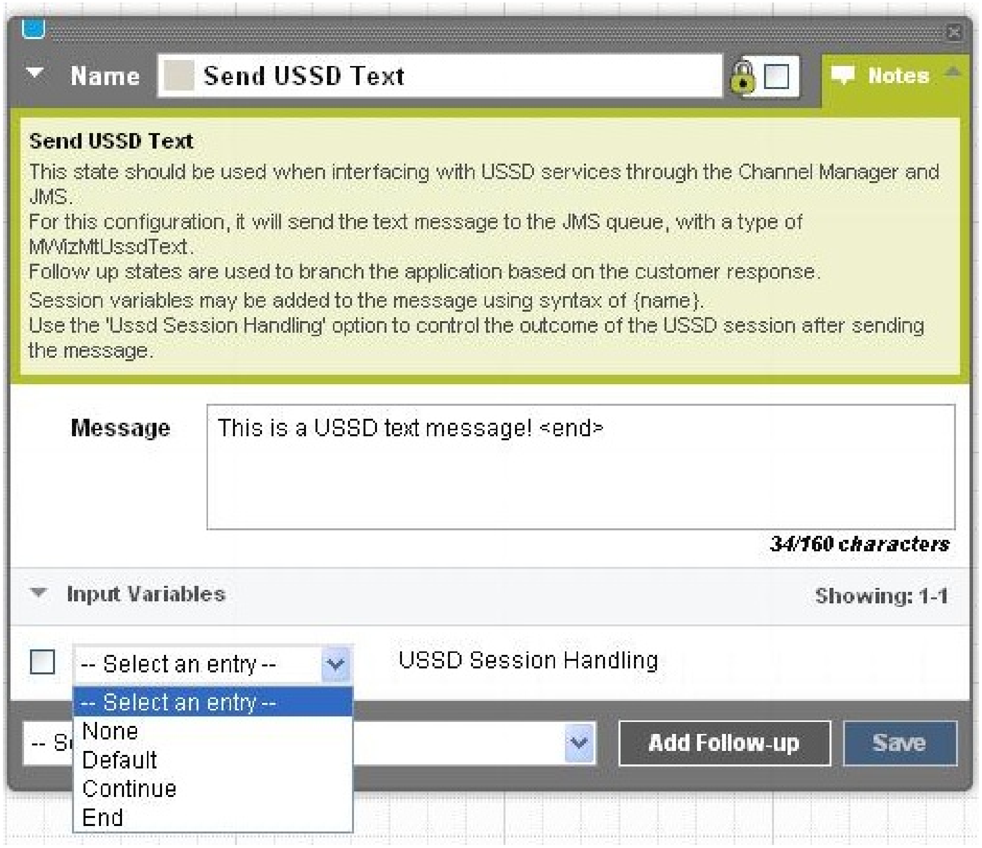
*Follow-up State – OK*

Not applicable.

*Follow-up State – Fail*
Not applicable.

*Follow-up State – Dynamic*
To determine the follow-up state, compare responses to values of Expression for follow-up states.

*State Editor*
In this example, you specify the text to send to subscribers in the Message field. Notes describe the state functionality and how to use it.



**Send USSD Text**
This state should be used when interfacing with USSD services through the Channel Manager and JMS.
For this configuration, it will send the text message to the JMS queue, with a type of MWizMtUssdText.
Follow up states are used to branch the application based on the customer response.
Session variables may be added to the message using syntax of {name}.
Use the 'Ussd Session Handling' option to control the outcome of the USSD session after sending the message.

Message: This is a USSD text message! <end>
34/160 characters

Input Variables — Showing: 1-1
USSD Session Handling: -- Select an entry --

*Notes*
To tell the channel manager to end the USSD session, the state appends `[$[End]$]` to the message text. The channel manager strips off this text before sending the message to the USSD Gateway.

**See also**

- *Send USSD Input State* on page 111
- *Send USSD Menu State* on page 112

## Set Variable State

Sets a session variable with a specified string value. If you specify a numeric value, it is saved as a string.

### *Input Variables*

- **Variable** – name of the session variable to set.
- **Value** – value to save in the session variable. To set **Variable** with the value of another session variable, specify the session variable name as {**sessionVariable**} where **sessionVariable** contains the value to copy.

### *Output Variables*

None.

### *Follow-up State – OK*

The name of the follow-up state after successful processing.

This process always succeeds and moves to the next state.

**Note:** This state performs no error checking. Even if the input variables are empty, it proceeds to the follow-up state. SAP recommends that you use the Copy Variables state to set session variables, because it performs input validations, and uses the Fail follow-up state for error handling and debugging.

### *Follow-up State – Fail*

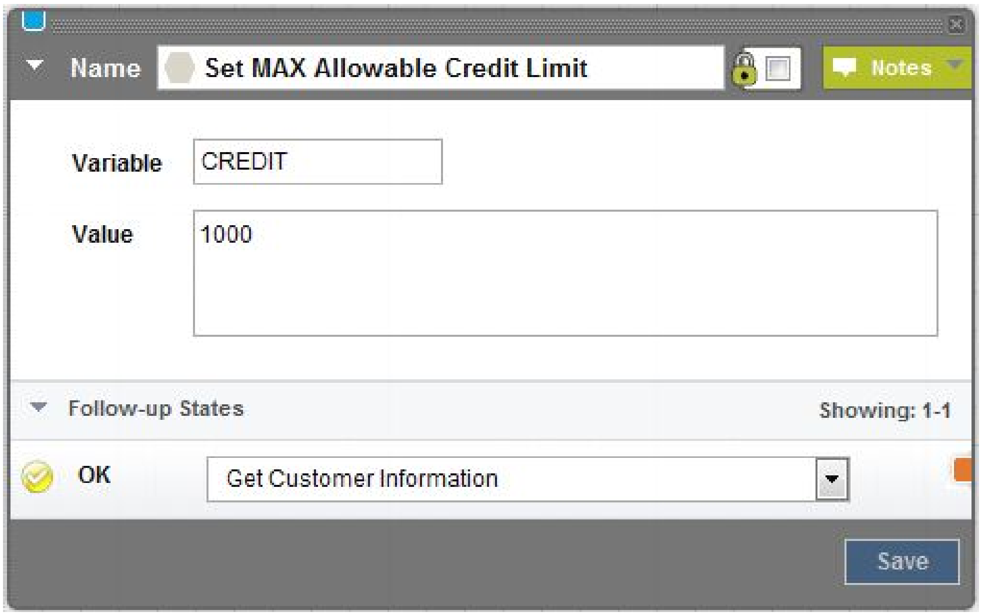
Not applicable.

### *Follow-up State – Dynamic*

Not applicable.

### *State Editor*

This example sets the session variable **CREDIT** to 1000. The variable can be accessed by any state in the application.



Set MAX Allowable Credit Limit

Variable CREDIT

Value 1000

Showing: 1-1

OK Get Customer Information

*Notes*

Session variables are also set in these circumstances:

- If you specify a value surrounded by parentheses in the Expression field for a follow-up state, and specify the session variable name in the Assign To field.
- If a state returns values, they are copied to session variables, so they are accessible by follow-up states.

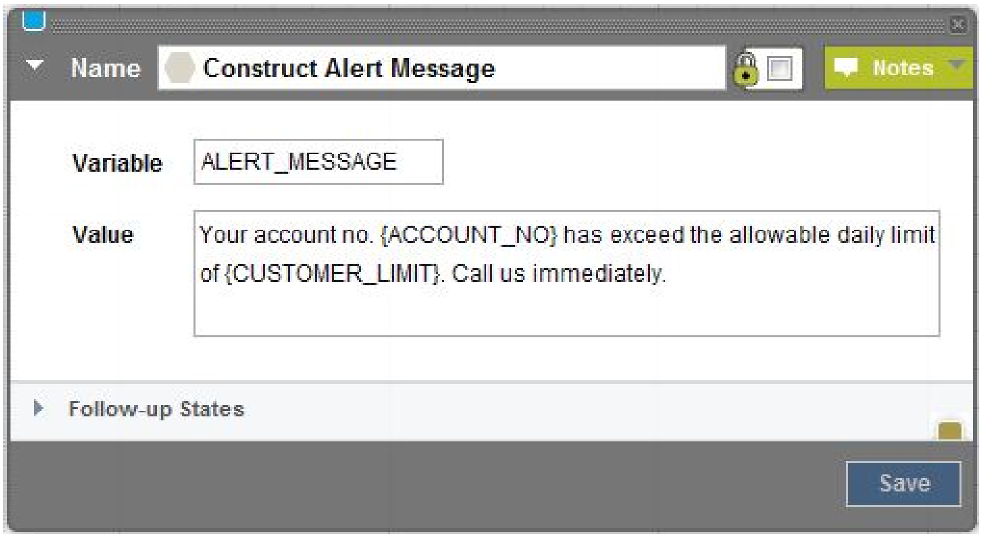
---

**Note:** Setting session variables overwrites any values that are already set for them. For example, if a state returns a value in the session variable X, and the follow-up state also sets variable X, the return value is lost. To avoid this issue, use the Copy Variables state, instead of Set Variable.

---

*Usage*

This example sets the session variable **ALERT_MESSAGE** with a message sent by the Send SMS state.


Name: Construct Alert Message

Notes

Variable: ALERT_MESSAGE

Value: Your account no. {ACCOUNT_NO} has exceed the allowable daily limit of {CUSTOMER_LIMIT}. Call us immediately.

Follow-up States

Save

**See also**

- *Copy Variables State* on page 98

## Start Application State

The Start Application state is the initial state in applications. It is created automatically, and cannot be deleted.

*Input Variables*
None.

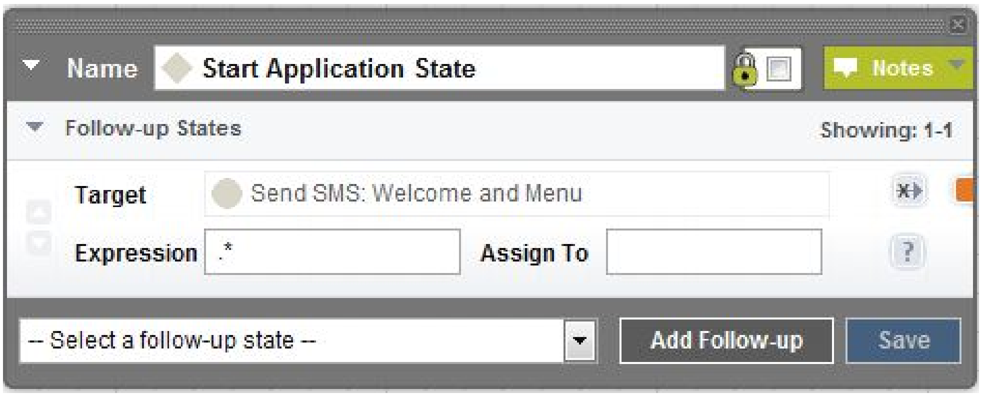
*Output Variables*
None.

*Follow-up State – OK*
Not applicable.

*Follow-up State – Fail*
Not applicable.

*Follow-up State – Dynamic*
Keywords sent by subscribers initiate applications. An application can have multiple keywords. Dynamic transitions enable custom flows, and are based on incoming keywords.

*State Editor*
A Start Application state with a single follow-up state, Send SMS: Welcome and Menu.

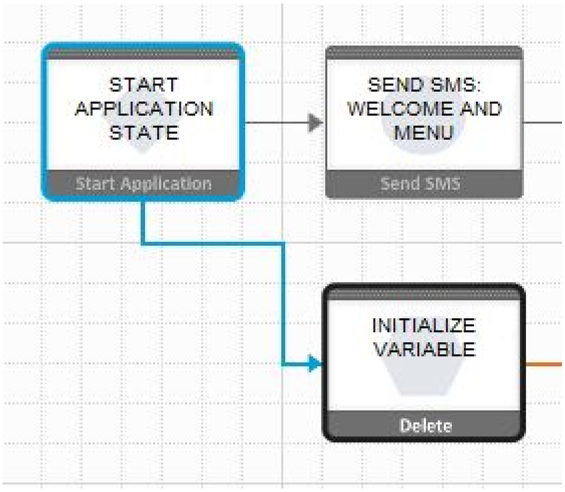


*Notes*

At least one follow-up state is required.

*Usage*

In this example, the Start Application state processes multiple keywords using different task flows.



## Update Subscriber State

Updates subscriber attributes in the selected subscriber set. Gets the subscriber's MSISDN from a session variable, and updates as many as 20 attributes.

*Input Variables*

- **Subscriber Set** – select a subscriber set from a list.
- **Subscriber MSISDN** – unique key for retrieving a subscriber's attributes.

- **Attribute 1**, **Attribute 2**, ... **Attribute 20** – subscriber attributes.

*Output Variables*
None.

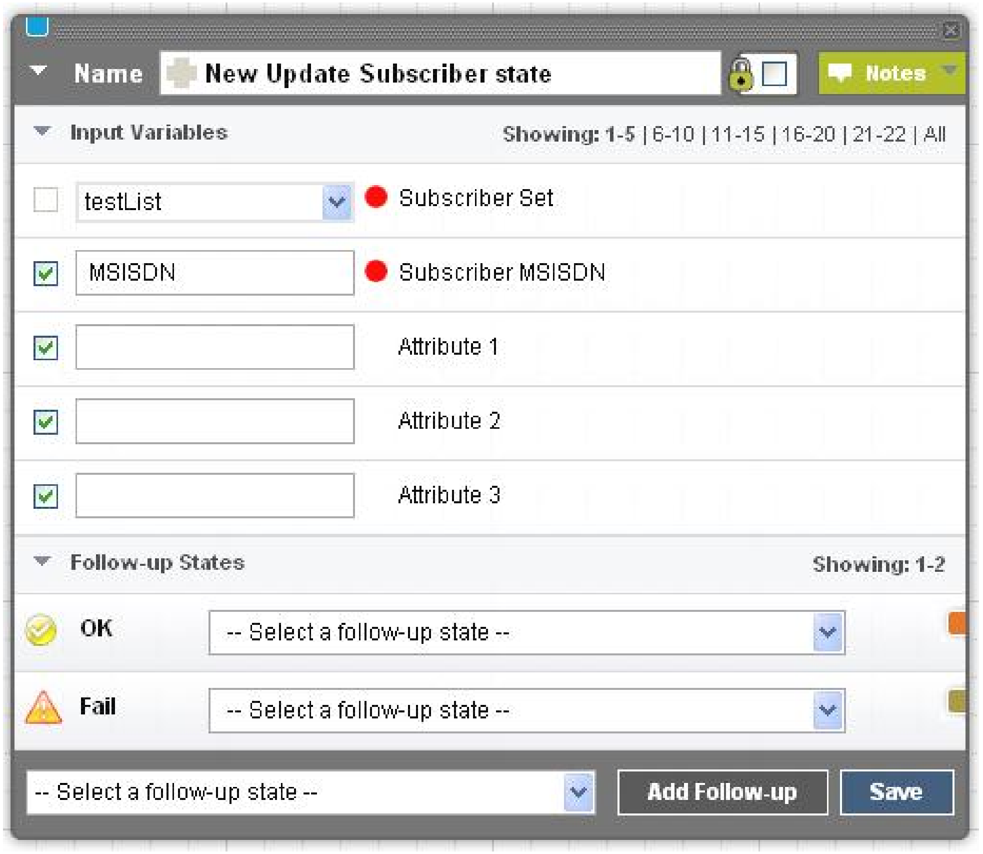
*Follow-up State – OK*
Subscriber updated successfully.

*Follow-up State – Fail*
Error while updating the subscriber, possibly because:

- MSISDN already exists.
- Unrecoverable system error, such as database-connection failure.

*Follow-up State – Dynamic*
Not applicable.

*State Editor*
In this example, the Update Subscriber state updates attributes for subscribers in the testList subscriber set.



Name: New Update Subscriber state

Notes

Input Variables — Showing: 1-5 | 6-10 | 11-15 | 16-20 | 21-22 | All

| | | |
|---|---|---|
| ☐ | testList | ● Subscriber Set |
| ✓ | MSISDN | ● Subscriber MSISDN |
| ✓ | | Attribute 1 |
| ✓ | | Attribute 2 |
| ✓ | | Attribute 3 |

Follow-up States — Showing: 1-2

| | |
|---|---|
| OK | -- Select a follow-up state -- |
| Fail | -- Select a follow-up state -- |

-- Select a follow-up state --

*Notes*
None.

*Usage*
One possible use for the Update Subscriber state is a voting application, in which a voter is added to the Voting Results list, and subsequently, the Update Subscriber state can insert information in other fields.

**See also**
- *Add Subscriber State* on page 86
- *Get Subscriber State* on page 101
- *Process Subscriber State* on page 106

## SMS Application API Reference

Use the SMS application APIs to develop custom application states.

### brand package

*Members*
All public members of the brand package.

- **plugins package –**
- **template package –**

#### plugins package

*Members*
All public members of the plugins package.

- **api package –**
- **base package –**
- **exceptions package –**
- **smapp package –**
- **useful package –**

##### *api package*

*Members*
All public members of the api package.

- **smapp package –**
- **PluginInterface interface –** Plugin interfaces.

###### *smapp package*

*Members*
All public members of the smapp package.

- **controls package –**
- **dao package –**
- **SmappStateEditorContext interface –** The context passes from the state editor to the StatePlugin providing all the necessary information, such as, the current Client or also know as workspace.

- **SmappStateProcessingAction class –** This class is used by the state or StatePlugin to communicate to the "<b>Processing Engine</b>" on the requested follow-up transition.
- **SmappStateProcessingContext class –** The processing engine creates this state processing context or SmappStateProcessingContext before delegating the task to the state.
- **SmappTemplate interface –** Interface for the Application Flow template provider.
- **SmsTextI18n class –**
- **StatePlugin interface –** This is the main interfaces of the state development.

### *controls package*

#### *Members*

All public members of the controls package.

- **IAttribute interface –** Interface for the state attribute.

### *IAttribute interface*

Interface for the state attribute.

#### *Syntax*

```
public interface IAttribute
```

#### *Derived classes*

- *com.sybase365.mobiliser.brand.plugins.smapp.controls.Attribute* on page 175

#### *Remarks*

```
Implemented by
com.sybase365.mobiliser.brand.plugins.smapp.controls.Attribute from
the following bundle.

        groupId=com.sybase365.mobiliser.brand.plugins
        artifactId=mobiliser-brand-state-sdk
```

2012, Sybase Inc.

#### *getDescription() method*

Detailed description of the attribute.

##### Syntax

```
String getDescription ()
```

##### Returns

the description of the attribute

#### Usage

Used in the UI State Editor.

the description of the attribute

### *getId() method*
Attribute Id.

#### Syntax
`String getId ()`

#### Returns
attribute id

#### Usage

Used as a default session variable name.

attribute id

### *getText() method*
The text that was entered in the input field on the state editor.

#### Syntax
`String getText ()`

#### Returns
input text

#### Usage

input text

### *setContext(SmappStateProcessingContext) method*
Sets the processing engine context.

#### Syntax
`void setContext ( SmappStateProcessingContext` *context* `)`

#### Parameters

- **context –** state processing context

## Usage

SmappStateProcessingContext

is used by the attribute to access or store the attribute value as a session variable in the datasource so that it is durable across session.

WARNING

: please refrain from setting or resetting the

SmappStateProcessingContext

. This method is reserved to be used by the processing engine only.

*setDescription(String) method*
Detailed description of the attribute.

