SYBASE®

Reference: Custom Unwired Server Development

Sybase Unwired Platform 1.5.2

DOCUMENT ID: DC01333-01-0152-01
LAST REVISED: July 2010

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

Contents

Introducing Custom Development for Unwired Server 1
Server API 1
Interfaces 2
Javadocs 2
Documentation Road Map for Unwired Platform 3
Result Set Filters 7
Result Set Filter Data Flow 8
Implementing Custom Result Set Filters 8
Writing a Custom Result Set Filter 9
Deploying Custom Filters to Unwired Server 11
Validating Result Set Filter Performance 12
Filter Class Debugging 13
Enabling JPDA 13
Result Checkers 15
Implementing Customized Result Checkers 15
Writing a Custom Result Checker 16
Adding a Result Checker 19
Default Result Checker Code 22
Data Change Notifications 27
Data Change Notification Data Flow 28
Invoking upsert and delete Operations Using Data Change Notification 28
Data Change Notification Requirements 31
Data Change Notification Results 34
Data Change Notification Filters 35
Implementing a Data Change Notification Filter 36
Custom XSLT Transforms 39
Custom XSLT Use Cases 39
Implementing Custom Transforms 39
XSLT Stylesheet Syntax 40

XSLT Stylesheet Example 42
Index 45

Introducing Custom Development for Unwired Server

This developer reference provides information about using the Sybase® Unwired Platform Server API to customize Unwired Server. The audience is advanced developers who are familiar working with APIs, but who may be new to Sybase Unwired Platform.

This guide describes the contents of the Server API, and how you can use packages to create custom server code.

Companion guides include:

- *System Administration of the Unwired Platform*
- *Sybase Unwired WorkSpace online help*
- *Javadocs, which provide a complete reference to the APIs.*

See *Fundamentals* for high-level mobile computing concepts, and a description of how Sybase Unwired Platform implements the concepts in your enterprise.

Server API

Sybase Unwired Platform includes several interfaces that open specific features and functionality of Unwired Server for custom development. Customizing Unwired Server allows you to better control behaviors of these features.

- Result set filter – use a custom Java class to filter the rows or columns of data returned from a read operation for a mobile business object (MBO). You can write a filter to add, delete, or change columns, or to add and delete rows.
  Result set filters depend on the `sup-ds.jar` file, located in the `com.sybase.uep.tooling.api/lib` subdirectory. For example, `C:\Sybase\UnwiredPlatform\Unwired_WorkSpace\Eclipse\sybase_workspace\mobile\eclipse\plugins\com.sybase.uep.tooling.api_<version_and_timestamp>\lib`.
- Result checkers – use the custom Java class to implement custom error checking for enterprise information system (EIS) business objects.
  Result checkers depend on the `sup-ds.jar` file, located in the `com.sybase.uep.tooling.api/lib` subdirectory.
- Data change notifications (DCNs) – use an HTTP interface to immediately propagate EIS changes to Unwired Server, rather than using the built-in cache refresh mechanism configured by the Unwired Server administrator.
  DCN requests contain the changed data (delta) of the EIS for which the Unwired Server cache needs to be updated. This request can be in the standard JSON format that Unwired

Server expects or in a different format along with a translation logic to convert it into the standard format. This translation logic is coded in a DCN Filter.

- Custom transforms – create a transform to modify the structure of generated Web Services message data, so it can be used by an Unwired Platform MBO.

You can program these functions in any order; each class is implemented independently.

Interfaces

There are several Server API interfaces that developers can invoke.

Interface	Includes methods that
com.sybase.uep.eis.ResultSetFilter	Define how a custom filter for the data is called, and perform the filtering of data.
com.sybase.uep.eis.ResultSetFilterMetaData	Obtain output column and datatype information without executing a chain of mobile business object operations and filters with real data.
com.sybase.sup.ws.rest.RestResultChecker	Implement a result checker for a RESTful Web service datasource.
com.sybase.sup.ws.soap.WSResultChecker	Implement a result checker for a SOAP Web service datasource.
com.sybase.sup.sap.SAPResultChecker	Implement a result checker for a SAP® datasource.
com.sybase.sup.server.dcn.DCNFilter	Preprocess – digests the DCN request as blob, converts it into a valid JSON DCN request format and returns the DCN. PostProcess – takes the DCN result in a valid JSON format, converts to the EIS format and returns it.