## Syntax
`void setDescription ( String` *description* `)`

## Parameters

- **description –** the description of the attribute

## Usage

Used in the UI State Editor.

*setId(String) method*
Attribute Id.

## Syntax
`void setId ( String` *id* `)`

## Parameters

- **id –** attribute id

## Usage

Used as a default session variable name.

*dao package*

*Members*
All public members of the dao package.

- **StateDaoImpl class –**
- **SubscriberDaoImpl class –**

### *StateDaoImpl class*

*Syntax*

`public class StateDaoImpl`

### *StateDaoImpl(final SmappStateDao, final MwizMessageContext) constructor*

**Syntax**

`StateDaoImpl ( final SmappStateDao` *inDao* `, final MwizMessageContext` *msgContext* `)`

### *bulkSaveSessionAttributes(Map< String, String >) method*

Bulk saving of SessionAttributes for the current state in the current session.

**Syntax**

`void bulkSaveSessionAttributes ( Map< String, String >` *attributesMap* `)` throws `DBException`

**Parameters**

- **attributesMap –** Key-Value pair. The key should be obtained using the com.sybase365.mobiliser.brand.plugins.api.smapp.controls.IAttribute#getText() method, and the value from the com.sybase365.mobiliser.brand.plugins.smapp.controls.IAttribute#getHoldValue(). For example,attributesMap.put(outAttrib.getText(), outAttrib.getHoldValue());

**Exceptions**

- **DBException –** Exception while accessing or saving the session variable from database

**Usage**

This method is designed to be used in state implementations that produces multiple

com.sybase365.mobiliser.brand.plugins.api.smapp.controls.IAttribute

s, and it minimizes the database roundtrip.

### *deleteSessionAttribute(String) method*

**Syntax**

`void deleteSessionAttribute ( String` *key* `)` throws `DBException`

*getLanguage(Long) method*
Get language.

**Syntax**
`Language getLanguage ( Long` *languageId* `)` throws `DBException`

**Parameters**

- **languageId –**

**Exceptions**

- **DBException –**

**Usage**

No longer supported.

*getOrCreateCustomerForMsisdn(String, Client) method*

**Syntax**
`Customer getOrCreateCustomerForMsisdn ( String` *msisdn* `,  Client` *c* `)`
throws `DBException`

*getSessionAttributeForKey(String) method*
Get the value of session attribute (SessionAttribute) in the current state based on the specified key from the current session.

**Syntax**
`SessionAttribute getSessionAttributeForKey ( String` *attribKey* `)`
throws `DBException`

**Parameters**

- **attribKey –** session attribute key

**Returns**
the session attribute value

**Exceptions**

- **DBException –** Exception while accessing the session variable from database

#### Usage

the session attribute value

### *getSessionAttributes() method*
Get the session attributes for the current state in the current session.

#### Syntax
`List< SessionAttribute > getSessionAttributes ()` throws `DBException`

#### Returns
list of session attributes for the current session

#### Exceptions

- **DBException –** Exception while accessing the session variable from database

#### Usage

list of session attributes for the current session

### *getSessionAttributesMap() method*
Get session attributes for the current state in the current session.

#### Syntax
`HashMap< String, String > getSessionAttributesMap ()` throws `DBException, CryptoException`

#### Returns
HashMap of session attributes

#### Exceptions

- **DBException –** Exception while accessing the session variable from database
- **CryptoException –** Encryption exception while decrypting the session attribute

#### Usage

HashMap of session attributes

*saveSessionAttribute(String, String) method*
Save the input parameters to the session attribute (SessionAttribute) of the current session.

**Syntax**

`void saveSessionAttribute ( String` *attribKey* `, String` *attribValue* `)` throws `DBException, CryptoException`

**Parameters**

- **attribKey –** session attribute key
- **attribValue –** session attribute value

**Exceptions**

- **DBException –** Exception while saving the session variable from database
- **CryptoException –** Exception during encryption.

**Usage**

For encryption support, see

saveSessionAttribute(String, String, boolean)

.

*saveSessionAttribute(String, String, boolean) method*
Save the input parameters to the session attribute (SessionAttribute) to the current session.

**Syntax**

`void saveSessionAttribute ( String` *attribKey* `, String` *attribValue* `, boolean` *encrypt* `)` throws `DBException, CryptoException`

**Parameters**

- **attribKey –** session attribute key
- **attribValue –** session attribute value
- **encrypt –** True/False whether encryption is needed or not, respectively.

**Exceptions**

- **DBException –** Exception while saving the session variable from database
- **CryptoException –** Exception during encryption.

#### Usage

Encrypt the value prior to saving, if needed (encrypt = true).

Also see

saveSessionAttribute(String, String)

method, if encryption is not needed.

### *saveSessionAttributes(Map< String, String >) method*

Save the session attributes in the Map of the current state in the current session to database.

#### Syntax

`void saveSessionAttributes ( Map< String, String >` *attrs* `)` throws `DBException, CryptoException`

#### Parameters

- **attrs –** session attributes in Map

#### Exceptions

- **DBException –** Exception while saving the session variable from database
- **CryptoException –** Exception during encryption.

### *saveSmappTransitionLogEntry(SmappTransitionLog) method*

Insert into Transition log or database table: M_SMAPP_TRANSITION_LOG.

#### Syntax

`void saveSmappTransitionLogEntry ( SmappTransitionLog` *transitionLog* `)` throws `DBException`

#### Parameters

- **transitionLog –**

#### Exceptions

- **DBException –**

### *SubscriberDaoImpl class*

#### *Syntax*

```
public class  SubscriberDaoImpl
```

*SubscriberDaoImpl(final SubscriberMasterDao, final Client) constructor*

**Syntax**
`SubscriberDaoImpl ( final SubscriberMasterDao` *subsDao* `, final Client` *currentClient* `)`

*addSubscriberToListReturnCount(Long, SubscriberMaster) method*

**Syntax**
`long addSubscriberToListReturnCount ( Long` *subscriberListId* `, SubscriberMaster` *newSubscriber* `)` throws `DBException`

*getSubscriber(Long, String) method*

**Syntax**
`SubscriberMaster getSubscriber ( Long` *subscriberListId* `, String` *subscriberMsisdn* `)` throws `DBException`

*getSubscriberLists() method*

**Syntax**
`List< SubscriberMasterList > getSubscriberLists ()` throws `DBException`

*saveSubscriber(SubscriberMaster) method*

**Syntax**
`SubscriberMaster saveSubscriber ( SubscriberMaster` *subscriber* `)` throws `DBException`

*SmappStateEditorContext interface*
The context passes from the state editor to the StatePlugin providing all the necessary information, such as, the current Client or also know as workspace.

*Syntax*
```
public interface  SmappStateEditorContext
```

*getClient() method*
Client is referred as workspace in the UI.

**Syntax**
`Client getClient ()`

**Returns**
The current Client (or workspace) hosting the application that this state is used

**Usage**

Workspaces are completely isolated and therefore each workspace has unique properties (such as: shortcode), and assets (such as: subscribers, applications, events, etc).,]

The current Client (or workspace) hosting the application that this state is used

### *SmappStateProcessingAction class*
This class is used by the state or StatePlugin to communicate to the "<b>Processing Engine</b>" on the requested follow-up transition.

*Syntax*
```
public class  SmappStateProcessingAction
```

*Remarks*
The processing engine treats the requested follow-up transition as a suggestion, and executes it when appropriate.

Currently, the processing engine supports three actions: continue, wait, and terminate.

This SmappStateProcessingAction is passed by the processing engine to the state as an input parameter to the StatePlugin#processState(SmappStateProcessingContext, SmappStateProcessingAction) method, that is custom implemented by each state. At the end of this method, the implementation need to use the provided SmappStateProcessingAction to call one of the provided methods. Otherwise, the processing engine will automatically terminate the application flow.

### *SmappStateProcessingAction(SmappState) constructor*

**Syntax**
`SmappStateProcessingAction ( SmappState` *currentState* `)`

### *continueProcessing(SmappState) method*
Normal processing action, or continue the application flow to the specified follow-up state (i.e., provided by the input parameter).

**Syntax**
`void continueProcessing ( SmappState` *continueState* `)`

**Parameters**

- **continueState –** the follow-up state to continue to

### *getContinueState() method*

**Syntax**
`SmappState getContinueState ()`

### *isContinueProcessing() method*
Current suggested action is to continue processing.

**Syntax**
`boolean isContinueProcessing ()`

**Returns**
True/False for is continuing or not, respectively

**Usage**

True/False for is continuing or not, respectively

### *isTerminateProcessing() method*
Current suggested action is to terminate the processing.

**Syntax**
`boolean isTerminateProcessing ()`

**Returns**
True/False for is terminating or not, respectively

**Usage**

True/False for is terminating or not, respectively

### *isWaitProcessing() method*
Current suggested action is to wait processing.

**Syntax**
`boolean isWaitProcessing ()`

**Returns**
True/False for is waiting or not, respectively

**Usage**

This is basically telling the processing engine not to continue nor

terminate.

True/False for is waiting or not, respectively

### *terminateProcessing() method*

Suggesting to the processing engine to terminate the application flow when it's appropriate.

#### **Syntax**

```
void terminateProcessing ()
```

#### **Usage**

NOTE:

state implementation should use this action wisely. It is recommended to handle failure with

continueProcessing(SmappState)

with the

*continueFail*

follow-up transition.

### *waitForMessage() method*

Processing should be temporarily paused to wait for additional incoming trigger.

#### **Syntax**

```
void waitForMessage ()
```

#### **Usage**

For example, this is used by the Send SMS state after sending the sms message out. The state is waiting for a reply from the consumer that will be used in determining the follow-up state.

### *continueProcessing variable*

#### *Syntax*

```
boolean continueProcessing
```

### *continueState variable*

#### *Syntax*

```
SmappState continueState
```

*terminateProcessing variable*

*Syntax*
`boolean terminateProcessing`

*SmappStateProcessingContext class*
The processing engine creates this state processing context or SmappStateProcessingContext before delegating the task to the state.

*Syntax*
`public class  SmappStateProcessingContext`

*Remarks*
The context contains all the necessary resources and information, such as: session, data access, processing engine context, etc. for processing the state.

*SmappStateProcessingContext(final SmappState, final MwizMessageContext, final boolean, final OutgoingQueue, final Language, final Language, final SmappStateDao, final SubscriberMasterDao, final CacheManagerDao) constructor*

**Syntax**
`SmappStateProcessingContext ( final SmappState` *currentState* `, final MwizMessageContext` *msgContext* `, final boolean` *newSession* `, final OutgoingQueue` *outgoingQueue* `, final Language` *langDefault* `, final Language` *langRequest* `, final SmappStateDao` *dao* `, final SubscriberMasterDao` *subDao* `, final CacheManagerDao` *cacheMgr* `)` throws `CryptoException`

*getLangRequest() method*

**Syntax**
`Language getLangRequest ()`

*getSession() method*
Get the current session.

**Syntax**
`Session getSession ()`

**Returns**
current session

**Usage**

The processing engine creates a

Session

based on the unique MSISDN (or phone number) before starting the target application. The session will be persisted to database when the application reaches a state that wait for a trigger before continuing. For example, the "Send SMS" state sends SMS message out and wait for reply message before continuing. The processing engine will persist the session to the database prior to transition into the wait mode.

The session only lasts for a definite amount of time. By default, it is set to 450 secs. The session will be used by the "session expiration" daemon to terminate expired sessions.

current session

#### *getStateDao() method*

Get the data access object for the state.

**Syntax**

`StateDaoImpl getStateDao ()`

**Returns**

data access object of StateDaoImpl type

**Usage**

data access object of StateDaoImpl type

#### *getSubscriberDao() method*

**Syntax**

`SubscriberDaoImpl getSubscriberDao ()`

#### *isAckMessageRequested() method*

When sending SMS message using SMPP, an acknowledgement from the gateway or SMSC can be requested.

**Syntax**

`boolean isAckMessageRequested ()`

**Returns**

True or false for whether SMPP acknowledgement is requested or not, respectively

### Usage

This flag indicates whether acknowledgement is requested or not, or true or false, respectively.

True or false for whether SMPP acknowledgement is requested or not, respectively

*isCurrentStateEncrypted() method*
Bulk saving of SessionAttributes for the current state in the current session.

### Syntax

```
boolean isCurrentStateEncrypted ()
```

### Parameters

- **attributesMap –** Key-Value pair. The key should be obtained using the OutputAttribute#getText() method, and the value from the OutputAttribute#getHoldValue(). For example,attributesMap.put(outAttrib.getText(), outAttrib.getHoldValue());
- **attribKey –** session attribute key
- **attribKey –** session attribute key
- **attribValue –** session attribute value
- **attribKey –** session attribute key
- **attribValue –** session attribute value
- **encrypt –** True/False whether encryption is needed or not, respectively.
- **attrs –** session attributes in Map
- **languageId –**
- **transitionLog –**

### Returns

list of session attributes for the current sessionthe session attribute valueHashMap of session attributesTrue or false for encrypted or not, respectively.

### Exceptions

- **DBException –** Exception while accessing or saving the session variable from database Get the session attributes for the current state in the current session.
- **DBException –** Exception while accessing the session variable from database Get the value of session attribute (SessionAttribute) in the current state based on the specified key from the current session.
- **DBException –** Exception while accessing the session variable from database Get session attributes for the current state in the current session.
- **DBException –** Exception while accessing the session variable from database

- **CryptoException –** Encryption exception while decrypting the session attribute Save the input parameters to the session attribute (SessionAttribute) of the current session.
- **DBException –** Exception while saving the session variable from database
- **CryptoException –** Exception during encryption. Save the input parameters to the session attribute (SessionAttribute) to the current session. Encrypt the value prior to saving, if needed (encrypt = true).
- **DBException –** Exception while saving the session variable from database
- **CryptoException –** Exception during encryption. Save the session attributes in the Map of the current state in the current session to database.
- **DBException –** Exception while saving the session variable from database
- **CryptoException –** Exception during encryption. Get language. No longer supported.
- **DBException –** Insert into Transition log or database table: M_SMAPP_TRANSITION_LOG
- **DBException –** Indicate whether the current state is encrypted. This flag can be set in the state editor.

## Usage

This method is designed to be used in state implementation that produces multiple { OutputAttribute}s, and it minimize the database roundtrip.

list of session attributes for the current session

the session attribute value

HashMap of session attributes

For encryption support, see

saveSessionAttribute(String, String, boolean)

.

Also see

saveSessionAttribute(String, String)

method, if encryption is not needed.

When set to true, a state to never show the messages it sends out or inputs it receives back in clear text in the message logs. This is a security feature to allow passwords and PINs to be restricted.

True or false for encrypted or not, respectively.

*setAckMessageRequest(boolean) method*

Set the SMPP acknowledgement request flag for the current state in the current session.

**Syntax**

`void setAckMessageRequest ( boolean` *ackMessage* `)`

**Parameters**

- **ackMessage –** True or false for whether SMPP acknowledgement is requested or not, respectively

*setLangRequest(Language) method*

**Syntax**

`void setLangRequest ( Language` *langRequest* `)`

*updateSession() method*

**Syntax**

`void updateSession ()` throws `DBException`

*cacheMgr variable*

*Syntax*

`CacheManagerDao cacheMgr`

*client variable*

*Syntax*

`Client client`

*clientMsisdn variable*

*Syntax*

`ClientMsisdn clientMsisdn`

*currentState variable*

*Syntax*

`SmappState currentState`

*customer variable*

*Syntax*
`Customer customer`

*langDefault variable*

*Syntax*
`Language langDefault`

*matchingPattern variable*

*Syntax*
`Pattern matchingPattern`

*mr variable*

*Syntax*
`MessageReceiver mr`

*msg variable*

*Syntax*
`MwizMessage msg`

*newSession variable*

*Syntax*
`boolean newSession`

*outgoingQueue variable*

*Syntax*
`OutgoingQueue outgoingQueue`

*SmappTemplate interface*
Interface for the Application Flow template provider.

*Syntax*
```
public interface  SmappTemplate
```

### *Derived classes*

- *com.sybase365.mobiliser.brand.template.SmappTemplateProvider* on page 260

### *Remarks*

Application flow(s) are exported into a XML file that can be re-imported back into the processing engine using the web UI.

The XML file (also called template) can also be packaged into an OSGi bundle and deployed as a plugin bundle. The template will appear in Brand Mobiliser "Dashboard - Quick Start Templates" enabling direct import into the processing engine.

Also see com.sybase365.mobiliser.brand.template.SmappTemplateProvider which is an implementation of this interface. The implementation may be used via Spring injection, as listed below:

- Insert the following into the Spring configuration file (beans-context.xml), to instantiate the provider. <bean id="NewSystem" class="com.sybase365.mobiliser.brand.template.SmappTemplateProvider"> <property name="name" value="New Mobile Service System"> <property name="description" value="A system of applications for Mobile system."> <property name="resource" value="classpath:META-INF/template/new_system.xml"> </bean>
- Register the provider with the OSGi service to make it discover-able by the web application <osgi:service id="NewSystemService" ref="NewSystem" interface="com.sybase365.mobiliser.brand.plugins.api.smapp.SmappTemplate" />

### *getDescription() method*

Get detailed description of the system.

#### Syntax

```
String getDescription ()
```

#### Returns

Detailed description.

#### Usage

This is used by the UI.

Detailed description.

*getInputStream() method*
Call by the Brand Web UI to get access to the XML containing the application flows.

**Syntax**
`InputStream getInputStream ()`

**Returns**
Inputstream containing application flows in XML format.

**Usage**

This is used to load the template into the system.

Inputstream containing application flows in XML format.

*getName() method*
Get the name of the system.

**Syntax**
`String getName ()`

**Returns**
System name.

**Usage**

This is used by the UI.

System name.

*getResource() method*
Get the location of the XML file containing the application flow relative to the bundle classpath.

**Syntax**
`String getResource ()`

**Returns**
Location of the XML file containing the application flows.

**Usage**

Location of the XML file containing the application flows.

*getVersion() method*
Get the version of the system.

**Syntax**
`String getVersion ()`

**Returns**
System version.

**Usage**

Information only.

System version.

*setDescription(String) method*
Set the detailed description of the system.

**Syntax**
`void setDescription ( String` *value* `)`

**Parameters**

- **value –**

**Usage**

Call by Spring during injection.

*setName(String) method*
Set the name of the system.

**Syntax**
`void setName ( String` *value* `)`

**Parameters**

- **value –**

**Usage**

Call by Spring during injection.

### *setResource(String) method*

Set the location of the XML file containing the application flow relative to the bundle classpath.

**Syntax**

`void setResource ( String` *value* `)`

**Parameters**

- **value –** Location of the XML file containing the application flows.

**Usage**

For example:

classpath:META-INF/template/money_mobiliser.xml

### *setVersion(String) method*

Set the version of the system.

**Syntax**

`void setVersion ( String` *value* `)`

**Parameters**

- **value –**

**Usage**

Can be call by Spring during injection.

### *SmsTextI18n class*

*Syntax*

```
public class  SmsTextI18n
```

### *SmsTextI18n(String, Language) constructor*

**Syntax**

`SmsTextI18n ( String` *text* `,  Language` *language* `)`

### *getLanguage() method*

**Syntax**

```
Language getLanguage ()
```

### *getText() method*

**Syntax**

```
String getText ()
```

### *language variable*

*Syntax*

```
Language language
```

### *text variable*

*Syntax*

```
String text
```

### *StatePlugin interface*

This is the main interfaces of the state development.

*Syntax*

```
public interface  StatePlugin
```

*Derived classes*

- *com.sybase365.mobiliser.brand.plugins.smapp.state.SmappStatePlugin* on page 230

*Remarks*

States need to implement this interface so that: it can be displayed on the Application Composer and State Editor, and invoked by the "Processing Engine" at application runtime. As such it should be regarded as the 'external' API. Any classes that extend this interface will be referred to as State for the purpose of this API documentations. Some methods are used only by the state editor or the processing engine, others may be used by both, as stated in each methods description.

INTRODUCTION

An *Application* defines the process flow in the form of interconnected states. An application usually has more than one states, and can include more than one types of state. Therefore, states are the building blocks for composing an application. This StatePlugin is the interface that needs to be implemented by the states.

LIFE CYCLE

At runtime, the processing engine will step through the application flow based on the matched transition logic (called follow-up transition) and execute the corresponding state. The processing engine will execute the application and state based on the following life cycle.

1. Application can be invoked by many different external activation mechanisms, such as: incoming SMS message, scheduled event, or web service invocation. The first state in any applications is the "Application Start" state, that is automatically created by default and cannot be removed. So immediately after invoking the application, the first state, the "Application Start" state will be executed.
2. When executing a state, the processing engine will call one of the following processing methods: the processMessage(SmappStateProcessingContext) and the processState(SmappStateProcessingContext, SmappStateProcessingAction), based on the activation mechanism, described in details below (see ACTIVATION).The method(s) need to be implemented by each state to encapsulate the business logic that the state represents.
3. The state can be invoked by two methods: *external activation* or *follow-up transition*. The <u>external</u> occurs when the state processing has been temporarily paused (or hibernated) waiting for an external event, and the state is re-activated by the arrival of the external event. In this case, the `processMessage()` will be called.If the state is invoked by the follow-up transition, in this case from another state, the second method or `processState()` will be called.
4. The first method (i.e., `processMessage()`) returns the follow-up transition object (SmappState) that will become the current state for the subsequent method (i.e., `processState()`).For example, let's say the application flow looks like the following: state1 ->Send SMS state -> state2 ->state3 When the flow reaches the Send SMS state, a message will be sent out and the flow will be paused waiting for a response. The current state remains as the Send SMS state. On arrival of the response message, first the `processMessage()` method will be called. This method will find the follow-up transition that match the incoming message, and return the corresponding state. Let's say it's "state2". Next, the processing engine will make "state2" as the current state and then call the `processState()` method of "state2".
5. The `processState()` method does not return the follow-up transition object (or SmappState), but instead setting the input parameter (SmappStateProcessingAction), by calling the appropriate method, to communicate to the processing engine the intended follow-up transition.
6. Using the above application flow example, the `processState()` method of "state2" specify "state3" as the follow-up transition state. So the processing engine will continue the process by calling the `processState()` method of "state3".
7. This process of stepping through the flow will continue until one of the following conditions:

- reaching a state that set the processing engine to the waitForMessage status, such as after sending SMS out. The process will re-start on arrival of the message, and it will invoke the `processMessage()` method. The process then repeat from step 4 again.
- the state has no follow-up transition, so the processing engine will terminate

STATE ACTIVATION

The state can be activated by two methods: *external* or *follow-up*. In most cases, the state is activated by the follow-up method. The <u>follow-up activation</u> occurs as a result of a follow-up transition from the previous connected state. In addition, the follow-up activation can also occur when the state performs the *loop-back* mechanism from the one of the processing methods: processMessage(SmappStateProcessingContext) or processState(SmappStateProcessingContext, SmappStateProcessingAction) methods, by conditionally returning the current state (or `context.currentState`). NOTE: please use the loop-back mechanism with caution because it could easily resulting in an infinite loop in the state processing. The follow-up activation calls the processState(SmappStateProcessingContext, SmappStateProcessingAction) method, and the state implements the required business logic in this method.

The <u>external activation</u> occurs when the state processing has been temporarily paused (or hibernated) waiting for an external event, and the state is re-activated by the arrival of the external event by invoking the the processMessageLogic(SmappStateProcessingContext) method. The external activation mechanism is used by states that support the Send SMS, such as: the base "Send SMS" state and the states that extend the com.sybase365.mobiliser.brand.plugins.smapp.state.AbstractDynamicMenu. For example, the "Send SMS" state sends an SMS message to the mobile device, and then goes into hibernation. When a reply message arrives, this will re-activate state using the external activation method that calls the processMessage(SmappStateProcessingContext) method. In addition, any state that implements the calls to the SmappStateProcessingAction - `waitForMessage()` in the processing methods will also need to ensure that external activation will happen and it can handle the external activation in the processMessage(SmappStateProcessingContext) method. Otherwise, the application will never continue and it will be terminated when the session timeout.

The processMessage(SmappStateProcessingContext) and processState(SmappStateProcessingContext, SmappStateProcessingAction) methods are referred to as the **processing methods**.

RETURNING FLOW BACK TO PROCESING ENGINE

There are two mechanisms to return the process back to the processing engine:

- Follow-up transition - the state returns the SmappState from the (processMessage(SmappStateProcessingContext) method, that tells the processing engine which follow-up transition to follow and hence the corresponding follow-up state.
- Follow-up action - by setting the `SmappStateProcessingAction` object provided by processing engine as an input parameter to the processing methods. The

`SmappStateProcessingAction` provides methods for setting the needed actions. For example, calling `action.waitForMessage()` tells the processing engine to wait for external activation before proceeding.The processState(SmappStateProcessingContext, SmappStateProcessingAction) does not return SmappState object. So it needs to use the input parameter `SmappStateProcessingAction` to tell the processing engine which follow-up transition to follow.When the follow-up action is used, the processing methods should return null.

### *getInputAttributes() method*

Return the input attributes specified by this state.

#### Syntax

```
List< IAttribute > getInputAttributes ()
```

#### Returns

List of input variables

#### Usage

This method is called by the State Editor to get the input attribute list that is used to render the Input Variables on the state editor.

This method could be called by the state implementation logic to gather all the input variable values.

Used by: state editor and processing engine

List of input variables

### *getOutputAttributes() method*

Return the output attributes specified by this state.

#### Syntax

```
List< IAttribute > getOutputAttributes ()
```

#### Returns

List of output variables

#### Usage

This method is called by the State Editor to get the output attribute list that is used to render the Output Variables on the state editor.

This method could be called by the state implementation logic to bind all the output variable values, and save to the session variable.

List of output variables

*getStateId() method*
The state unique ID.

**Syntax**
`long getStateId ()`

**Returns**
the state unique ID of type long.

**Usage**

Used by: state editor and processing engine

Refer to 'State Developers Guide' for ranges of values allowed.

the state unique ID of type long.

*getStateName() method*
The state name is used for display on the state editor.

**Syntax**
`String getStateName ()`

**Returns**
the state name

**Usage**

Used by: state editor

the state name

*getStateNotes() method*
The detailed description about the state, and documentations on how to use the state including the follow-up transitions.

**Syntax**
`String getStateNotes ()`

**Returns**
detailed description and documentation of the state

### Usage

Using wicket MultiLineLabel automatic filtering, any newlines '

' will be rendered as

and any double new lines '

' will render as a separate paragraph.

NOTE: Use newline mechanism only for layout of text - do not embed HTML into the text strings

Used by: state editor

detailed description and documentation of the state

### *isSelectable() method*

To indicate whether the state can be used as the follow-up state.

### Syntax

`boolean isSelectable ()`

### Returns

True/false to indicate that it can be used as the follow-up state or not, respectively

### Usage

This is used by the state editor to filter out the follow-up drop down list. Currently, it is only used by the Application state, that is the first state automatically added, and cannot be deleted in any application.

As is now, it's not a very useful method. State should always implement it with a return of true.

Used by: state editor

True/false to indicate that it can be used as the follow-up state or not, respectively

### *loadStateAttributes(SmappStateEditorContext) method*

This method is called by the state editor only.

### Syntax

`void loadStateAttributes ( SmappStateEditorContext` *editorContext* `)`

### Parameters

- **editorContext –** The context passes from the state editor to the StatePlugin providing all the necessary information, such as, the current Client or also know as workspace.

### Usage

This is a call back method allowing the state to perform any initialization need prior to being rendered in the state editor. For example, if the state uses a drop down input attribute. The drop down entries can be loaded from the data source in this method.

The context

SmappStateEditorContext

provides information that can be used to further filter the initialization.

Used by: state editor

*processMessage(SmappStateProcessingContext) method*

This method is called by the processing engine when the state is activated from external source, for example, incoming SMS message.

### Syntax

`SmappState processMessage ( SmappStateProcessingContext` *context* `)`
throws `MwizProcessingException, DBException, CryptoException`

### Parameters

- **context –** the state processing context containing all the necessary object needed while processing the state.

### Returns

the follow-up transition

### Exceptions

- **MwizProcessingException –** any exceptions that warrant terminating the application
- **DBException –** any database exceptions that warrant terminating the application
- **CryptoException –** any encryption exceptions that warrant terminating the application

### Usage

See this API documentation listed above.

Typically, this method should be implemented when the

supportsSendSmsMessage()

is set to true, to handle the reply SMS message.

Helper methods that will return the

SmappState

are as follow:

- continueOk()
- continueFail()
- continueFail(String)
- continueDyn(String)
- continueDyn(Integer)
- continueDyn(Long)

Used by: processing engine

the follow-up transition

### *processState(SmappStateProcessingContext, SmappStateProcessingAction) method*

This method is always called by the processing engine when the state is activated by the follow-up transition.

#### Syntax

`void processState ( SmappStateProcessingContext` *context* `, SmappStateProcessingAction` *action* `)` throws `MwizProcessingException, DBException, CryptoException`

#### Parameters

- **context –** the state processing context containing all the necessary object needed while processing the state.
- **action –** mechanism to communicate suggested follow-up action to the processing engine

#### Exceptions

- **MwizProcessingException –** any exceptions that warrant terminating the application
- **DBException –** any database exceptions that warrant terminating the application
- **CryptoException –** any encryption exceptions that warrant terminating the application

#### Usage

Used by: processing engine

### *supportEncryption() method*

Indicate whether this state support encryption.

#### Syntax

`boolean supportEncryption ()`

**Returns**
True/False for support encryption or not, respectively.

**Usage**

This will be used in the State Editor to enable/disable the encryption checkbox.

The state notes should described what is supported. For example, support encryption of session variables before storing in database, or support encryption of before logging in file or database.

True/False for support encryption or not, respectively.

### *supportsDynTransition() method*
Indicates if the state supports dynamic (Dyn) transition.

**Syntax**
```
boolean supportsDynTransition ()
```

**Returns**
True/False - support Dyn transition, or not, respectively

**Usage**

When set to true the state editor will display the dynamic dropdown UI control listing all the possible states to chose from.

The state may opt to support both OK and dynamic follow-up transitions, with the dynamic follow-up transitions for handling error conditions.

When returning true, the processing method(s) should have at least one condition that returns

```
continueDyn()
```

.

Used by: state editor

True/False - support Dyn transition, or not, respectively

### *supportsFailTransition() method*
Indicate if the state uses the Fail follow-up transition type.

**Syntax**
```
boolean supportsFailTransition ()
```

**Returns**
True/False - support fail transition, or not, respectively

#### Usage

When set to true the state editor will display the Fail dropdown UI control listing all the possible states to chose from.

When returning true, the processing method(s) should have at least one condition that returns

`continueFail()`

.

Typically, states with database or external web service calls should support the

`continueFail()`

to handle all the errors from these calls.

Used by: state editor

True/False - support fail transition, or not, respectively

### *supportsGoToApplication() method*

Indicate if the state supports transfer flow to another application.

#### Syntax

```
boolean supportsGoToApplication ()
```

#### Returns

True/False - support Goto application, or not, respectively

#### Usage

When set to true the state editor will display a dropdown UI control containing a list of applications in the workspace that can be "goto".

This is not a very useful method. It's mainly used the base state Goto Application to flag the state editor to display the control. This is not needed by other states because the same mechanism can be accomplished with the base Goto Application state.

Used by: state editor

True/False - support Goto application, or not, respectively

### *supportsOkTransition() method*

Indicate if the state uses the OK follow-up transition type.

#### Syntax

```
boolean supportsOkTransition ()
```

**Returns**
True/False - support OK transition, or not, respectively

**Usage**

When set to true the state editor will display the OK dropdown UI control listing all the possible states to chose from.

When returning true, the processing method(s) should have at least one condition that returns

`continueOk()`

.

It's possible that the state opts to not supporting the OK follow-up transition, but instead handle everything using the dynamic follow-up transition. See

supportsDynTransition()

.

Used by: state editor

True/False - support OK transition, or not, respectively

### *supportsSendSmsMessage() method*
Indicate if the state may send SMS message to the current consumer.

**Syntax**
`boolean supportsSendSmsMessage ()`

**Returns**
True/False - support send SMS, or not, respectively

**Usage**

When set to true the state editor will display a textbox UI control for entering the SMS message.

The SMS textbox will support all the functionalities supported in the base Send SMS state, like: the session variable token replacement, truncate message longer than 160 characters and send it as another message, etc.

When set to true, the state should provide implementation for the

processMessage(SmappStateProcessingContext)

method to handle activation from the reply message of this SMS.

Used by: state editor

True/False - support send SMS, or not, respectively

### *PluginInterface interface*

Plugin interfaces.

*Syntax*

```
public interface  PluginInterface
```

*Derived classes*

- *com.sybase365.mobiliser.brand.plugins.api.smapp.StatePlugin* on page 149
- *com.sybase365.mobiliser.brand.plugins.base.Plugin* on page 162

*Remarks*

To be implemented by class that needs to be plugged in the platform.

#### *getInstanceName() method*

**Syntax**

```
String getInstanceName ()
```

**Returns**

The identifier for this instance of this plugin.

**Usage**

The identifier for this instance of this plugin.

#### *getRevisionString() method*

**Syntax**

```
String getRevisionString ()
```

**Returns**

Version of the plugin.

**Usage**

Version of the plugin.

*setInstanceName(String) method*
A plugin may be instantiated multiple times with different set of attributes, the instanceName may be set externally to uniquely identify a particular instance.

**Syntax**
`void setInstanceName ( String` *instanceName* `)`

**Parameters**

- **instanceName –** The name to identify this instance of the plugin.

*shutdown() method*
Stop this plugin instance.

**Syntax**
`void shutdown ()`

*startup(HashMap< String, String >) method*
Initiate the startup process for this plugin instance.

**Syntax**
`void startup ( HashMap< String, String >` *attributes* `)` throws `MwizStartupException`

**Parameters**

- **attributes –** List of configuration attributes for the plugin

**Exceptions**

- **MwizStartupException –** If there is an issue with startup.

**Usage**

This method is called once per startup or during reload of the plugin.

*base package*

*Members*
All public members of the base package.

- **Plugin class –** This is the abstract super class of the PluginInterface.

### *Plugin class*

This is the abstract super class of the PluginInterface.

*Syntax*

```
public class  Plugin
```

*Derived classes*

- *com.sybase365.mobiliser.brand.plugins.smapp.state.SmappStatePlugin* on page 230

*Remarks*

Currently, there are two types of plugin supported:

- State - adding new states that can be used to compose application flow
- Infrastructure Channel - Input/Output message to/from the application

State plugin need to implement additional interfaces StatePlugin.

#### *getInstanceName() method*

Get the instance name.

**Syntax**

```
String getInstanceName ()
```

**Returns**

instance name

**Usage**

instance name

#### *setInstanceName(String) method*

Set the instance name.

**Syntax**

`void setInstanceName ( String` *instanceName* `)`

**Parameters**

- **instanceName –** name to assign to this instance of the plugin.

*instanceName variable*
Instance name.

*Syntax*
`String instanceName`

### *exceptions package*

*Members*
All public members of the exceptions package.

- **InputValueFormatException class –** InputValueFormatException can be used as a convenient way to catch all the input value format issue in the state development, as shown below.

*InputValueFormatException class*
InputValueFormatException can be used as a convenient way to catch all the input value format issue in the state development, as shown below.

*Syntax*
```
public class  InputValueFormatException
```

*Remarks*
```
        try {
            InputValue iValue;

            iValue = inCustomerName.getInputValue();
            if (iValue!=null) {
              String name = iValue.getString().trim();
                if (!validFullName(name)) {
                   throw new InputValueFormatException("Invalid: " +
                                 inCustomerName.getDescription());
                }
                customer.setDisplayName(name);
            }

            //
            // Check the second attribute, etc..
            //

        } catch (InputValueFormatException fex) {

            //
            // Catch all in one location
            //
            log.error(fex.getMessage(), fex);
            return continueDyn(-9);
        }
```

### *InputValueFormatException(String) constructor*

**Syntax**
```
InputValueFormatException ( String message )
```

### *InputValueFormatException(String, Throwable) constructor*

**Syntax**
```
InputValueFormatException ( String message ,  Throwable th )
```

### *InputValueFormatException(Throwable) constructor*

**Syntax**
```
InputValueFormatException ( Throwable th )
```

## *smapp package*

*Members*
All public members of the smapp package.

- **beans package –**
- **controls package –**
- **state package –**

### *beans package*

*Members*
All public members of the beans package.

- **BeanConverterInterface< T > interface –** Bean that needs to be persisted to the session variable using the SessionVariableAttribute needs to implement this interface.
- **GenericBean class –** For beans to be persisted to session variable, it needs to implement the BeanConverterInterface.

### *BeanConverterInterface< T > interface*
Bean that needs to be persisted to the session variable using the SessionVariableAttribute needs to implement this interface.

*Syntax*
```
public interface  BeanConverterInterface< T >
```

*Remarks*
The bean will be serialized into String before saving to the session variable. Conversely, on retrieval the string be de-serialized back to the bean.

*convert(String) method*
Called when de-serializing the string retrieved from the session variable back into the bean.

**Syntax**
`T convert ( String` *value* `)`

**Parameters**

- **value –** String from the session variable

**Returns**
bean of type `T` after the de-serialization

**Usage**

bean of type `T` after the de-serialization

*convert(T) method*
Used to serialized the bean of type T into string so that it can be saved into the session variable.

**Syntax**
`String convert ( T` *object* `)`

**Parameters**

- **object –** bean of type `T` that needs to be serialized

**Returns**
serialized string

**Usage**

serialized string

*GenericBean class*
For beans to be persisted to session variable, it needs to implement the BeanConverterInterface.

*Syntax*
```
public class  GenericBean
```

*Remarks*

So typically, like in the dynamic SMS menu that extends the AbstractDynamicMenu, a helper bean that implements the BeanConverterInterface is created. The bean (i.e., domain bean) is transformed to the helper bean and then serialize.

When domain bean is simple, containing fields of String type only, this helper GenericBean can be used instead of creating a custom bean. This bean contains a unique ID field, and 10 (0..9) properties.

Simple example of how to store list of GenericBean into session variable:

```
        List beanList = new ArrayList<GenericBean>();
        for (DomainObject dObj: domainObjList) {
            // Convert domain object to HashMap
            HashMap<String, String> objMap = 
convertDomainObjToMap(dObj);
            beanList.add(GenericBean.parse(objMap));
        }

        // Store the list into session variable 
SessionVariableAttribute
        SessionVariableAttribute session = new 
SessionVariableAttribute("SESSION_LIST", "");
        session.setList(beanList);
```

To retrieve the list from session variable and convert back to GenericBean:

```
        List beanList = new ArrayList<GenericBean>();

        // Session variable defined earlier
        beanList = session.getList(new GenericBean());
```

*compareTo(GenericBean) method*

Sortable by the unique Id, or GB_ID.