For details on these classes, and the methods that implement them, see the Javadocs for `com.sybase.sup.admin.client`.

Javadocs

The Server API installation includes Javadocs. Use the Sybase Javadocs for your complete API reference.

As you review the contents of this document, ensure you review the reference details documented in the Javadoc delivered with this API. By default, Javadocs are installed to two different locations:

- For Result Set Filters and Result Checkers: `<UnwiredPlatform_InstallDir>/Servers/UnwiredServer/APIdocs/DS/index.html`
- For Data Change Notifications: `<UnwiredPlatform_InstallDir>/Servers/UnwiredServer/APIdocs/MMS/index.html`.

Documentation Road Map for Unwired Platform

Learn more about Sybase® Unwired Platform documentation.

Table 1. Unwired Platform documentation

Document	Description
Sybase Unwired Platform Installation Guide	Describes how to install or upgrade Sybase Unwired Platform. Check the *Sybase Unwired Platform Release Bulletin* for additional information and corrections. Audience: IT installation team, training team, system administrators involved in planning, and any user installing the system. Use: during the planning and installation phase.
Sybase Unwired Platform Release Bulletin	Provides information about known issues, and updates. The document is updated periodically. Audience: IT installation team, training team, system administrators involved in planning, and any user who needs up-to-date information. Use: during the planning and installation phase, and throughout the product life cycle.
New Features	Describes new or updated features. Audience: all users. Use: any time to learn what is available.
Fundamentals	Describes basic mobility concepts and how Sybase Unwired Platform enables you design mobility solutions. Audience: all users. Use: during the planning and installation phase, or any time for reference.

Document	Description
System Administration	Describes how to plan, configure, manage, and monitor Sybase Unwired Platform. Use with the *Sybase Control Center for Sybase Unwired Platform* online documentation. Audience: installation team, test team, system administrators responsible for managing and monitoring Sybase Unwired Platform, and for provisioning device clients. Use: during the installation phase, implementation phase, and for ongoing operation, maintenance, and administration of Sybase Unwired Platform.
Sybase Control Center for Sybase Unwired Platform	Describes how to use the Sybase Control Center administration console to configure, manage and monitor Sybase Unwired Platform. The online documentation is available when you launch the console (**Start > Sybase > Sybase Control Center**, and select the question mark symbol in the top right quadrant of the screen). Audience: system administrators responsible for managing and monitoring Sybase Unwired Platform, and system administrators responsible for provisioning device clients. Use: for ongoing operation, administration, and maintenance of the system.
Troubleshooting	Provides information for troubleshooting, solving, or reporting problems. Audience: IT staff responsible for keeping Sybase Unwired Platform running, developers, and system administrators. Use: during installation and implementation, development and deployment, and ongoing maintenance.

Document	Description

Getting started tutorials	Tutorials for trying out basic development functionality.
	Audience: new developers, or any interested user.
	Use: after installation.
	• Learn mobile business object (MBO) basics, and create a mobile device application:
	  • *Tutorial: Mobile Business Object Development*
	  • *Tutorial: BlackBerry Application Development using Device Application Designer*
	  • *Tutorial: Windows Mobile Device Application Development using Device Application Designer*
	• Create native mobile device applications:
	  • *Tutorial: BlackBerry Application Development using Custom Development*
	  • *Tutorial: iPhone Application Development using Custom Development*
	  • *Tutorial: Windows Mobile Application Development using Custom Development*
	• Create a mobile workflow package:
	  • *Tutorial: Mobile Workflow Package Development*

Sybase Unwired WorkSpace – Mobile Business Object Development	Online help for developing MBOs. Audience: new and experienced developers. Use: after system installation.
Sybase Unwired WorkSpace – Device Application Development	Online help for developing device applications. Audience: new and experienced developers. Use: after system installation.