**Syntax**

`int compareTo ( GenericBean` *o* `)`

*convert(String) method*

Called to transform the session variable value back into this bean.

**Syntax**

`GenericBean convert ( String` *value* `)`

*convert(GenericBean) method*

Called to transform the bean into string for persistence to session variable.

**Syntax**

`String convert ( GenericBean` *bean* `)` *equals(Object) method*

**Syntax**

`boolean equals ( Object` *obj* `)`

*getAttrib0() method*

**Syntax**

`String getAttrib0 ()`

**Returns**

Property 1

**Usage**

Property 1

*getAttrib1() method*

**Syntax**

`String getAttrib1 ()`

**Returns**

Property 2

**Usage**

Property 2

*getAttrib2() method*

**Syntax**

`String getAttrib2 ()`

**Returns**

Property 3

**Usage**

Property 3

*getAttrib3() method*

**Syntax**
`String getAttrib3 ()`

**Returns**
Property 4

**Usage**

Property 4

*getAttrib4() method*

**Syntax**
`String getAttrib4 ()`

**Returns**
Property 5

**Usage**

Property 5

*getAttrib5() method*

**Syntax**
`String getAttrib5 ()`

**Returns**
Property 6

**Usage**

Property 6

*getAttrib6() method*

**Syntax**
`String getAttrib6 ()`

**Returns**
Property 7

**Usage**

Property 7

*getAttrib7() method*

**Syntax**

`String getAttrib7 ()`

**Returns**

Property 8

**Usage**

Property 8

*getAttrib8() method*

**Syntax**

`String getAttrib8 ()`

**Returns**

Property 9

**Usage**

Property 9

*getAttrib9() method*

**Syntax**

`String getAttrib9 ()`

**Returns**

Property 10

**Usage**

Property 10

*getId() method*

**Syntax**

`String getId ()`

**Returns**
Unique Id

**Usage**

Unique Id

*hashCode() method*

**Syntax**
`int hashCode ()`

*parse(HashMap< String, String >) method*
Domain bean needs to be converted to HashMap, and use this method to transform into this GenericBean.

**Syntax**
`GenericBean parse ( HashMap< String, String > obj )`

**Parameters**

- **obj –** list of values to be converted to GenericBean.

**Returns**
Transformed bean from the map contents.

**Usage**

Use the provided keys for the

HashMap

entries.

Helper method to transform into this bean.

NOTE

: the combined length of the values string should NOT exceed 1000 chars, because the session storage is limited to 1000 chars

Transformed bean from the map contents.

*setAttrib0(String) method*

**Syntax**
`void setAttrib0 ( String attrib0 )`

**Parameters**

- **attrib0 –** Property 1

*setAttrib1(String) method*

**Syntax**

`void setAttrib1 ( String` *attrib1* `)`

**Parameters**

- **attrib1 –** Property 2

*setAttrib2(String) method*

**Syntax**

`void setAttrib2 ( String` *attrib2* `)`

**Parameters**

- **attrib2 –** Property 3

*setAttrib3(String) method*

**Syntax**

`void setAttrib3 ( String` *attrib3* `)`

**Parameters**

- **attrib3 –** Property 4

*setAttrib4(String) method*

**Syntax**

`void setAttrib4 ( String` *attrib4* `)`

**Parameters**

- **attrib4 –** Property 5

*setAttrib5(String) method*

**Syntax**

`void setAttrib5 ( String` *attrib5* `)`

**Parameters**

- **attrib5 –** Property 6

*setAttrib6(String) method*

**Syntax**

`void setAttrib6 ( String` *attrib6* `)`

**Parameters**

- **attrib6 –** Property 7

*setAttrib7(String) method*

**Syntax**

`void setAttrib7 ( String` *attrib7* `)`

**Parameters**

- **attrib7 –** Property 8

*setAttrib8(String) method*

**Syntax**

`void setAttrib8 ( String` *attrib8* `)`

**Parameters**

- **attrib8 –** Property 9

*setAttrib9(String) method*

**Syntax**

`void setAttrib9 ( String` *attrib9* `)`

**Parameters**

- **attrib9 –** Property 10

*setId(String) method*

**Syntax**

`void setId ( String` *id* `)`

**Parameters**

- **id –** Unique Id

*GB_ATTRIB0 variable*

Key for field 1 Used for the HashMap input parameter for the parse(HashMap) method.

*Syntax*

```
final String GB_ATTRIB0
```

*GB_ATTRIB1 variable*

Key for field 2 Used for the HashMap input parameter for the parse(HashMap) method.

*Syntax*

```
final String GB_ATTRIB1
```

*GB_ATTRIB2 variable*

Key for field 3 Used for the HashMap input parameter for the parse(HashMap) method.

*Syntax*

```
final String GB_ATTRIB2
```

*GB_ATTRIB3 variable*

Key for field 4 Used for the HashMap input parameter for the parse(HashMap) method.

*Syntax*

```
final String GB_ATTRIB3
```

*GB_ATTRIB4 variable*

Key for field 5 Used for the HashMap input parameter for the parse(HashMap) method.

*Syntax*

```
final String GB_ATTRIB4
```

*GB_ATTRIB5 variable*

Key for field 6 Used for the HashMap input parameter for the parse(HashMap) method.

*Syntax*

```
final String GB_ATTRIB5
```

*GB_ATTRIB6 variable*

Key for field 7 Used for the HashMap input parameter for the parse(HashMap) method.

*Syntax*

```
final String GB_ATTRIB6
```

### *GB_ATTRIB7 variable*

Key for field 8 Used for the HashMap input parameter for the parse(HashMap) method.

#### *Syntax*

```
final String GB_ATTRIB7
```

### *GB_ATTRIB8 variable*

Key for field 9 Used for the HashMap input parameter for the parse(HashMap) method.

#### *Syntax*

```
final String GB_ATTRIB8
```

### *GB_ATTRIB9 variable*

Key for field 10 Used for the HashMap input parameter for the parse(HashMap) method.

#### *Syntax*

```
final String GB_ATTRIB9
```

### *GB_ID variable*

Key for the Unique ID Used for the HashMap input parameter for the parse(HashMap) method.

#### *Syntax*

```
final String GB_ID
```

### *controls package*

#### *Members*

All public members of the controls package.

- **Attribute class –** Attribute is the super abstract class that is extended by both the InputAttribute and OutputAttribute.
- **InputAttribute class –** InputAttribute is used by the state to gather input value that will be used in the state processing, or to be included with the external web service call.
- **OutputAttribute class –** OutputAttribute is used by the state to store output resulting from processing the state.
- **SelectionBoxAttribute class –** Input attribute of selection dropdown type.
- **SessionVariableAttribute class –** This is a special type of Attribute that does not have the UI component on the State Editor.
- **TextBoxAttribute class –** Input attribute of textbox type.

### *Attribute class*

Attribute is the super abstract class that is extended by both the InputAttribute and OutputAttribute.

#### *Syntax*

```
public class  Attribute
```

#### *Derived classes*

- *com.sybase365.mobiliser.brand.plugins.smapp.controls.InputAttribute* on page 178
- *com.sybase365.mobiliser.brand.plugins.smapp.controls.OutputAttribute* on page 190
- *com.sybase365.mobiliser.brand.plugins.smapp.controls.SessionVariableAttribute* on page 199

#### *Remarks*

This class encapsulates the basic fields of an attribute, and provide the get/set methods. Attribute(s) are used in the state implementation to represent input and output attributes.

In addition, Attribute holds the state processing context that is used, for example, to access or store the attribute value as a session variable in the datasource so that it is durable across session. The state processing context is assigned to the attribute by the processing engine during initialization of the state using the setContext(SmappStateProcessingContext) method. Hence, this method is reserved to be used only by the processing engine.

NOTE: Attributes that are used in the same states must have unique ids.

### *Attribute(String, String) constructor*

Default value (if provided)

#### Syntax

`Attribute ( String` *id* `,  String` *description* `)`

#### Parameters

- **id –** unique Id for the attribute
- **description –** description that will be displayed on the state editor

#### Usage

Constructor

Attribute

s within the same state needs to have a unique id

### *getContext() method*

Sets the processing engine context.

#### Syntax

`SmappStateProcessingContext getContext ()`

#### Returns

state processing context

#### Usage

SmappStateProcessingContext

is used in getting the datasource to retrieve or persist the attribute as a session variable so that it is durable across session.

state processing context

### *getDescription() method*

Detailed description of the attribute.

#### Syntax

`String getDescription ()`

#### Returns

the description of the attribute

#### Usage

Used in the UI State Editor.

the description of the attribute

### *getId() method*

Attribute Id.

#### Syntax

`String getId ()`

#### Returns

attribute id

#### Usage

Used as a default session variable name.

attribute id

*getText() method*
The text that was entered in the input field on the state editor.

**Syntax**
`String getText ()`

**Returns**
input text

**Usage**

input text

*setContext(SmappStateProcessingContext) method*
Sets the processing engine context.

**Syntax**
`void setContext ( SmappStateProcessingContext` *context* `)`

**Parameters**

- **context –** state processing context

**Usage**

SmappStateProcessingContext

is used by the attribute to access or store the attribute value as a session variable in the datasource so that it is durable across session.

WARNING

: please refrain from setting or resetting the

SmappStateProcessingContext

. This method is reserved to be used by the processing engine only.

*setDescription(String) method*
Detailed description of the attribute.

**Syntax**
`void setDescription ( String` *description* `)`

## Parameters

- **description –** the description of the attribute

## Usage

Used in the UI State Editor.

*setId(String) method*
Attribute Id.

## Syntax

`void setId ( String` *id* `)`

## Parameters

- **id –** attribute id

## Usage

Used as a default session variable name.

*description variable*
Description that will be displayed on the state editor for user consumption.

*Syntax*
`String description`

*id variable*
Unique Id of the attribute.

*Syntax*
`String id`

*Remarks*
Used as a default session variable name.

*InputAttribute class*
InputAttribute is used by the state to gather input value that will be used in the state processing, or to be included with the external web service call.

*Syntax*
```
public class  InputAttribute
```

### *Derived classes*

- *com.sybase365.mobiliser.brand.plugins.smapp.controls.SelectionBoxAttribute* on page 195
- *com.sybase365.mobiliser.brand.plugins.smapp.controls.TextBoxAttribute* on page 201

### *Remarks*

On the State Editor, InputAttribute is referred to as "Input Variable" and is represented with three UI components:

- A checkbox
- An "input field" (textbox or dropdown box)
- An mandatory attribute indicator
- A description of the input attribute
- Mouse-hover showing the description, and the default session variable name

The checkbox indicates whether the value entered in the "input field" should be treated as "<i>session variable name</i>" when checked, or "<i>static value</i>" when not checked. When checked, the session variable name will be used in retrieving the value from the session variable.

In either cases, the value will always be persisted for durability across session.

Also, see OutputAttribute.

### *InputValue class*

Helper class to return the input value and provide helper methods to convert the value into needed types.

### *Syntax*

```
public class  InputValue
```

### *Remarks*

Internally, InputValue has two fields: name and value. If not provided, the name will be set to an empty string. The name and value will be set in the exception message when occurs.

### *InputValue(String) constructor*

Constructor.

#### **Syntax**

`InputValue ( String` *value* `)`

**Parameters**

- **value –** input value

*InputValue(String, String) constructor*
Constructor.

**Syntax**
`InputValue ( String` *varName* `,  String` *value* `)`

**Parameters**

- **varName –** input variable name; to be set on the exception message
- **value –** input variable

*getBoolean() method*
The Boolean value of this input.

**Syntax**
```
Boolean getBoolean ()
```

**Returns**
Boolean value

**Usage**

Null or empty value will return false.

In valid string will return false.

Boolean value

*getDouble() method*
The Double value of this input.

**Syntax**
```
Double getDouble ()
```

**Returns**
Double value

**Exceptions**

- **NumberFormatException –** the value cannot be converted to Double

**Usage**

Double value

*getInt() method*
The Integer value of this input.

**Syntax**
`Integer getInt ()`

**Returns**
Integer value

**Exceptions**

- **NumberFormatException –** the value cannot be converted to Integer

**Usage**

Integer value

*getLong() method*
The Long value of this input.

**Syntax**
`Long getLong ()`

**Returns**
Long value

**Exceptions**

- **NumberFormatException –** the value cannot be converted to Long

**Usage**

Long value

*getMsisdn(String) method*
Get the MSISDN in international format based on the specified country code.

**Syntax**
`String getMsisdn ( String` *countryCode* `)`

#### Parameters

- **countryCode –** country code

#### Returns

MSISDN in International format string

#### Usage

Null will return empty string.

MSISDN in International format string

### *getString() method*

The String value of this input.

#### Syntax

`String getString ()`

#### Returns

input value

#### Usage

input value

### *getString(int) method*

Get the string value that is truncated to the specify size.

#### Syntax

`String getString ( int` *size* `)`

#### Parameters

- **size –** truncate to the specify size

#### Returns

format string

#### Usage

format string

*toString() method*
Returns a string representation of this instance.

**Syntax**
`String toString ()`

*InputAttribute(String, String, boolean) constructor*
Constructor.

**Syntax**
`InputAttribute ( String` *id* `,  String` *description* `,  boolean` *optional* `)`

**Parameters**

- **id –** unique id of this state. NOTE: id needs to be unique within the same state
- **description –** detailed description of input attribute, to be displayed on the State Editor
- **optional –** True/False for whether the field is optional or not, respectively

*getInputType() method*
Input type can be changed from the State Editor by checking/unchecking the input attribute checkbox, to set to InputType#SESSION or InputType#ATTRIBUTE, respectively.

**Syntax**
`InputType getInputType ()`

**Returns**
InputType of this input attribute

**Usage**

By default, it is set to

InputType#SESSION

.

This method is used by the State Editor to check/uncheck the input attribute checkbox.

InputType of this input attribute

### *getInputValue() method*

More efficient way to check and obtain the value of the input attribute using a single database call.

#### Syntax

`abstract InputValue getInputValue ()` throws `DBException`

#### Returns

NULL - the variable has not been populated InputValue - populated value

#### Exceptions

- **DBException –** Exception when accessing the session variable from database
- **RequiredParameterMissingException –** This is a required input field but the value has not been populated

#### Usage

The return InputValue can be used to retrieve the actual input value.

In the past, this has been accomplished using a combined call to

isSet()

and

getValue()

methods which is less efficient because each method make a separate database call.

```
This method is designed to be used with an optional (not mandatory)
input attribute. For example,

  InputValue iv = optionalInputVariable.getInputValue();
  if (iv!=null) {
     retrieve the value
  }
 For mandatory input attribute, access the value directly using the
getInputValueWithWarning(), and handle the
RequiredParameterMissingException appropriately.
```

NULL - the variable has not been populated InputValue - populated value

### *getInputValueWithWarning() method*

Similar to getInputValue() but this method throws RequiredParameterMissingException when the input attribute is not optional (or mandatory) but the value is null (or not populated).

#### Syntax

`InputValue getInputValueWithWarning ()` throws `DBException, RequiredParameterMissingException`

#### Returns

value in InputValue type

#### Exceptions

- **DBException –** Exception when accessing the session variable from database
- **RequiredParameterMissingException –** This is a required input field but the value has not been populated

#### Usage

This method is useful for processing multiple required input attributes as shown below:

```
        try {
            // Attribute 1
            attrib1.getInputValueWithWarning();

            // Attribute 2
            attrib2.getInputValueWithWarning();

        catch (RequiredParameterMissingException rex) {
            handle the missing mandatory attributes
        }
```

value in InputValue type

### *getRawValue() method*

Helper class used internally by the getInputValue(), getInputValueWithWarning(), and getValue().

#### Syntax

`String getRawValue ()` throws `DBException, RequiredParameterMissingException`

#### Returns

The unmodified value of the attribute.

#### Exceptions

- **DBException –** Exception when accessing the session variable from database
- **RequiredParameterMissingException –** This is a required input field but the value has not been populated

#### Usage

The unmodified value of the attribute.

*getRawValueLog() method*
Helper class called internally by getRawValue().

#### Syntax
`String getRawValueLog ()` throws `DBException, RequiredParameterMissingException`

#### Returns
value in string type

#### Exceptions

- **DBException –** Exception when accessing the session variable from database
- **RequiredParameterMissingException –** This is a required input field but the value has not been populated

#### Usage

Overriden by the

SelectionBoxAttribute

.

value in string type

*getValue() method*
Get the InputValue with no warning.

#### Syntax
`InputValue getValue ()` throws `DBException`

#### Returns
input value

### Exceptions

- **DBException –** Exception when accessing the session variable from database

### Usage

When the input value is required but not populated, an InputValue with empty string is quietly returned.

NOTE:

the recommended methods for retrieving input attribute value are:

getInputValue()

or

getInputValueWithWarning()

.

input value

### *isCheckboxEnabled() method*

Whether the UI checkbox component is enabled or not.

### Syntax

```
boolean isCheckboxEnabled ()
```

### Returns

True/Flase for the checkbox is checked or not, respectively.

### Usage

Whe enabled, user can check or uncheck.

In some cases, the state implementation may want the checkbox to be "not changeable" from the State Editor. In this case, the checkbox can be disabled using then

setCheckboxEnabled(boolean)

.

The checkbox indicates whether the value entered in the "input field" should be treated as "<i>session variable name</i>" when checked, or "<i>static value</i>" when not checked.

This method is used by the UI State Editor to enable/disable the UI checkbox component.

True/Flase for the checkbox is checked or not, respectively.

### *isOptional() method*

Whether the input attribute is optional.

#### Syntax

`boolean isOptional ()`

#### Returns

the optional True if optional; False is required input attribute.

#### Usage

the optional True if optional; False is required input attribute.

### *isSet() method*

Return True/False when the InputAttribute contains value or not, respectively.

#### Syntax

`abstract boolean isSet ()` throws `DBException`

#### Returns

True/False when the InputAttribute contains value or not, respectively.

#### Exceptions

- **DBException –** Exception when accessing the session variable from database

#### Usage

NOTE

: use this method only for finding out whether the

InputAttribute

contains value, but the value itself is not needed. Otherwise, use the more efficient methods:

getInputValue()

or

getInputValueWithWarning()

that will return null if the input attribute is not not set.

True/False when the InputAttribute contains value or not, respectively.

*setCheckboxEnabled(boolean) method*
Enable or disable the checkbox.

**Syntax**
`void setCheckboxEnabled ( boolean` *checkboxEnabled* `)`

**Parameters**

- **checkboxEnabled –** True/Flase for the checkbox is checked or not, respectively.

**Usage**

This method can be used by state implementation to set whether the checkbox can be checked/ unchecked.

In some cases, the state implementation may want the checkbox to be "not changeable" from the State Editor. In this case, the checkbox can be set to disabled so that it is not changable.

The checkbox indicates whether the value entered in the "input field" should be treated as "<i>session variable name</i>" when checked, or "<i>static value</i>" when not checked.

*setInputType(InputType) method*
Input type can be changed from the State Editor by checking/unchecking the input attribute checkbox, to set it to InputType#SESSION or InputType#ATTRIBUTE, respectively.

**Syntax**
`void setInputType ( InputType` *inputType* `)`

**Parameters**

- **inputType –** InputType of this input attribute

**Usage**

This method is used by the UI State Editor when the checkbox is changed.

By default, it is set to

InputType#SESSION

.

However,

SelectionBoxAttribute

overwrite the default to

InputType#ATTRIBUTE

.

### *setOptional(boolean) method*

Set the input attribute optional status.

#### Syntax

```
void setOptional ( boolean optional )
```

#### Parameters

- **optional –** True if optional; False is required input attribute.

#### Usage

NOTE:

Typically, the optional status is set in the constructor.

### *InputType() enumeration*

Indicate how to treat the value provided in the input field, either as a "static value" or as a "session variable name" to retrieve the actual value from session variable using the specified name.

#### *Enum Constant Summary*

- **ATTRIBUTE –** Indicate that the input value is a static value, hence it can be used as is.
- **SESSION –** Indicate that the input value is a session variable name, hence the input value needs to be retrieved from the session variable of the specified name.

### *encrypted variable*

Indicator of whether this attribute is encrypted or not.

#### *Syntax*

```
boolean encrypted
```

#### *Remarks*

NOTE: currently it's just a place holder for future implementation. Setting it to true does NOT encrypt the attribute value yet.

### *OutputAttribute class*

OutputAttribute is used by the state to store output resulting from processing the state.

#### *Syntax*

```
public class  OutputAttribute
```

*Remarks*

This could be the results from external web service call, status code, error message, error code, etc. The value set to the output attribute will be automatically saved to the session variable so that it can be used even when the state is out of scope, or no longer in the processing mode.

On the State Editor, the OutputAttribute is referred to as "Output Variable" and is represented with the following UI components:

- A checkbox. Checked and not editable.
- Textbox field for specifying the session variable name to store the output value. By default, it is set to this output attribute ID.
- A description of the output attribute
- Mouse-hover showing the description, and the default session variable name

The output attribute has only one type, the InputType#SESSION type.

There are two mechanisms to set value on the output attribute: "<i>set and persist</i>", and "<i>set and hold</i>". The "set and persist" using one of the setValue() methods will set the value and immediately persist it to the session variable. This is good if there are only a few output attributes to set, because each calls will result in an independent call to the datasource.

```
If there are many output attributes to set, the later mechanism (set
and hold) is more efficient because the value is temporarily held in
the output attribute until an explicit bulk save is called by the
state implementation as shown on the following.

    outAttrib1.setHoldValue("value1");
    outAttrib2.setHoldValue("value2");
    outAttrib3.setHoldValue("value3");


    // The following helper method belongs to the SmappStatePlugin
    // This method will also call the resetHoldValue() method
    // of each output attributes, after a successful save.
    saveOutputAttributes();
```

*OutputAttribute(String, String) constructor*

Constructor.

**Syntax**

`OutputAttribute ( String` *id* `,  String` *description* `)`

**Parameters**

- **id –** unique Id for the attribute
- **description –** description that will be displayed on the state editor

### *getHoldValue() method*

The holdValue pending to bulk save.

#### Syntax

`String getHoldValue ()`

#### Returns

Null - when there is no held value; Otherwise, the temporary held value

#### Usage

After a successful save using the

SmappStatePlugin

`saveOutputAttributes()`

, the value will be automativcally set to

`null`

.

Null - when there is no held value; Otherwise, the temporary held value

### *resetHoldValue() method*

Reset the holdValue to null.

#### Syntax

`void resetHoldValue ()`

### *setHoldValue(String) method*

Set the holdValue to the given string.

#### Syntax

`void setHoldValue ( String` *value* `)`

#### Parameters

- **value –** string value to set the holdValue

### *setHoldValue(Long) method*

Set the holdValue to the string value of the given Long value.

#### Syntax

`void setHoldValue ( Long` *value* `)`

**Parameters**

- **value –** Long value to set the holdValue

*setHoldValue(Integer) method*
Set the holdValue to the string value of the given Integer value.

**Syntax**
`void setHoldValue ( Integer` *value* `)`

**Parameters**

- **value –** Integer value to set the holdValue

*setHoldValue(Boolean) method*
Set the holdValue to the string value of the given Boolean value.

**Syntax**
`void setHoldValue ( Boolean` *value* `)`

**Parameters**

- **value –** Boolean value to set the holdValue

*setValue(String) method*
Set the value to this output attribute, and also persist the value to the session variable using the name specified in the output textbox field.

**Syntax**
`void setValue ( String` *value* `)` throws `DBException`

**Parameters**

- **value –** string value to be set on the output attribute

**Exceptions**

- **DBException –** Exception when setting the session variable to datasource

**Usage**

By default, the output textbox is set to the ID of this output attribute.

*setValue(Long) method*

Set the string value of given Long to this output attribute, and also persist the value to the session variable using the name specified in the output textbox field.

**Syntax**

`void setValue ( Long` *value* `)` throws `DBException`

**Parameters**

- **value –** Long value to be set on the output attribute

**Exceptions**

- **DBException –** Exception when setting the session variable to datasource

**Usage**

By default, the output textbox is set to the ID of this output attribute.

*setValue(Integer) method*

Set the string value of given Integer to this output attribute, and also persist the value to the session variable using the name specified in the output textbox field.

**Syntax**

`void setValue ( Integer` *value* `)` throws `DBException`

**Parameters**

- **value –** Integer value to be set on the output attribute

**Exceptions**

- **DBException –** Exception when setting the session variable to datasource

**Usage**

By default, the output textbox is set to the ID of this output attribute.

*setValue(Boolean) method*

Set the string value of given Boolean to this output attribute, and also persist the value to the session variable using the name specified in the output textbox field.

**Syntax**

`void setValue ( Boolean` *value* `)` throws `DBException`

### Parameters

- **value –** Boolean value to be set on the output attribute

### Exceptions

- **DBException –** Exception when setting the session variable to datasource

### Usage

By default, the output textbox is set to the ID of this output attribute.

*SelectionBoxAttribute class*
Input attribute of selection dropdown type.

*Syntax*
`public class SelectionBoxAttribute`

*Remarks*
The State Editor will display this input attribute using a selection dropdown component with preset list. User can select a value from the dropdown list but not entering a new one.

By default, the SelectionBoxAttribute has a InputAttribute.InputType#ATTRIBUTE type, so the input attribute checkbox will be unchecked, and it means the value is static and will be used as is. The choice made by the user on the State Editor will be treated as a static value, and used as is. The dropdown is populated with a "Key-Value" list, and the selected value is the "Key". The "Value" is the displayed text shown on the UI.

This SelectionBoxAttribute can also be set to InputAttribute.InputType#SESSION type, to retrieve the value from the session variable. The name of the session variable will be the ID of this selectionbox attribute, and it cannot be changed. On the state editor, the ID will be shown on mouse-hover over the input attribute description. When set to the InputAttribute.InputType#SESSION type, the selection should be left null.

*SelectionBoxAttribute(String, String, boolean) constructor*
Constructor.

### Syntax

`SelectionBoxAttribute ( String` *id* `, String` *description* `, boolean` *optional* `)`

### Parameters

- **id –** unique id of this state. NOTE: id needs to be unique within the same state
- **description –** detailed description of input attribute, to be displayed on the state editor

- **optional –** True/False for whether the field is optional or not, respectively

### *getInputValue() method*

More efficient way to check and obtain the value of the input attribute using a single database call.

**Syntax**

`InputValue getInputValue ()` throws `DBException`

**Returns**

NULL - the variable has not been populated InputValue - populated value

**Exceptions**

- **DBException –** Exception when accessing the session variable from database
- **RequiredParameterMissingException –** This is a required input field but the value has not been populated

**Usage**

The return InputValue can be used to retrieve the actual input value.

In the past, this has been accomplished using a combined call to

isSet()

and

getValue()

methods which is less efficient because each method make a separate database call.

```
This method is designed to be used with an optional (not mandatory)
input attribute. For example,

  InputValue iv = optionalInputVariable.getInputValue();
  if (iv!=null) {
     retrieve the value
  }
 For mandatory input attribute, access the value directly using the
getInputValueWithWarning(), and handle the
RequiredParameterMissingException appropriately.
```

NULL - the variable has not been populated InputValue - populated value

### *getItems() method*

List of items that will be used for the dropdown list.

**Syntax**

`List< KeyValuePair< String, String > > getItems ()`

**Returns**
List of KeyValuePair items for the dropdown list.

**Usage**

List of KeyValuePair items for the dropdown list.

*getItemValue(String) method*
Helper method to find the value of the specified key from the dropdown list.

**Syntax**
`String getItemValue ( String` *key* `)`

**Parameters**

- **key –** Key

**Returns**
Value of the specified key

**Usage**

Value of the specified key

*getRawValueLog() method*
Helper class called internally by getRawValue().

**Syntax**
`String getRawValueLog ()` throws `DBException,`
`RequiredParameterMissingException`

**Returns**
value in string type

**Exceptions**

- **DBException –** Exception when accessing the session variable from database
- **RequiredParameterMissingException –** This is a required input field but the value has not been populated

**Usage**

Overriden by the

SelectionBoxAttribute

.

value in string type

### *isSet() method*

Return True/False when the InputAttribute contains value or not, respectively.

#### Syntax

`boolean isSet ()` throws `DBException`

#### Returns

True/False when the InputAttribute contains value or not, respectively.

#### Exceptions

- **DBException –** Exception when accessing the session variable from database

#### Usage

NOTE

: use this method only for finding out whether the

InputAttribute

contains value, but the value itself is not needed. Otherwise, use the more efficient methods:

getInputValue()

or

getInputValueWithWarning()

that will return null if the input attribute is not not set.

True/False when the InputAttribute contains value or not, respectively.

### *setItems(List< KeyValuePair< String, String >>) method*

Set the list of items to be used in populating the dropdown list.

#### Syntax

`void setItems ( List< KeyValuePair< String, String >>` *list* `)`

#### Parameters

- **list –** List of KeyValuePair items for the dropdown list.

#### Usage

```
In the state implementations, this method can be called from the
StatePlugin#getInputAttributes() method to set the list, as shown on
```

```
the following.

        List<KeyValuePair<String, String>> list = new
ArrayList<KeyValuePair<String, String>>();
       list.add(new KeyValuePair<String, String>("key1", "value1"));
       list.add(new KeyValuePair<String, String>("key2", "value2"));
       list.add(new KeyValuePair<String, String>("key3", "value3"));
        if (list.size()>0) inInputAttr.setItems(credTypes);


list


List of KeyValuePair items for the dropdown list.
```

*SessionVariableAttribute class*

This is a special type of Attribute that does not have the UI component on the State Editor.

*Syntax*

```
public class  SessionVariableAttribute
```

*Remarks*

SessionVariableAttribute is used to hold and persist List of BeanConverterInterface type.

```
Each item in the list is stored in one session variable using the
following naming convention for the session variable name.

      this_sessionVariableAttribute_ID + "_" + state_ID + "[" + index
+ "]"
```

where: index is the list index.

```
In addition, one session variable with the following name is used to
store the total number of item in the list.

        this_sessionVariableAttribute_ID + "_" + state_ID + "_count"
```

SessionVariableAttribute is used in the AbstractDynamicMenu to store the list in the input parameter of the `setMenuListToSession(List<T>)` method. This list is used to generate SMS menu message.

*SessionVariableAttribute(String, String) constructor*

**Syntax**

`SessionVariableAttribute ( String` *id* `,  String` *description* `)`

### *getList(BeanConverterInterface< T >) method*

Get the complete List of BeanConverterInterface type from the session variable.

#### Syntax

`public< T > List< T > getList ( BeanConverterInterface< T >` *beanType* `)` throws `DBException`

#### Parameters

- **beanType –** bean that implement the BeanConverterInterface

#### Returns

List containing the BeanConverterInterface type

#### Exceptions

- **DBException –** Exception when retrieving the session variable from datasource

#### Usage

List containing the BeanConverterInterface type

### *setList(List< T >) method*

Set the List that needs to be stored in this SessionVariableAttribute.

#### Syntax

`public< T extends BeanConverterInterface< T > > void setList ( List< T >` *list* `)` throws `DBException`

#### Parameters

- **list –** List to be stored

#### Exceptions

- **DBException –** Exception when setting the session variable to datasource

#### Usage

NOTE

: each item in the list is saved independently and not atomic. Therefore, if DBException occurs while saving in the middle of the list, the rest of list items will be abandoned resulting in a partial save.

*TextBoxAttribute class*
Input attribute of textbox type.

*Syntax*
`public class TextBoxAttribute`

*Remarks*
The State Editor will display this input field using a textbox component, allowing user to enter value.

By default, this TextBoxAttribute is of InputAttribute.InputType#SESSION (so the checkbox will be checked), and the input field will be populated with the string ID of this input attribute. So during state processing, the input attribute value will be retrieved from the session variable of the input attribute ID.

Change the value of the input field, to retrieve the input attribute value from a different session variable name.

The textbox attribute type can be changed to InputAttribute.InputType#ATTRIBUTE so that the value in the input field is used as a static value.

*TextBoxAttribute(String, String, boolean) constructor*
Constructor.

**Syntax**
`TextBoxAttribute ( String` *id* `, String` *description* `, boolean` *optional* `)`

**Parameters**

- **id –** unique id of this state. NOTE: id needs to be unique within the same state
- **description –** detailed description of input attribute, to be displayed on the state editor
- **optional –** True/False for whether the field is optional or not, respectively

*getInputValue() method*
More efficient way to check and obtain the value of the input attribute using a single database call.

**Syntax**
`InputValue getInputValue ()` throws `DBException`

**Returns**
NULL - the variable has not been populated InputValue - populated value

### Exceptions

- **DBException –** Exception when accessing the session variable from database
- **RequiredParameterMissingException –** This is a required input field but the value has not been populated

### Usage

The return InputValue can be used to retrieve the actual input value.

In the past, this has been accomplished using a combined call to

isSet()

and

getValue()

methods which is less efficient because each method make a separate database call.

```
This method is designed to be used with an optional (not mandatory)
input attribute. For example,

  InputValue iv = optionalInputVariable.getInputValue();
  if (iv!=null) {
     retrieve the value
  }
 For mandatory input attribute, access the value directly using the
getInputValueWithWarning(), and handle the
RequiredParameterMissingException appropriately.
```

NULL - the variable has not been populated InputValue - populated value

*isSet() method*
Return True/False when the InputAttribute contains value or not, respectively.

### Syntax

`boolean isSet ()` throws `DBException`

### Returns

True/False when the InputAttribute contains value or not, respectively.

### Exceptions

- **DBException –** Exception when accessing the session variable from database

### Usage

NOTE

: use this method only for finding out whether the

InputAttribute

contains value, but the value itself is not needed. Otherwise, use the more efficient methods:

getInputValue()

or

getInputValueWithWarning()

that will return null if the input attribute is not not set.

True/False when the InputAttribute contains value or not, respectively.

### *state package*

#### *Members*

All public members of the state package.

- **AbstractDynamicMenu class –** Abstract implementation of the dynamic menu state.
- **Page class –** Page class represents a one page of menu items based on the specified maximum number of items per page.
- **RequiredParameterMissingException class –** This exception is thrown during runtime in the processing engine when the com.sybase365.mobiliser.brand.plugins.smapp.controls.InputAttribute is designated as 'not optional' or "mandatory" but no value was provided.
- **SmappStatePlugin class –** This class represents the main class to be inherited by *state* that needs to be displayed on the Application Composer and State Editor, and to be invoked by the "Processing Engine" at runtime of an application.
- **StateUtils class –**

#### *AbstractDynamicMenu class*

Abstract implementation of the dynamic menu state.

#### *Syntax*

```
public class AbstractDynamicMenu
```

#### *Remarks*

Typically extended by the states that needs to return a list of items. The list is sent to the mobile handset as Send SMS, and in the form of a menu, as shown on the following example.

```
    Transactions:
    1. 26 Jan 2012 09:16 - USD 10.00 Pay cab
    2. 26 Jan 2012 10:10 - USD 3.45  Starbucks
    3. 26 Jan 2012 12:25 - USD 20.00 Lunch
    4. 26 Jan 2012 13:30 - USD 3.00 Starbucks
    9: More
    0: Exit
```

The menu item can be selected by sending reply with the menu item number. Example, 1, 2, 3, or 4, to select the item; 9 will be shown when they are more items and selecting 9 will show them, 0 is for exiting from the menu and proceed to the alternative flow. Typically, when the menu item is selected, a more detailed information of the selected item will be sent as SMS.

If there are more items then those displayed, the *paging* menu item will be displayed. For the above example, it is `"9: More"`. When 9 is sent back as the reply, the menu will refreshed showing the next page of the menu, and so on. The menu index is always starting from 1. When the menu reaches the end, and the 9 is sent back as a reply again, the menu will rotate back to the first page.

The state can be configured to show the *exit* menu item. In the above case, it's the `"0: Exit"`. When 0 is sent back as a reply, nothing is selected so the state will continue the transition that is associated with the dynamic "EXIT" value. In some application flows, when a selection is required, the *exit* menu item can be suppressed by setting "No" to the "Show Exit menu" drop down box in the state editor.

Three `Attribute` are automatically registered by this class, including:

- SelectionBoxAttribute - to specify whether to Show Exit menu item
- OutputAttribute - to specify the variable name to store the selected menu KEY
- OutputAttribute - to specify the variable name to store the selected menu VALUE

The life cycle of AbstractDynamicMenu is based on the life cycle of the SmappStatePlugin and StatePlugin with the following customizations to meet the menu functionalities.

`AbstractDynamicMenu` is just an abstract implementation of a state that can Send SMS in the form menu. The subclass provides implementation of getting the list for the menu. The processMessageLogic(SmappStateProcessingContext) and processStateLogic(SmappStateProcessingContext, SmappStateProcessingAction) methods have been final because they have the implementation for the menu functionalities. The subclass need to implement the following abstract methods instead: init(SmappStateProcessingAction), constructMenuList(), getStateAttributeList(), and saveSessionVariables(String, String).

- init(SmappStateProcessingAction) method will be called by the processStateLogic(SmappStateProcessingContext, SmappStateProcessingAction).Please refer to the method description for details.
- constructMenuList() and saveSessionVariables(String, String) methods will be called by both the processMessageLogic(SmappStateProcessingContext) and processStateLogic(SmappStateProcessingContext, SmappStateProcessingAction) methods.Please refer to the method description for details.
- getStateAttributeList() is called from the getStateAttributes() that will aggregate the attributes returned by the getStateAttributeList() with some attributes defined in this abstract class for the menu, such as the input attribute for specifying whether to show the exit menu item, the output attributes for the key and value of the selected menu item.

- The `AbstractDynamicMenu` state will initially be activated by the follow-up transition from a previous state, so the processing engine will call the processStateLogic(SmappStateProcessingContext, SmappStateProcessingAction) method. The `init` and `constructMenuList` method will be called sequentially to initialize and construct the menu. Eventually the menu will be sent out as SMS message, and the processing engine will be set to `action.waitForMessage()` waiting for the reply message. In this case, the user selects a menu item.
- In the special case when the `constructMenuList` returns a single item only, the state will immediately call the saveSessionVariables(String, String) method proceed with the the default `continueDyn(1)` follow-up transition. The state can customized the default behavior by overriding the continueWhenSingleEntry(SmappState) method.
- When the reply message arrives, the processing engine will trigger the `AbstractDynamicMenu` state using the external activation, hence calling the processMessageLogic(SmappStateProcessingContext) method.The `constructMenuList` method will be called again to assemble the menu that will be used to interpret the user selected menu index. If the selection is one of the valid menu item, saveSessionVariables(String, String) method will be called allowing the state to prepare all the select item details for output, and proceed with the follow-up transition as returned by the `saveSessionVariables` method. If null is null, the default OK follow-up transition will be used.