Document	Description
Developer references for device application customization	Information for client-side custom coding using the Client Object API. Audience: experienced developers. Use: to custom code client-side applications. • *Developer Reference for BlackBerry* • *Developer Reference for iPhone* • *Developer Reference for Mobile Workflow Packages* • *Developer Reference for Windows and Windows Mobile*
Developer reference for Unwired Server side customization – *Reference: Custom Development for Unwired Server*	Information for custom coding using the Server API. Audience: experienced developers. Use: to customize and automate server-side implementations for device applications, and administration, such as data handling. Dependencies: Use with *Fundamentals* and *Sybase Unwired WorkSpace – Mobile Business Object Development*.
Developer reference for system administration customization – *Reference: Administration APIs*	Information for custom coding using administration APIs. Audience: experienced developers. Use: to customize and automate administration at a coding level. Dependencies: Use with *Fundamentals* and *System Administration*.

Result Set Filters

A result set filter is a custom Java class an experienced developer writes in order to specifically manipulate the rows or columns of data returned from a read operation for an MBO.

When a read operation returns data that does not completely suit the business requirements for your MBO, you can write and add a filter to the MBO to customize the data into the form you need.

You can chain multiple filters together. Multiple filters are processed in the order they are added, each applying an incremental change to the data. Consequently, Sybase recommends that you always preview the results, taking note that the MBO has a different set of attributes than it would have had directly from the read operation. You can map and use the altered attributes in the same way you would a regular attribute from an unfiltered read operation.

Example: a simple SELECT statement filter

Suppose you have an MBO based on this query, and you do not want fname and lname divided between two columns:

```
SELECT * FROM sampledb.customer
```

Instead, write a filter that replaces these columns with a single concatenated "commonName" column.

Note: You could also implement the above example with a more advanced SQL statement with additional computation in the MBO definition:

```
SELECT id, commonName=fname+' '+lname, address, city, state,
zip, phone, company_name FROM customer
```

Example: two separate data sources filter

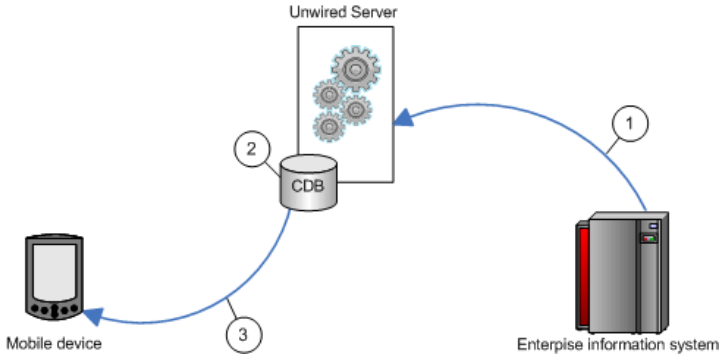
Suppose you have customer data in two data sources: basic customer information is in an SAP repository, and more complete details are contained in another database on your network, for example, SQL Anywhere™. You can use a result set filter to combine the SAP customer data with detailed customer data from the database, so that the MBO displays a complete set of information in a single view. You can accomplish this by:

1. Creating a filter for the SAP backend and add it to an SAP MBO.
2. Add a JDBC connection for the SQL Anywhere backend in the filter, then use the SQL Anywhere data to filter the SAP result.
3. Validate the results are what you expect upon completion. When you synchronize the SAP MBO, you should see data from both SAP backend and SQL Anywhere backend.

Result Set Filter Data Flow

A `ResultSetFilter` is a custom Java class deployed to Unwired Server that manipulates rows and columns of data before synchronization.

Result set filters are more versatile (and more complicated to implement) than an attribute filter implemented through a synchronization parameter, since you must write code that implements the filter, instead of simply mapping a parameter to a column to use as the filter. See *Developers Reference: Server API*.



1. Enterprise information system (EIS) data is sent to Unwired Server.
2. The result set filter filters the results, and applies those results to the CDB for a given MBO. For example, the result set filter combines two columns into one.
3. The device application synchronizes with the results contained in the CDB.

   The client cannot distinguish between MBOs that have had their attributes transformed through a `ResultSetFilter` from those that have not.

Implementing Custom Result Set Filters

Developers can write a filter to add, delete, or change columns as well as to add and delete rows.

Prerequisites

To write a filter, developers must have previous experience with Java programming — particularly with the reference implementations for `javax.sql.RowSet`, which is used to implement the filter interface and described in the *JDBC RowSet Implementations Tutorial*.

Note: Sybase strongly encourages developers to initially create filters in Unwired WorkSpace: a wizard assists you by autogenerating required imports, and methods correctly generated so the implementation already compiles and runs. Then to customize the code, you

can cut and paste fragments from the sample, and make the required changes to get the desired end result.

Once the filter has been implemented and deployed to the server, the mobile business object (MBO) developer can use the filter created from Unwired WorkSpace. See *Sybase Unwired WorkSpace - Mobile Business Object Development > Develop > Developing a Mobile Business Object > Binding Mobile Business Objects to Data Sources > Adding a Result Filter > Deploying Result Filter Classes to Unwired Server*.

Note: Validate the performance of any custom result set filters, before deploying packages to Unwired Server.

1. *Writing a Custom Result Set Filter*

   Write a custom result set filter to define specific application processing logic. Save the compiled Java class file to location that is accessible from Unwired WorkSpace.
2. *Deploying Custom Filters to Unwired Server*

   Deploy custom filters as part of a deployment unit.
3. *Validating Result Set Filter Performance*

   After you deploy the filters to Unwired Server, synchronize data and ensure that filters are performing as you expect.

Writing a Custom Result Set Filter

Write a custom result set filter to define specific application processing logic. Save the compiled Java class file to location that is accessible from Unwired WorkSpace.

In the custom filter, configure attribute properties so that the returned record set can be better consumed by the device client application. Sometimes, a result set returned from a data source requires unique processing; a custom filter can perform that function before the information is downloaded to the client.

Data in the cache is shared by all clients. If you need to identify data in the cache to a specific client, you must define a primary key attribute that identifies the client (such as remote_id or username).

1. (Required) Create a record set filter class that implements the `com.sybase.uep.eis.ResultSetFilter` interface.

   This interface defines how a custom filter for the data is called.

   For example, this code fragment sets the package name and imports the required classes:

   ```
   package com.mycompany.myname;
   import java.sql.ResultSet;
   import java.util.Map;
   ```

2. (Recommended) Implement the `com.sybase.uep.eis.ResultSetFilter` and `com.sybase.uep.eis.ResultSetFilterMetaData` interface on your filter class as required by your business requirements.

   If you choose to implement this interface, you must instead execute a chain of mobile business object operations and filters with real data before you can get actual results of the output columns and their datatypes. This can impact information on the data source, which may eventually need to be reverted. By first implementing these interfaces, the operation does not need to be executed first. Instead, the `getMetaData` obtains the necessary column or data type information.

   This example sets the package name but uses a different combination of classes than in the example for step 1:

   ```
   package com.mycompany.myname;
   import java.sql.ResultSetMetaData;
   import java.util.Map;
   ```

3. Call the appropriate method, which depends on the interfaces you implement.

   `ResultSetFilter` filters the data in the first option documented in step 1. Each filter defines a distinct set of arguments. Therefore, use only the arguments with the appropriate filter that defines these arguments in `getArguments()`, rather than use all filters and data source operations.

   The result set passed in contains the grid data, which should be considered read-only—do not use operations that change or transform data.

   ```
   public interface ResultSetFilter {
      ResultSet filter(ResultSet in, Map<String, Object> arguments)
   throws
        Exception;
      Map <String, Class> getArguments();
   }
   ```

   Next, use `ResultSetFilterMetaData` to format the data from step 1. Use this interface to avoid executing an extraneous data source operation to generate a sample data set.

   ```
   public interface ResultSetFilterMetaData {
     ResultSetMetaData getMetaData(ResultSetMetaData in, Map<String,
        Object> arguments) throws Exception;
   }
   ```

   Note: If the filter returns different columns depending on the argument values supplied, the filter may not work reliably. Ensure that any arguments that affect metadata have constant values in the final mobile business object definition, so the schema does not dynamically change.

4. Implement the class you have created, defining any custom processing logic.

5. Save the classes to an accessible Unwired WorkSpace location. This allows you to select the class, when you configure result set filters for your mobile business object.
6. In Unwired WorkSpace, refresh configured MBO attributes, to see the result.

MBO load operations can take parameters on the enterprise information system (EIS) side. These load parameters are defined from Unwired WorkSpace as you create the MBO. For example, defining an MBO as:

```
SELECT * FROM customer WHERE region = :region
```

results in a load parameter named "region".