*Pagination class*

Helper class that transforms list of menu items into the list of Page classes.

*Syntax*

`private class  Pagination`

*Remarks*

Each page contains the maximum number of menu items as returned by the getMaxMenuItems() method.

*Pagination(List< KeyValuePair< String, String >>, String, boolean) constructor*

**Syntax**

`Pagination ( List< KeyValuePair< String, String >>` *list* `,  String` *pageHeader* `,  boolean` *showExitMenu* `)`

**Parameters**

- **list –** The list of items
- **pageHeader –** The header of each page
- **showExitMenu –** whether to show the Exit menu item

### *getPages() method*

**Syntax**

```
List< Page > getPages ()
```

### *hasNext() method*

**Syntax**

```
boolean hasNext ()
```

### *availableChars variable*

*Syntax*

```
int availableChars
```

### *list variable*

*Syntax*

```
List< KeyValuePair< String, String > > list
```

### *pageHeader variable*

*Syntax*

```
String pageHeader
```

### *pages variable*

*Syntax*

```
List< Page > pages
```

### *showExitMenu variable*

*Syntax*

```
boolean showExitMenu
```

### *constructMenuList() method*

Return a list of menu item that will be used to construct the SMS menu message.

**Syntax**

```
abstract List< KeyValuePair< String, String > >
constructMenuList () throws DBException
```

**Returns**
a list of KeyValuePair menu item

**Exceptions**

- **DBException –** Exception when accessing or saving the session variable from database

**Usage**

The menu item type is

KeyValuePair

. This method allows the extending class to provide a list of menu items to be used by the

`processMessageLogic(SmappStateProcessingContext)`

and

`processStateLogic(SmappStateProcessingAction)`

.

a list of KeyValuePair menu item

### *continueWhenSingleEntry(SmappState) method*
When the menu contains a single item, the state will automatically proceed to the continueDyn(1) transition by default.

**Syntax**
`SmappState continueWhenSingleEntry ( SmappState` *continueState* `)` throws `DBException`

**Parameters**

- **continueState –** the transition returns by the saveSessionVariables() method

**Returns**
the followUp transition when the menu contains a single item

**Exceptions**

- **DBException –** Exception when accessing or saving the session variable from database

**Usage**

This method allows the subclass to override the default behavior when there is a need to automatically proceed with different `continueDyn()`

based on the selected menu item. For example, if the menu presents the wallet entries of various types: SVA, CreditCard, BankAccount, etc. Let's say there is one wallet entry of SVA type. In such case, the implementation can override this method to proceed with, for example,

`continueDyn('SVA')`

instead of the default

`continueDyn(1)`

.

This method is called immediately after

`saveSessionVariables(String, String)`

. The input

`SmappState`

parameter is that returned by the

`saveSessionVariables(String, String)`

. If the

saveSessionVariables(String, String)

already returns the correct

`SmappState`

, or

`continueDyn('SVA')`

for the above example, then this method implementation can just return the input parameter (i.e., return continueState;).

the followUp transition when the menu contains a single item

### *createPage(boolean) method*

Provide subclasses the option to extend or override the default Page functionality.

#### Syntax

`Page createPage ( boolean` *showExitMenu* `)`

#### Parameters

- **showExitMenu –** Whether to show the exit menu on the page.

**Returns**
A new Page.

**Usage**

A new Page.

*getLineBreak() method*
Line break characters are used in the SMS menu message.

**Syntax**
`String getLineBreak ()`

**Returns**
the current line break characters

**Usage**

This method returns the line break characters to be used in the SMS menu message. By default, it is set to "\n" characters. Some operators support different characters.

Subclass may override it with other supported characters, or use settings obtained from the configuration file.

the current line break characters

*getMaxMenuItems() method*
Maximum number of menu items to display or send as SMS menu message including the pagination "9: More" and "0: Exit" items.

**Syntax**
`int getMaxMenuItems ()`

**Returns**
maximum number of menu items

**Usage**

By default, it is set to 8.

Subclass may override with setting according to the size of each menu item.

maximum number of menu items

### *getMenuListFromSession(BeanConverterInterface< T >) method*
Get the menu list.

#### Syntax
`final< T > List< T > getMenuListFromSession`
`( BeanConverterInterface< T >` *menuBean* `)` throws `DBException`

#### Parameters

- **menuBean –** object type holding the menu information

#### Returns
menu list of `menuBean` type

#### Exceptions

- **DBException –** Exception when accessing or saving the session variable from database

#### Usage

This method is typically called from the

constructMenuList()

and

saveSessionVariables(String, String)

methods.

menu list of `menuBean` type

### *getMessageOptions(Page) method*
Provide subclasses the option to set some default message options based on the current page before sending the message.

#### Syntax
`MwizMessageOptions getMessageOptions ( Page` *page* `)`

#### Returns
Default message options if any.

#### Usage

Default message options if any.

*getPaginationExit() method*
String showing: Menu index and description, for exit from the menu.

**Syntax**
`String getPaginationExit ()`

**Returns**
pagination exit characters

**Usage**

By default, it is set to "<code>\n0: Exit</code>".

Subclass may override with the preferred text or language specific content, or override it with settings obtained from the configuration file. NOTE: the menu index for exit is

FIXED to 0

, and cannot be changed.

Note the state editor will display an input dropdown attribute called "Show Exit menu". This is used for selecting whether to include the exit menu in the SMS menu message. This is used in cases where the exit menu is not be allowed, forcing the user to select a menu item.

pagination exit characters

*getPaginationNext() method*
String showing: Menu index and description, for paging the menu, or pagination to the next page.

**Syntax**
`String getPaginationNext ()`

**Returns**
pagination next characters

**Usage**

By default, it is set to "<code>\n9: More</code>" because the default pagination index is 9, as returned by the

getPaginationNextIndex()

method.

Implementation state may override it to the preferred language specific content, or override it with settings obtained from the configuration file.

NOTE

:

- Please ensure that the pagination index string returned by this method is in-synch with the number returned by the getPaginationNextIndex() method.
- The pagination index can be treated separately from the maximum number of menu item. In other words, the maximum number of menu item can be set to 4, for example, and the pagination index is kept as default (i.e., 9).

pagination next characters

### *getPaginationNextIndex() method*

Index recognized by the algorithm for pagination to the next page.

#### Syntax

```
int getPaginationNextIndex ()
```

#### Returns

pagination index number

#### Usage

```
By default, this is set to 9. The default maximum number of menu
items is set to 8 (including pagination and possible exit index). So,
there is a gap of 7 and 8. By default, the full menu will be displayed
as follow.

        1: Item1
        2: Item2
        3: Item3
        4: Item4
        5: Item5
        6: Item6
        9: More
        0: Exit
```

Implementation state may override this to a different number, if needed.

NOTE

:

- The pagination index number returned by this method is in synch with the pagination index string returned by the getPaginationNext() method.
- The pagination index can be treated separately from the maximum number of menu items. In other words, the maximum number of menu item can be set to 4, for example, and the pagination index is kept as default (i.e., 9). However, if the maximum number of menu items (getMaxMenuItems()) is increased to greater than 10, then the pagination index number and text needs to be adjusted to a larger number accordingly.

pagination index number

### *getStateAttributeList() method*

Return an array of Attribute specified in the subclass.

**Syntax**

```
abstract Attribute[] getStateAttributeList ()
```

**Returns**

an array of Attribute specified by the subclass

**Usage**

This method will be used in constructing the array returned by the

`getStateAttributes()`

method. The array returns the aggregated attributes of this abstract class and the subclass.

an array of Attribute specified by the subclass

### *getStateAttributes() method*

Returns the complete set of Attribute as specificed by this class and the subclass.

**Syntax**

```
final Attribute[] getStateAttributes ()
```

**Returns**

an array of `Attribute` including those registered by the subclass

**Usage**

Intentionally made final so that it can not be override.

The subclass should use the

getStateAttributeList()

to return all the

`Attribute`

s it specified, so that it can be aggregated by this method.

an array of `Attribute` including those registered by the subclass

### *getStateNotes() method*

Notes on the default behaviour of the dynamic menu.

**Syntax**

`String getStateNotes ()`

**Usage**

The text is as follow:

```
Use the following follow up states:
OK: If user selected a menu itemFAIL: Unexpected error occurredDyn
EXIT: User selected to exit the menuDyn 1: Menu contains only 1
items, so it's auto selectedDyn 0: Menu contains 0 items, so skipped


This notes can be appended to the subclass notes. For example,
the subclass override the method and embedded the default notes






    public String getStateNotes() {
        StringBuilder sb = new StringBuilder();
        sb.append("State details.\n\n");
        sb.append(super.getStateNotes());
        sb.append("- Dyn -9: Missing required input, or invalid
format.\n");
        return sb.toString();
    }
```

### *init(SmappStateProcessingAction) method*

Allow the subclass to perform all the necessary initialization including constructing the menu list.

**Syntax**

`abstract SmappState init ( SmappStateProcessingAction` *action* `)` throws `DBException`

**Parameters**

- **action –** Processing action object, is used as a mechanism to communicate to the processing engine the desired processing action

**Returns**

follow-up transition, if necessary. In the normal circumstances, `null` will be returned. Otherwise, the processing will be interrupted by this follow-up transition.

### Exceptions

- **DBException –** Exception when accessing or saving the session variable from database

### Usage

Typically initialization includes: getting the menu list from the datasource, such as web service.

follow-up transition, if necessary. In the normal circumstances, `null` will be returned. Otherwise, the processing will be interrupted by this follow-up transition.

*processMessageLogic(SmappStateProcessingContext) method*
Process incoming message.

### Syntax

`final SmappState processMessageLogic`
`( SmappStateProcessingContext` *context* `)` throws
`MwizProcessingException, DBException`

### Parameters

- **context –** processing engine context; should be used to obtain context information but should NOT be modified.

### Returns

follow-up transition

### Exceptions

- **MwizProcessingException –** processing engine exception
- **DBException –** Exception when accessing or saving the session variable from database

### Usage

The

AbstractDynamicMenu

support "Send SMS" feature, meaning it can send message and wait for the response, a very similar behavior to the base "Send SMS" state. This

processMessageLogic(SmappStateProcessingContext)

method will be called during processing of the incoming message. This method processes the incoming message, most likely, containing the selected menu item. The message can be either a selection or a menu control (such as: more or exit).

If the incoming message contains a valid selection of menu item index, the

saveSessionVariables(String, String)

method will be called followed by

continueOk()

.

If the incoming message conatins a valid menu control index, the processing is forwarded to

processStateLogic(SmappStateProcessingContext, SmappStateProcessingAction)

method.

Intentionally made final so that it cannot be override.

follow-up transition

### *processStateLogic(SmappStateProcessingContext, SmappStateProcessingAction) method*

Contains the actual processing logic for constructing the menu list.

#### Syntax

```
final SmappState processStateLogic
( SmappStateProcessingContext context ,
SmappStateProcessingAction action ) throws MwizProcessingException,
DBException
```

#### Parameters

- **context –** processing engine context; should be used to obtain context information but should NOT be modified.

#### Returns

follow-up transition

#### Exceptions

- **MwizProcessingException –** processing engine exception
- **DBException –** Exception when accessing or saving the session variable from database

#### Usage

The following methods will be called:

- init(SmappStateProcessingAction) - for initialization and getting the menu items from the source

- constructMenuList() - for getting the formatted menu list in key-value pair form

If the

constructMenuList()

returns a list containing a single item only, the

saveSessionVariables(String, String)

will be called and the method will return with the follow-up transition (

`SmappState`

) from the

continueWhenSingleEntry(SmappState)

. Otherwise, the constructed SMS menu message will be sent out.

Intentionally made final so that it cannot be override.

follow-up transition

### *saveSessionVariables(String, String) method*

This method is called after a selection is made from the menu, allowing the subclass to prepare the OutputAttributes (if needed) before transition to the follow-up state.

#### Syntax

`abstract SmappState saveSessionVariables ( String` *key* `, String` *value* `)` throws `DBException`

#### Parameters

- **key –** Selected menu item key
- **value –** Selected menu item value

#### Returns

`null` - correspond to everything is OK, proceed as normal `SmappState` - other than OK, and it needs to be handle accordingly.

#### Exceptions

- **DBException –** Exception when accessing or saving the session variable from database

#### Usage

When the returned value is

`null`

, it will be treated as

continueOk()

. Otherwise, it will be used as the follow-up transition.

In addition, this method is also called in the special cases when the menu contains a single item only. Instead of displaying the menu, the state will immediately transition to the

*single-item*

follow-up state. This single-item follow-up state is whatever

`SmappState`

returned by the

continueWhenSingleEntry(SmappState)

method. By default, it is

`continueDyn(1)`

. See

continueWhenSingleEntry(SmappState)

for details. The

`SmappState`

returned by this method will be submitted to the

continueWhenSingleEntry(SmappState)

method.

It's possible to make additional service called within this method to gather additional information based on the selected item. For example, if the menu presents the transaction list, a call to get transaction details based on the selected transaction item can be done.

`null` - correspond to everything is OK, proceed as normal `SmappState` - other than OK, and it needs to be handle accordingly.

### *setMenuListToSession(List< T >) method*

Set a new menu list and save it into the session attribute.

#### Syntax

`final< T extends BeanConverterInterface< T > > void setMenuListToSession ( List< T >` *list* `)` throws `DBException`

#### Parameters

- **list –** menu list to be persisted to session attribute

#### Exceptions

- **DBException –** Exception when accessing or saving the session variable from database

#### Usage

Menu list is retrieved from the datasource in the

init(SmappStateProcessingAction)

method that is implemented by the subclass. The menu list needs to be persisted into session attribute so that it can exist over multiple SMS sessions without reloading it from the source (database or service). To persist the menu list using this method, the object in the list has to implement

BeanConverterInterface

and

Comparable

.

This method is typically called from the

init(SmappStateProcessingAction)

method.

### *supportsFailTransition() method*

Support fail transition.

#### Syntax

```
final boolean supportsFailTransition ()
```

#### Returns

true, or the transition fail supported

#### Usage

This is always true to force subclass in handling fail condition.

true, or the transition fail supported

### *supportsOkTransition() method*

Support OK transition.

#### Syntax

```
boolean supportsOkTransition ()
```

#### Returns

whether transition OK is supported or not

#### Usage

If true, the OK transition drop down list will be displayed in the state editor. The drop down list contains all the valid follow-up states. By default, this method returns

```
true
```

Subclass has the option to turn it off by override this method. Let's say, for example, the menu displays the wallet entries list consisting of various types of payment instruments (PI). When a menu item is selected, it will be convenient to know the selected PI type as well so that the follow-up state can process it accordingly. For example, if the selected PI is SVA then the transition will be wired to the get SVA details follow-up state.

The above example can be accomplished by overide this method to false, so that OK transition is not displayed on the state editor. In the subclass implementation, the OK transition is supported using the

```
continueDyn()
```

instead. For example, if the PI is SVA, the state will return

```
continueDyn("SVA")
```

, etc.

<u>NOTE</u>: if subclass override this method, most likely, it should also override the

continueWhenSingleEntry(SmappState)

method.

whether transition OK is supported or not

### *supportsSendSmsMessage() method*

Support Send SMS message textbox toggle.

#### Syntax

```
boolean supportsSendSmsMessage ()
```

**Usage**

If true, the textbox will be displayed on the state editor. The content of the textbox will be used as the header in the SMS menu message. For example, if the content of the textbox is "Transactions:" then the menu will be shown as follow.

```
    Transactions:
    1. 26 Jan 2012 09:16 - USD 10.00 Pay cab
    2. 26 Jan 2012 10:10 - USD 3.45  Starbucks
    3. 26 Jan 2012 12:25 - USD 20.00 Lunch
    4. 26 Jan 2012 13:30 - USD 3.00 Starbucks
    9: More
    0: Exit
```

Subclass has the option to turn it off by override this method.

*inShowExitMenu variable*
Selection for whether to show the *exit* menu item.

*Syntax*
`final SelectionBoxAttribute inShowExitMenu`

*Remarks*
Default Id = "SHOW_EXIT"Default description = "Show Exit menu"Default entries are: [0, No], [1,Yes]

The subclass may modify the key and description as needed.

*outKey variable*
Selected key.

*Syntax*
`final OutputAttribute outKey`

*Remarks*
When a menu item is selected, the selected key will be stored in this session OutputAttribute.

Default Id = "SELECTED_KEY"Default description = "Variable name of the selected key"

The subclass may modify the key and description as needed.

*outValue variable*
Selected value.

*Syntax*
`final OutputAttribute outValue`

*Remarks*
When a menu item is selected, the selected value will be stored in this session OutputAttribute.

Default Id = "SELECTED_VALUE"Default description = "Variable name of the selected value"

The subclass may modify the key and description as needed.

*Page class*
Page class represents a one page of menu items based on the specified maximum number of items per page.

*Syntax*
`public class  Page`

*Remarks*
Used in the AbstractDynamicMenu implementation of Send SMS menu states.

*Page(boolean) constructor*
Constructor.

**Syntax**
`Page ( boolean` *showExitMenu* `)`

**Parameters**

- **showExitMenu –** True/False on whether the menu will display the Exit item

*getHeader() method*
Get the header text to be included with the menu items.

**Syntax**
`String getHeader ()`

**Returns**
menu page header

**Usage**

```
For example, the header text "Transactions:" will appear in the
following menu:

    Transactions:
    1. 26 Jan 2012 09:16 - USD 10.00 Pay cab
 2. 26 Jan 2012 10:10 - USD 3.45  Starbucks
 3. 26 Jan 2012 12:25 - USD 20.00 Lunch
```

```
4. 26 Jan 2012 13:30 - USD 3.00 Starbucks
9: More
0: Exit
```

menu page header

*getItems() method*
Get the List of KeyValuePair items to be used in generating the menu items.

**Syntax**
`List< KeyValuePair< String, String > > getItems ()`

**Returns**
List of KeyValuePair items for generating menu

**Usage**

List of KeyValuePair items for generating menu

*getLineBreak() method*
Line break characters to be used in the menu.

**Syntax**
`String getLineBreak ()`

**Returns**
the line break string

**Usage**

Some operator support different types of line break characters, so this specified line break characters will be used in the menu.

the line break string

*getPaginationExit() method*
Get the Exit menu item.

**Syntax**
`String getPaginationExit ()`

**Returns**
the exit menu item string

**Usage**

the exit menu item string

*getPaginationNext() method*
Get the pagination NEXT menu item.

**Syntax**
```
String getPaginationNext ()
```

**Returns**
pagination next menu item string

**Usage**

pagination next menu item string

*isExit() method*
True/false for whether to show the Exit menu item or not, respectively.

**Syntax**
```
boolean isExit ()
```

**Returns**
True/false for showing the Exit menu item or not, respectively.

**Usage**

True/false for showing the Exit menu item or not, respectively.

*isNext() method*
True/false for whether to show the pagination Next menu item or not, respectively.

**Syntax**
```
boolean isNext ()
```

**Returns**
True/false for whether to show the pagination Next menu item or not, respectively.

**Usage**

True/false for whether to show the pagination Next menu item or not, respectively.

*previewMenu() method*
String representation of the menu page, but without the pagination menu item.

**Syntax**
`String previewMenu ()`

**Returns**
string representation of menu page without the pagination item

**Usage**

By default, the pagination menu item is "9: More". The exit item will be included if specified.

string representation of menu page without the pagination item

*setExit(boolean) method*
Set True/false for whether to show the Exit menu item or not, respectively.