As an example, if you want a filter that combines fname and lname into commonName, add `MyCommonNameFilter` to the MBO. When `MyCommonNameFilter.filter()` is called, the "arguments" input value to this method is a `Map<String, Object>` that has an entry with the key "region". Your filter may or may not care about this parameter (it is the backed database that requires the value of region to execute the query). But your filter may need some other information to work properly, for example the remote user's zipcode. The `ResultSetFilter` interface includes `java.util.Map<java.lang.String,java.lang.Class> getArguments()` that you must implement. In order to arrange for the remote user's zipcode (as a String) to be provided to the filter, write some custom code in the body of the `getArguments` method, for example:

```
public Map<String, Object> getArguments {
       HashMap<String, Class> myArgs = new HashMap<String, Class>();
       myArgs.put("zipcode", java.lang.String.class);
       return myArgs;
    }
```

This informs Unwired WorkSpace that the "zipcode" parameter is required, and is of type String. Unwired WorkSpace automatically adds the parameter for the load operation, so this MBO now has two (region and zipcode). Your filter gets them both when its `filter()` method is called, but can ignore region if it wants.

See also

- *Deploying Custom Filters to Unwired Server* on page 11

Deploying Custom Filters to Unwired Server

Deploy custom filters as part of a deployment unit.

There are two methods that are supported.

1. Create a JAR of the class.
2. Deploy the JAR, by packaging it in a deployment unit using either:

- Unwired WorkSpace development tooling. See either *Unwired WorkSpace - Developing Mobile Business Objects > Packaging and Deploying Mobile Business Objects*.
- The Deploy command line utility. See *System Administration of the Unwired Platform > System Reference > Command Line Utilities > Unwired Server Runtime Utilities > Package Administration Utilities > Deploy Application Package (deploy) Utility*.

The packaged classes are copied to `<UnwiredPlatform_InstallDir>\Servers\UnwiredServer\deploy\sup\<deployment-packageName>\lib` by the tool you use. In this case, the deployed package automatically refreshes, so no server restart is required.

See also

- *Writing a Custom Result Set Filter* on page 9
- *Validating Result Set Filter Performance* on page 12

Validating Result Set Filter Performance

After you deploy the filters to Unwired Server, synchronize data and ensure that filters are performing as you expect.

1. Confirm that the columns appear correctly after the filter has been added to the mobile business object.
   a) Refresh the object.
   b) In the Properties view, select the **Attribute Mapping** tab.
   c) Verify that columns are correctly listed in the **Map to** column.
2. From the device client or the device simulator, open the mobile object, and check that the new column appears.
3. Synchronize the object from the device client or simulator.
4. Troubleshoot filters if issues arise:
   - During synchronization, all `System.out` statements are printed to the Unwired Server log.
   - If you started Unwired WorkSpace with the -consoleLog in java.exe, `System.out` statements are also printed to the console window.

See also

- *Deploying Custom Filters to Unwired Server* on page 11

Filter Class Debugging

Sybase Unwired Platform supports various debugging models: instrumented code, and JPDA (Java Platform Debugger Architecture).

You can also instrument code by including **System.out.println()** in the filter class, output from the class is captured in the Unwired Server log when the filter is being executed by the server.

Alternatively, you can use the standard Java debugger to debug the filter class.

Enabling JPDA

To enable JPDA for Unwired Platform debugging, the Unwired Server needs to be started in JPDA mode.

This task describes how to setup JPDA and attache the Java standard debugger to Unwired Server.

Alternatively, you can enable Eclipse debugging in Unwired WorkSpace by first setting up a project and switching to the Debug perspective. Within the filter source code, set breakpoints from the context menu in the default Java editor. Then, with the breakpoints in place, a debugging session can be created. When this is completed, double-click the remote Java application of the Debug Configurations wizard and configure the connection type as:

- use a standard connection (Socket Attach)
- use host 0.0.0.0
- set the the port to matches the one enabled in Unwired Server (by default 5005)