**Syntax**
`void setExit ( boolean` *exit* `)`

**Parameters**

- **exit –** True/false for showing the Exit menu item or not, respectively.

*setHeader(String) method*
Set the header text to be included with the menu items.

**Syntax**
`void setHeader ( String` *header* `)`

**Parameters**

- **header –** menu page header

**Usage**

```
For example, the header text "Transactions:" will appear in the
following menu:

    Transactions:
    1. 26 Jan 2012 09:16 - USD 10.00 Pay cab
 2. 26 Jan 2012 10:10 - USD 3.45  Starbucks
 3. 26 Jan 2012 12:25 - USD 20.00 Lunch
 4. 26 Jan 2012 13:30 - USD 3.00 Starbucks
```

```
9: More
0: Exit
```

*setItems(List< KeyValuePair< String, String >>) method*
Set the List of KeyValuePair items to be used in generating the menu items.

**Syntax**
`void setItems ( List< KeyValuePair< String, String >> items )`

**Parameters**

- **items –** List of KeyValuePair items for generating menu

*setLineBreak(String) method*
Line break characters to be used in the menu.

**Syntax**
`void setLineBreak ( String lineBreak )`

**Parameters**

- **lineBreak –** line break string

**Usage**

Some operator support different types of line break characters, so this specified line break characters will be used in the menu.

*setNext(boolean) method*
Set True/false for whether to show the pagination Next menu item or not, respectively.

**Syntax**
`void setNext ( boolean next )`

**Parameters**

- **next –** True/false for showing the pagination Next item or not, respectively

*setPaginationExit(String) method*
Set the Exit menu item.

**Syntax**
`void setPaginationExit ( String paginationExit )`

**Parameters**

- **paginationExit –** the exit menu item string

**Usage**

```
For example, the following menu has the "0: Exit" as the exit menu
item.

    Transactions:
    1. 26 Jan 2012 09:16 - USD 10.00 Pay cab
 2. 26 Jan 2012 10:10 - USD 3.45  Starbucks
 3. 26 Jan 2012 12:25 - USD 20.00 Lunch
 4. 26 Jan 2012 13:30 - USD 3.00 Starbucks
 9: More
 0: Exit
```

*setPaginationNext(String) method*
Set the pagination NEXT menu item.

**Syntax**
`void setPaginationNext ( String` *paginationNext* `)`

**Parameters**

- **paginationNext –** the pagination next menu item string

**Usage**

```
For example, the following menu has the "9: More" as the pagination
next menu item.

    Transactions:
    1. 26 Jan 2012 09:16 - USD 10.00 Pay cab
 2. 26 Jan 2012 10:10 - USD 3.45  Starbucks
 3. 26 Jan 2012 12:25 - USD 20.00 Lunch
 4. 26 Jan 2012 13:30 - USD 3.00 Starbucks
 9: More
 0: Exit
```

*toString() method*
Complete string representation of the menu page.

**Syntax**
`String toString ()`

**Returns**
string representation of complete menu page without the pagination item

**Usage**

The exit item will be included if specified.

string representation of complete menu page without the pagination item

*exit variable*
True/false for whether to show the Exit menu item or not, respectively.

*Syntax*
`boolean exit`

*header variable*
Header text to be included with the menu items.

*Syntax*
`String header`

*Remarks*

```
For example, the header text "Transactions:" will appear in the
following menu:

    Transactions:
    1. 26 Jan 2012 09:16 - USD 10.00 Pay cab
 2. 26 Jan 2012 10:10 - USD 3.45  Starbucks
 3. 26 Jan 2012 12:25 - USD 20.00 Lunch
 4. 26 Jan 2012 13:30 - USD 3.00 Starbucks
 9: More
 0: Exit
```

*items variable*
List of KeyValuePair items to be used in generating the menu items.

*Syntax*
`List< KeyValuePair< String, String > > items`

*lineBreak variable*
Line break characters to be used in the menu.

*Syntax*
`String lineBreak`

*Remarks*
Some operator support different types of line break characters, so this specified line break characters will be used in the menu.

*next variable*
True/false for whether to show the pagination menu item or not, respectively.

*Syntax*
`boolean next`

*paginationExit variable*
The Exit menu item.

*Syntax*
`String paginationExit`

*Remarks*

```
For example, the following menu has the "0: Exit" as the exit menu
item.

    Transactions:
    1. 26 Jan 2012 09:16 - USD 10.00 Pay cab
 2. 26 Jan 2012 10:10 - USD 3.45  Starbucks
 3. 26 Jan 2012 12:25 - USD 20.00 Lunch
 4. 26 Jan 2012 13:30 - USD 3.00 Starbucks
 9: More
 0: Exit
```

*paginationNext variable*
The pagination NEXT menu item.

*Syntax*
`String paginationNext`

*Remarks*

```
For example, the following menu has the "9: More" as the pagination
next menu item.

    Transactions:
    1. 26 Jan 2012 09:16 - USD 10.00 Pay cab
 2. 26 Jan 2012 10:10 - USD 3.45  Starbucks
 3. 26 Jan 2012 12:25 - USD 20.00 Lunch
 4. 26 Jan 2012 13:30 - USD 3.00 Starbucks
 9: More
 0: Exit
```

### *RequiredParameterMissingException class*

This exception is thrown during runtime in the processing engine when the com.sybase365.mobiliser.brand.plugins.smapp.controls.InputAttribute is designated as 'not optional' or "mandatory" but no value was provided.

#### *Syntax*

```
public class  RequiredParameterMissingException
```

#### *Remarks*

This could also happen, for example, when the input was expected from a session variable but the session variable was not created in the upstream flow.

### *RequiredParameterMissingException(String) constructor*

#### **Syntax**

`RequiredParameterMissingException ( String` *message* `)`

### *SmappStatePlugin class*

This class represents the main class to be inherited by *state* that needs to be displayed on the Application Composer and State Editor, and to be invoked by the "Processing Engine" at runtime of an application.

#### *Syntax*

```
public class  SmappStatePlugin
```

#### *Derived classes*

- *com.sybase365.mobiliser.brand.plugins.smapp.state.AbstractDynamicMenu* on page 203

#### *Remarks*

As such it should be regarded as the 'external' API. Any class that extends this class will be called State for the purpose of the API documentation.

There are a significant number of infrastructure functionalities performed by this base class that simplify the state development, including extra methods to allow processing of input and output attribute values and session variables.

Please refer to StatePlugin for more detailed description of the processing engine life cycle.

*continueDyn(String) method*

Helper method in state development.

**Syntax**

`SmappState continueDyn ( String` *value* `)` throws `DBException`

**Parameters**

- **value –** the value to log.

**Returns**

SmappState of the FAIL follow-up transition

**Exceptions**

- **DBException –** any database exceptions that warrant terminating the application

**Usage**

This method will attempt to find the dynamic follow-up transition that matched the input

String

parameter and return the corresponding follow-up state.

If not found, it will return the follow-up state from the

continueFail()

.

When error occurs, the return will be null.

SmappState of the FAIL follow-up transition

*continueDyn(Integer) method*

Helper method in state development.

**Syntax**

`SmappState continueDyn ( Integer` *value* `)` throws `DBException`

**Parameters**

- **value –** the value to log.

**Returns**

SmappState of the FAIL follow-up transition

#### Exceptions

- **DBException –** any database exceptions that warrant terminating the application

#### Usage

This method will attempt to find the dynamic follow-up transition that matched the input

Integer

parameter and return the corresponding follow-up state.

If not found, it will return the follow-up state from the

continueFail()

.

When error occurs, the return will be null.

SmappState of the FAIL follow-up transition

### *continueDyn(Long) method*

Helper method in state development.

#### Syntax

`SmappState continueDyn ( Long` *value* `)` throws `DBException`

#### Parameters

- **value –** the value to log.

#### Returns

SmappState of the FAIL follow-up transition

#### Exceptions

- **DBException –** any database exceptions that warrant terminating the application

#### Usage

This method will attempt to find the dynamic follow-up transition that matched the input

Long

parameter and return the corresponding follow-up state.

If not found, it will return the follow-up state from the

continueFail()

.

When error occurs, the return will be null.

SmappState of the FAIL follow-up transition

### *continueFail() method*

Helper method in state development.

**Syntax**

`SmappState continueFail ()` throws `DBException`

**Returns**

SmappState of the FAIL follow-up transition

**Exceptions**

- **DBException –** any database exceptions that warrant terminating the application

**Usage**

This method will attempt to find the follow-up transition of FAIL type and return the corresponding follow-up state.

If not found, it will set the processing engine to terminate, or

SmappStateProcessingAction

.terminateProcessing();

SmappState of the FAIL follow-up transition

### *continueFail(String) method*

Helper method in state development.

**Syntax**

`SmappState continueFail ( String` *logMsg* `)` throws `DBException`

**Parameters**

- **logMsg –** the message to be logged

**Returns**

SmappState of the FAIL follow-up transition

**Exceptions**

- **DBException –** any database exceptions that warrant terminating the application

**Usage**

This method will attempt to find the follow-up transition of FAIL type and return the corresponding follow-up state.

If not found, it will set the processing engine to terminate, or

SmappStateProcessingAction

.terminateProcessing();

SmappState of the FAIL follow-up transition

### *continueOk() method*

Helper method in state development.

**Syntax**

`SmappState continueOk ()` throws `DBException`

**Returns**

SmappState of the OK follow-up transition

**Exceptions**

- **DBException –** any database exceptions that warrant terminating the application

**Usage**

This method will attempt to find the follow-up transition of OK type and return the corresponding follow-up state.

If not found, it will set the processing engine to terminate, or

SmappStateProcessingAction

.terminateProcessing();

SmappState of the OK follow-up transition

### *determineFollowingSmappStateFromPattern(SmappStateProcessingContext) method*

Determine the follow-up transition that match the message in the state processing context.

**Syntax**

`SmappState determineFollowingSmappStateFromPattern ( SmappStateProcessingContext` *context* `)` throws `DBException, CryptoException`

**Parameters**

- **context –** the state processing context containing all the necessary object needed while processing the state.

**Returns**
the follow-up transition

**Exceptions**

- **DBException –** any database exceptions that warrant terminating the application
- **CryptoException –** any encryption exceptions that warrant terminating the application

**Usage**

the follow-up transition

*determineFollowingSmappStateFromPattern(SmappStateProcessingContext, MwizMessage) method*
Determine the follow-up transition that match the message from the input parameter.

**Syntax**
`SmappState determineFollowingSmappStateFromPattern ( SmappStateProcessingContext` *context* `, MwizMessage` *msg* `)` throws `DBException, CryptoException`

**Parameters**

- **context –** the state processing context containing all the necessary object needed while processing the state.
- **msg –** message to be used in matching the follow-up transition

**Returns**
the follow-up transition

**Exceptions**

- **DBException –** any database exceptions that warrant terminating the application
- **CryptoException –** any encryption exceptions that warrant terminating the application

**Usage**

the follow-up transition

### *determineFollowingSmappStateFromTransitionType(EnumSmappTransitionType, SmappState, SmappStateProcessingContext) method*

Determine the follow-up transition that match the type provided in the input parameter.

#### Syntax

`SmappState determineFollowingSmappStateFromTransitionType ( EnumSmappTransitionType` *tt* `, SmappState` *state* `, SmappStateProcessingContext` *context* `)` throws `DBException`

#### Parameters

- **tt –** EnumSmappTransitionType type
- **state –** current state
- **context –** the state processing context containing all the necessary object needed while processing the state. the follow-up transition

#### Exceptions

- **DBException –** any database exceptions that warrant terminating the application

### *getCurrentCustomer() method*

Current customer Customer of this session.

#### Syntax

`Customer getCurrentCustomer ()`

#### Returns

current customer or Customer

#### Usage

current customer or Customer

### *getInputAttributes() method*

Return the input attributes specified by the state.

#### Syntax

`List< IAttribute > getInputAttributes ()`

#### Returns

input attribute list

#### Usage

This method is called by the State Editor to get the input attribute list that is used to render the Input Variables on the state editor.

This method could be called by the state implementation logic to gather all the input variable values.

input attribute list

*getOutputAttributes() method*
Return the output attributes specified by the state.

#### Syntax

`List< IAttribute > getOutputAttributes ()`

#### Returns

output attribute list

#### Usage

This method is called by the State Editor to get the output attribute list that is used to render the Output Variables on the state editor.

This method could be called by the state implementation logic to bind all the output variable values, and save to the session variable.

output attribute list

*getSessionAttributeForKey(String) method*
Get the value of session attribute (SessionAttribute) based on the specified key.

#### Syntax

`SessionAttribute getSessionAttributeForKey ( String` *attribKey* `)`
throws `DBException`

#### Parameters

- **attribKey –** session attribute key

#### Returns

the session attribute value

#### Exceptions

- **DBException –** Exception when accessing the session variable from database

**Usage**

the session attribute value

*getSessionAttributes() method*

Get session attributes for the current state in the current session.

**Syntax**

`List< SessionAttribute > getSessionAttributes ()` throws `DBException`

**Returns**

List of SessionAttributes

**Exceptions**

- **DBException –** Exception when accessing the session variable from database

**Usage**

List of SessionAttributes

*getSessionId() method*

Session ID that the application is running.

**Syntax**

`Long getSessionId ()`

*getSmsText18nReplaced() method*

Process the SMS message by replacing all the session variables with the actual session variable value.

**Syntax**

`SmsTextI18n getSmsText18nReplaced ()` throws `DBException, CryptoException`

**Returns**

processed SMS message in SmsTextI18n

**Exceptions**

- **DBException –** Exception when accessing or saving the session variable from database
- **CryptoException –** Encryption exception

**Usage**

processed SMS message in SmsTextI18n

*getStateAttributes() method*
Returns all the attributes (input and output) specified in the state.

**Syntax**
`abstract Attribute[] getStateAttributes ()`

**Returns**
Input and output attributes specified by the state

**Usage**
```
For example, the state implementation can be done as follow:

    // Input Attribute
    private static final TextBoxAttribute inPIId = new
TextBoxAttribute("PI_ID", "Payment Instrument Id", false);
    private static final TextBoxAttribute inOutputDateFormat = new
TextBoxAttribute("OUTPUT_DATE_FORMAT", "Override default output date
format", true);



    // Output Attributes
    private static final OutputAttribute outAcctNumber = new
OutputAttribute("PI_BANKACCT_ACCTNUMBER", "Account number");
    private static final OutputAttribute outHolderName = new
OutputAttribute("PI_BANKACCT_HOLDERNAME", "Account holder name");



    private static Attribute[] stateAttr = new Attribute[] { inPIId,
inOutputDateFormat, outAcctNumber, outHolderName};




    protected Attribute[] getStateAttributes() {
        return stateAttr;
    }
```

Input and output attributes specified by the state

### *getStateNotes() method*
The detailed description about the state, and documentations on how to use the state including the follow-up transitions.

#### Syntax
`String getStateNotes ()`

#### Returns
null by default

#### Usage
Using wicket MultiLineLabel automatic filtering, any newlines '

' will be rendered as

and any double new lines '

' will render as a separate paragraph.

NOTE: Use newline mechanism only for layout of text - do not embed HTML into the text strings

Used by: state editor

null by default

### *handleFatal(SmappStateProcessingContext, SmappStateProcessingAction) method*
Helper method in state development.

#### Syntax
`void handleFatal ( SmappStateProcessingContext` *context* `, SmappStateProcessingAction` *action* `)` throws `MwizProcessingException, DBException`

#### Parameters

- **context –** the state processing context containing all the necessary object needed while processing the state.
- **action –** SmappStateProcessingAction is used to communicate back to the processing engine how to proceed.

#### Exceptions

- **MwizProcessingException –** any exceptions that warrant terminating the application

- **DBException –** any database exceptions that warrant terminating the application

**Usage**

This method will attempt to find the follow-up transition of FAIL type and set the corresponding follow-up state to the

SmappStateProcessingAction

.

If not found, it will set the processing engine to terminate, or

SmappStateProcessingAction

.terminateProcessing();

*isCurrentStateEncrypted() method*
Indicate whether the current state is encrypted.

**Syntax**
`boolean isCurrentStateEncrypted ()`

**Returns**
True or false for encrypted or not, respectively.

**Usage**

This flag can be set in the state editor.

When set to true, a state to never show the messages it sends out or inputs it receives back in clear text in the message logs. This is a security feature to allow passwords and PINs to be restricted.

True or false for encrypted or not, respectively.

*isSelectable() method*
To indicate whether the state can be used as the follow-up state.

**Syntax**
`boolean isSelectable ()`

**Returns**
True - state appears in the State Editor follow-up dropdown list

#### Usage

This is used by the state editor to filter out the follow-up drop down list. Currently, it is only used by the Application state, that is the first state automatically added, and cannot be deleted in any application.

As is now, it's not a very useful method. State should always implement it with a return of true.

Used by: state editor

True - state appears in the State Editor follow-up dropdown list

### *loadStateAttributes(SmappStateEditorContext) method*

Default implementation is empty.

#### Syntax

`void loadStateAttributes ( SmappStateEditorContext` *editorContext* `)`

#### Parameters

- **editorContext –** SmappStateEditorContext is the state editor context containing the datasource

#### Usage

Plugin implementations should override this method to perform any dynamic loading of state attributes for display in the state editor. This method is called by the state editor only, and not used by the processing engine.

### *processMessage(SmappStateProcessingContext) method*

This method is called by the processing engine when the state is activated from external source, for example, incoming SMS message.

#### Syntax

`SmappState processMessage ( SmappStateProcessingContext` *context* `)`
throws `MwizProcessingException, DBException, CryptoException`

#### Usage

See this API documentations for the processing engine life cycles.

Typically, this method should be implemented when the

supportsSendSmsMessage()

is set to true, to handle the reply SMS message.

The reference implementation provided in this method is described as follow. This reference implementation is provided for to help simplifying a typical state development. This implementation is sufficient for most cases.

- Retrieve the input and output attributes value from the datasource, and bind the value to the input and output attributes. Note: these attributes are persisted to the datasource as a session variable.
- Delegate the call to the subclass method for processing the state logic, processMessageLogic(SmappStateProcessingContext).
- The delegate method, processMessageLogic(SmappStateProcessingContext) returns the follow-up transition as SmappState object.
- If the follow-up transition (SmappState) returned by the `processMessageLogic()` is not null, proceed to return the same object to the processing engine. Otherwise, if the returned follow-up transition is null, then try to find the matched dynamic follow-up transition, and return that to the processing engine. NOTE: if a null follow-up transition is returned to the processing engine, the engine will terminate the flow.

  Used by: processing enginecontextthe state processing context containing all the necessary object needed while processing the state.the follow-up transitionMwizProcessingExceptionany exceptions that warrant terminating the applicationDBExceptionany database exceptions that warrant terminating the applicationCryptoExceptionany encryption exceptions that warrant terminating the application

*processMessageLogic(SmappStateProcessingContext) method*
Delegate method call by the processMessage(SmappStateProcessingContext).

## Syntax

`SmappState` `processMessageLogic` `(` `SmappStateProcessingContext` *context* `)` throws `MwizProcessingException, DBException`

## Parameters

- **context –** the state processing context containing all the necessary object needed while processing the state.

## Returns

null follow-up transition hat will result in application termination

## Exceptions

- **MwizProcessingException –** any exceptions that warrant terminating the application
- **DBException –** any database exceptions that warrant terminating the application

### Usage

See

processMessage(SmappStateProcessingContext)

for details.

This default implementation returns null, so it will

null follow-up transition hat will result in application termination

#### *processState(SmappStateProcessingContext, SmappStateProcessingAction) method*

This method is always called by the processing engine when the state is activated by the follow-up transition.

### Syntax

`void processState ( SmappStateProcessingContext` *context* `, SmappStateProcessingAction` *action* `)` throws `MwizProcessingException, DBException`

### Usage

The reference implementation provided in this method is described as follow. This reference implementation is provided for to help simplifying a typical state development. This implementation is sufficient for most cases.

- Retrieve the input and output attributes value from the datasource, and bind the value to the input and output attributes. Note: these attributes are persisted to the datasource as a session variable.
- Delegate the call to the subclass method for processing the state logic, processStateLogic(SmappStateProcessingContext, SmappStateProcessingAction).
- The delegate method, processStateLogic(SmappStateProcessingContext, SmappStateProcessingAction) returns the follow-up transition as SmappState object.
- If the follow-up transition (SmappState) returned by the `processStateLogic()` is not null, communicate the follow-up transition to the processing engine using the `SmappStateProcessingAction` object, as follow. action.continueProcessing(followUpTransition); Otherwise, if the returned follow-up transition is null, then the processing engine will terminate the application. NOTE: if a `SmappStateProcessingAction` object is not set the engine will terminate the flow.

  Used by: processing enginecontextthe state processing context containing all the necessary object needed while processing the state.actionmechanism to communicate suggested follow-up action to the processing engineMwizProcessingExceptionany exceptions that warrant terminating the applicationDBExceptionany database exceptions that warrant terminating the applicationCryptoExceptionany encryption exceptions that warrant terminating the application

### *processStateLogic(SmappStateProcessingContext, SmappStateProcessingAction) method*

Delegate method call by the processState(SmappStateProcessingContext, SmappStateProcessingAction).

#### Syntax

`abstract SmappState processStateLogic ( SmappStateProcessingContext` *context* `, SmappStateProcessingAction` *action* `)` throws `MwizProcessingException, DBException`

#### Parameters

- **action –** SmappStateProcessingAction is used to communicate back to the processing engine how to proceed.

#### Exceptions

- **MwizProcessingException –** Any exceptions that warrant terminating the application
- **DBException –** Severe database exceptions that warrant terminating the application

#### Usage

See

processState(SmappStateProcessingContext, SmappStateProcessingAction)

for details.

Helper methods are used to provide the follow-up transition, as follows:

- continueOk()
- continueFail()
- continueFail(String)
- continueDyn(String)
- continueDyn(Integer)
- continueDyn(Long)

```
If the SmappStateProcessingAction is used to communicate back to the
processing engine on how to proceed, the return SmappState should be
set to null. Otherwise, the returned SmappState (i.e.,
followUpTransition) will be used by in the
processState(SmappStateProcessingContext,
SmappStateProcessingAction) using the following:
    action.continueProcessing(followUpTransition);
```

### *saveOutputAttributes() method*

Saving attributes that has holdValue (see OutputAttribute#getHoldValue()) in bulk for better performance.

#### Syntax

`void saveOutputAttributes ()` throws `DBException`

#### Exceptions

- **DBException –** Exception while accessing or saving the session variable from database

### *saveSessionAttribute(String, String) method*

Save the input parameters to the session attribute (SessionAttribute).

#### Syntax

`void saveSessionAttribute ( String` *attribKey* `, String` *attribValue* `)` throws `DBException, CryptoException`

#### Parameters

- **attribKey –** session attribute key
- **attribValue –** session attribute value

#### Exceptions

- **DBException –** Exception while saving the session variable from database
- **CryptoException –** Exception during encryption; The session variable can be encrypted prior to saving, but not supported by this method.

#### Usage

For encryption support, see

saveSessionAttribute(String, String)

### *saveSessionAttribute(String, String, boolean) method*

Save the input parameters to the session attribute (SessionAttribute).

#### Syntax

`void saveSessionAttribute ( String` *attribKey* `, String` *attribValue* `, boolean` *encrypt* `)` throws `DBException, CryptoException`

**Parameters**

- **attribKey –** session attribute key
- **attribValue –** session attribute value
- **encrypt –** True/False whether encryption is needed or not, respectively.

**Exceptions**

- **DBException –** Exception while saving the session variable from database
- **CryptoException –** Exception during encryption.

**Usage**

Encrypt the value prior to saving, if needed (encrypt = true).

Also see

saveSessionAttribute(String, String)

method, if encryption is not needed.

*saveSessionAttributes(Map< String, String >) method*
Save the session attributes in the Map of the current state in the current session to database.

**Syntax**
`void saveSessionAttributes ( Map< String, String >` *attrs* `)` throws `DBException, CryptoException`

**Parameters**

- **attrs –** session attributes in Map

**Exceptions**

- **DBException –** Exception while saving the session variable from database
- **CryptoException –** Exception during encryption.

*sendSmappSms(MwizMessage, Language) method*
Send SMS from the current session.

**Syntax**
`void sendSmappSms ( MwizMessage` *smstext* `, Language` *selLang* `)`

**Parameters**

- **smstext –** SMS message to send out

- **selLang –** No longer supported; Legacy backward compatibility only

*sendSmappSms(MwizMessage, Language, MwizMessageOptions) method*
Send SMS from the current session.

**Syntax**
`void sendSmappSms ( MwizMessage` *smstext* `, Language` *selLang* `, MwizMessageOptions` *msgOptions* `)`

**Parameters**

- **smstext –** SMS message to send out
- **selLang –** No longer supported; Legacy backward compatibility only
- **msgOptions –** Optionally allow caller to set default message options.

*setAckMessage(boolean) method*
Set the SMPP acknowledgement request flag for the current session.

**Syntax**
`void setAckMessage ( boolean` *ackMessage* `)`

**Parameters**

- **ackMessage –** True or false for whether SMPP acknowledgement is requested or not, respectively

*shutdown() method*
No default implementation.

**Syntax**
`void shutdown ()`

*startup(HashMap< String, String >) method*
No default implementation.

**Syntax**
`void startup ( HashMap< String, String >` *attributes* `)` throws `MwizStartupException`

*supportEncryption() method*
Indicate whether this state support encryption.

**Syntax**
`boolean supportEncryption ()`

**Returns**
False not support encryption by default

**Usage**

This will be used in the State Editor to enable/disable the encryption checkbox.

The state notes should described what is supported. For example, support encryption of session variables before storing in database, or support encryption of before logging in file or database.

False not support encryption by default

*supportsDynTransition() method*
Indicates if the state supports dynamic (Dyn) transition.

**Syntax**
`boolean supportsDynTransition ()`

**Returns**
True - support Dyn transition by default

**Usage**

When set to true the state editor will display the dynamic dropdown UI control listing all the possible states to chose from.

The state may opt to support both OK and dynamic follow-up transitions, with the dynamic follow-up transitions for handling special case or error conditions.

By default is true, so the processing method(s) should have at least one condition that returns

```
continueDyn()
```

.

Used by: state editor

True - support Dyn transition by default

### *supportsFailTransition() method*

Indicate if the state uses the Fail follow-up transition type.

#### Syntax

`boolean supportsFailTransition ()`

#### Returns

False - does not support fail transition by default

#### Usage

When set to true the state editor will display the Fail dropdown UI control listing all the possible states to chose from.

When returning true, the processing method(s) should have at least one condition that returns

```
continueFail()
```

.

The recommended best practice is for states with database or external web service calls to handle errors from these calls using the

```
continueFail()
```

follow-up transition.

Used by: state editor

False - does not support fail transition by default

### *supportsGoToApplication() method*

Indicate if the state supports transfer flow to another application.

#### Syntax

`boolean supportsGoToApplication ()`

#### Returns

False - does not support Goto application by default

#### Usage

When set to true the state editor will display a dropdown UI control containing a list of applications in the workspace that can be "goto".

This is not a very useful method. It's mainly used the base state Goto Application to flag the state editor to display the control. This is not needed by other states because the same mechanism can be accomplished with the base Goto Application state.

Used by: state editor

False - does not support Goto application by default

*supportsOkTransition() method*
Indicate if the state uses the OK follow-up transition type.

**Syntax**
`boolean supportsOkTransition ()`

**Returns**
True - support OK transition by default

**Usage**

When set to true the state editor will display the OK dropdown UI control listing all the possible states to chose from.

By default is true, so the processing method(s) should have at least one condition that returns

`continueOk()`

.

Used by: state editor

True - support OK transition by default

*supportsSendSmsMessage() method*
Indicate if the state may send SMS message to the current consumer.

**Syntax**
`boolean supportsSendSmsMessage ()`

**Returns**
False - does not support send SMS by default

**Usage**

When set to true the state editor will display a textbox UI control for entering the SMS message.

The SMS textbox will support all the functionalities supported in the base Send SMS state, like: the session variable token replacement, truncate message longer than 160 characters and send it as another message, etc.

Set to false by default. When set to true, the state should provide implementation for the

processMessage(SmappStateProcessingContext) method to handle activation from the reply message of this SMS.

Used by: state editor

False - does not support send SMS by default

### *StateUtils class*

*Syntax*

```
public class  StateUtils
```

### *determineFollowingSmappStateFromPattern(SmappStateProcessingContext) method*

#### Syntax

`SmappState determineFollowingSmappStateFromPattern ( SmappStateProcessingContext` *context* `)` throws `DBException, CryptoException`

### *determineFollowingSmappStateFromPattern(SmappStateProcessingContext, MwizMessage) method*

Determine if one of the outgoing transition patterns matches this message.

#### Syntax

`SmappState determineFollowingSmappStateFromPattern ( SmappStateProcessingContext` *context* `,  MwizMessage` *msg* `)` throws `DBException, CryptoException`

#### Usage

Match-variables are set implicitly by this method.

Use normal regex for pattern. The group operator capures parts of the regex to put into a variable.

Sample: msg="ab1234cd"

var="" regex="ab(.*)cd" .. no variable will be set

var="userid" regex="ab(.*)cd" .. variable userid is set to "1234"

var="userid" regex="ab.*cd" .. variable userid is not modified - missing group expression

var="userid,poscode" regex="ab(.*)([cd]+)" .. variable userid is set to "1234", variable poscode to "cd"

var="userid,psocode" regex="ab(.*)cd(.*)" .. variable userid is set to "1234", variable poscode to "" - empty group match

### *determineFollowingSmappStateFromPaymentResult(SmappState, boolean, SmappStateProcessingContext) method*

**Syntax**

`SmappState` `determineFollowingSmappStateFromPaymentResult` `(` `SmappState` *state* `,` `boolean` *paymentSuccess* `,` `SmappStateProcessingContext` *context* `)` throws `DBException`

### *determineFollowingSmappStateFromTransitionType(EnumSmappTransitionType, SmappState, SmappStateProcessingContext) method*

**Syntax**

`SmappState` `determineFollowingSmappStateFromTransitionType` `(` `EnumSmappTransitionType` *tt* `,` `SmappState` *state* `,` `SmappStateProcessingContext` *context* `)` throws `DBException`

### *getSmsText18nReplaced(SmappStateProcessingContext) method*

Process the SMS message by replacing all the session variables with the actual session variable value.

**Syntax**

`SmsTextI18n` `getSmsText18nReplaced` `(` `SmappStateProcessingContext` *context* `)` throws `DBException, CryptoException`

**Parameters**

- **context –** state processing context

**Returns**

processed SMS message in SmsTextI18n

**Exceptions**

- **DBException –** Exception when accessing or saving the session variable from database

**Usage**

processed SMS message in SmsTextI18n

#### *replaceTextLabelsForSessionAttributes(String, SmappStateProcessingContext) method*

**Syntax**

`String` `replaceTextLabelsForSessionAttributes` ( `String` *msgOut* , `SmappStateProcessingContext` *context* ) throws `DBException, CryptoException`

#### *sendSmappSms(SmappStateProcessingContext, MwizMessage, Language, MwizMessageOptions) method*

**Syntax**

`void` `sendSmappSms` ( `SmappStateProcessingContext` *processingContext* , `MwizMessage` *smstext* , `Language` *selLang* , `MwizMessageOptions` *msgOptions* )

### *useful package*

#### *Members*

All public members of the useful package.

- **KeyValuePair< K, V > class –** A simple key-value pair helper class.
- **PhoneNumber class –** The PhoneNumber is mainly used to transform a phone number or MSISDN between national and international notation.

#### *KeyValuePair< K, V > class*

A simple key-value pair helper class.

#### *Syntax*

```
public class  KeyValuePair< K, V >
```

#### *Remarks*

```
Commonly used in creating a selection java.util.List for the
SelectionBoxAttribute. For example,

private static final SelectionBoxAttribute inIncludeInActive =
        new SelectionBoxAttribute("ACTIVE_ID","Include In-Active
[Default: false]", true);


static
{
    inIncludeInActive.getItems().add(new KeyValuePair<String,
String>("true", "True"));
    inIncludeInActive.getItems().add(new KeyValuePair<String,
String>("false", "False"));
}
```

*getKey() method*
Retrieves the key.

**Syntax**
`K getKey ()`

**Returns**
The key.

**Usage**

The key.

*getValue() method*
Retrieves the value.

**Syntax**
`V getValue ()`

**Returns**
The value.

**Usage**

The value.

*KeyValuePair() method*

**Syntax**
`KeyValuePair ()`

**Usage**

Creates an empty key-value pair. Both entries are set to

`null`

.

*KeyValuePair(K, V) method*

**Syntax**
`KeyValuePair ( K` *key* `,  V` *value* `)`

#### Parameters

- **key –** the *key* of the pair
- **value –** the *value* of the pair

#### Usage

Creates a key-value pair with the specified key and value.

### *setKey(K) method*

Overwrites the actual key with the specified one.

#### Syntax

`void setKey ( K` ***key*** `)`

#### Parameters

- **key –** The new key.

### *setValue(V) method*

Overwrites the actual value with the specified one.

#### Syntax

`void setValue ( V` ***value*** `)`

#### Parameters

- **value –** The new value.

### *toString() method*

#### Syntax

`String toString ()`

#### Returns

a `String` representing this instance

#### Usage

Returns a string representation of this instance.

a `String` representing this instance

*PhoneNumber class*

The PhoneNumber is mainly used to transform a phone number or MSISDN between national and international notation.

*Syntax*

```
public class  PhoneNumber
```

*PhoneNumber(String, String) constructor*

**Syntax**

`PhoneNumber ( String` *msisdn* `,  String` *countryCode* `)`

**Parameters**

- **msisdn –** the phone number (not necessarily a mobile phone number); e.g. +491791234567
- **countryCode –** the country code; e.g. 49

**Usage**

Creates a new instance of

PhoneNumber

.

All non numeric chars will be removed.

If country is "1" or United States, Tries to parse the specified MSISDN and falls back to use the specified country code if no internationalised format is recognized.

*PhoneNumber(String) constructor*

Creates a new instance of PhoneNumber using the DEFAULT_COUNTRY_CODE.

**Syntax**

`PhoneNumber ( String` *msisdn* `)`

**Parameters**

- **msisdn –** the phone number (not necessarily a mobile phone number)

**Usage**

Tries to parse the specified MSISDN and falls back to use the

DEFAULT_COUNTRY_CODE

if no internationalised format is recognized.

### *equals(Object) method*

**Syntax**

`boolean equals ( Object o )`

**Parameters**

- **o –** the `Object` to compare with

**Returns**

`true` if `o` is a `PhoneNumber` representing the same MSISDN

**Usage**

Indicates if another object is

*equal to*

this one.

`true` if `o` is a `PhoneNumber` representing the same MSISDN

### *getInternationalFormat() method*

**Syntax**

`String getInternationalFormat ()`

**Returns**

the MSISDN in international format

**Usage**

Returns the phone number in international form (e.g. +491701234567).

the MSISDN in international format

### *getNationalFormat() method*

**Syntax**

`String getNationalFormat ()`

**Returns**

the MSISDN in national format

**Usage**

Returns the phone number in the national form.

An example is 01701234567 for the phone number +491701234567

the MSISDN in national format

### *getNumericInternationalFormat() method*

**Syntax**

```
String getNumericInternationalFormat ()
```

**Returns**

the MSISDN in numeric international format

**Usage**

Returns the phone number in numeric international form (e.g. 00491701234567 for +491701234567).

the MSISDN in numeric international format

### *getShortInternationalFormat() method*

**Syntax**

```
String getShortInternationalFormat ()
```

**Returns**

the MSISDN in truncated international format

**Usage**

Returns the phone number in a truncated international form (e.g. 491701234567 for +491701234567).

the MSISDN in truncated international format

### *hashCode() method*

**Syntax**

```
int hashCode ()
```

**Returns**

a hash value for this instance

#### Usage

Returns a hash value.

a hash value for this instance

### *toString() method*

#### Syntax

```
String toString ()
```

#### Returns

getInternationalFormat()

#### Usage

Returns a string representation of this instance.

getInternationalFormat()

### *DEFAULT_COUNTRY_CODE variable*

A default country code, set to 49.

#### *Syntax*

```
final String DEFAULT_COUNTRY_CODE
```

## template package

#### *Members*

All public members of the template package.

- **SmappTemplateProvider class –** Reference implementation of SmappTemplate.

### *SmappTemplateProvider class*

Reference implementation of SmappTemplate.

#### *Syntax*

```
public class  SmappTemplateProvider
```

#### *Remarks*

See detailed descriptions in the SmappTemplate.

NOTE: this is not an API and should be used as is.

The application flow XML file needs to be packaged inside the bundle when using this provider.

### *getDescription() method*

Get detailed description of the system.

**Syntax**

```
String getDescription ()
```

**Returns**

Detailed description.

**Usage**

This is used by the UI.

Detailed description.

### *getInputStream() method*

Call by the Brand Web UI to get access to the XML containing the application flows.

**Syntax**

```
InputStream getInputStream ()
```

**Returns**

Inputstream containing application flows in XML format.

**Usage**

This is used to load the template into the system.

Inputstream containing application flows in XML format.

### *getName() method*

Get the name of the system.

**Syntax**

```
String getName ()
```

**Returns**

System name.

**Usage**

This is used by the UI.

System name.

### *getResource() method*

Get the location of the XML file containing the application flow relative to the bundle classpath.

#### Syntax

`String getResource ()`

#### Returns

Location of the XML file containing the application flows.

#### Usage

Location of the XML file containing the application flows.

### *getVersion() method*

Get the version of the system.

#### Syntax

`String getVersion ()`

#### Returns

System version.

#### Usage

Information only.

System version.

### *setDescription(String) method*

Set the detailed description of the system.

#### Syntax

`void setDescription ( String` *value* `)`

#### Parameters

- **value –**

#### Usage

Call by Spring during injection.

### *setName(String) method*

Set the name of the system.

**Syntax**

`void setName ( String value )`

**Parameters**

- **value –**

**Usage**

Call by Spring during injection.

### *setResource(String) method*

Set the location of the XML file containing the application flow relative to the bundle classpath.

**Syntax**

`void setResource ( String value )`

**Parameters**

- **value –** Location of the XML file containing the application flows.

**Usage**

For example:

classpath:META-INF/template/money_mobiliser.xml

### *setVersion(String) method*

Set the version of the system.

**Syntax**

`void setVersion ( String value )`

**Parameters**

- **value –**

**Usage**

Can be call by Spring during injection.

*LOG variable*

*Syntax*

```
final Logger LOG
```

# Index