1. Change to the `<UnwiredPlatform_InstallDir>\Servers\UnwiredServer\bin`.
2. By default JPDA connects over port 5005. Change the port by running **djc-setenv.bat** from the same folder and issuing this command:
   ```
   set DJC_JPDA_PORT=5005
   ```
3. Start Unwired Server in JPDA mode . How to do this varies, depending on whether or not Unwired Server is installed as a service:
   - If Unwired Server is not a service, run:
     ```
     start-unwired-server.bat -jpda
     ```
   - If Unwired Server is installed as a Windows service:
     1. Remove the service:
        ```
        sup-server-service.bat remove
        ```
     2. Recreate the service to run in JPDA mode:
        ```
        sup-server-service.bat install -jpda
        ```

4. Once Unwired Server is restarted, verify that JPDA mode is working by running:
   ```
   netstat -an | grep 5005
   ```
   Look for these results:
   ```
   TCP  0.0.0.0:<JPDAport>   0.0.0.0:0  LISTENING
   ```
5. Use a standard Java debugger and attach it to Unwired Server by specifying the correct host and the JPDA port used.

Begin debugging the result filter class with the Java debugger.

Result Checkers

Use the custom Java class to implement custom error checking.

A custom result checker can throw errors for both a scheduled cache refresh as well as an on demand cache refresh:

- For a scheduled refresh – the result checker writes a log message that describes the nature of the error to the Unwired Server log. As a consequence of this error, the transaction for the entire cache group is rolled back. The device client user is not notified of these errors; no client log records are generated.
- On demand refresh – instead of writing the error to the server log, the log message is written to the Unwired Server. Services in the server handle the exception. As a consequence of this error, the transaction for the cache group is rolled back. But in this case, a client log record is generated, which is visible to the client application after synchronization.

Both cases send the `OperationStatusEvent`. This event indicates that an operation failed to execute properly. The server uses `OperationStatusEvent` to populate a statistics repository that tracks the success or failure of EIS operation invocations. An administrator can review these statistics in Sybase Control Center, by clicking the Monitor node in the left navigation pane. See *System Administration of the Unwired Platform > System Maintenance and Monitoring > Status and Performance Monitoring > Reviewing System Monitoring Data*.

Implementing Customized Result Checkers

Implement a custom result checker with the required Java class to implement custom error checking for EIS-specific business objects.

1. *Writing a Custom Result Checker*

   A result checker is a custom Java class that implements error checking for mobile business objects (MBOs).
2. *Adding a Result Checker*

   Add a result checker when you edit Attribute or Operation properties for a mobile business object derived from a data source. Add a result checker after you have either written a custom one or use a predefined one in Unwired WorkSpace (the latter of which can be configured when you create an object).

Writing a Custom Result Checker

A result checker is a custom Java class that implements error checking for mobile business objects (MBOs).

Not all MBO operations use a "standard" error reporting technique; you may want to implement your own custom result checker. Doing so allows you to check any field for errors, or implement logic that determines what constitutes an error, and the severity of the error.

1. Provide a Java class that implements the appropriate interface.

Data source	Interface

SAP	`package com.sybase.sup.sap;`
	`public interface SAPResultChecker`
	`{`
	`/**`
	`*`
	`* @param f - JCO function that has already been executed.`
	`* Use the JCO API to retrieve returned values and determine if the RFC has executed`
	`* successfully.`
	`* @return a single Map.Entry. The boolean "key" value should be set to true if the`
	`* RFC is deemed to have succeeded. Normal result processing will ensue.<P>`
	`* If the String value is not empty/null, that value will be treated as a warning message,`
	`* which will be logged on the server,`
	`* and returned as a warning in transaction logs to the client.<P>`
	`* Set the key value to false if it is deemed the RFC has failed. The String value will`
	`* be thrown in the body of an exception. The error will be logged on the server, and the`
	`* client will receive a transaction log indicating failure, including the string value.`
	`*/`
	`Map.Entry<Boolean, String> checkReturn(JCO.Function f);`
	`}`

Data source	Interface

Web service (SOAP)	`package com.sybase.sup.ws.soap;`
	`public interface WSResultChecker`
	`{`
	`/**`
	`* @param is the method for passing a parameter, and does not support setting a`
	`default value.`
	`* @param response – the SOAP Envelope response from a Web service execute.`
	`* Use the SOAP API to retrieve values and determine if the SOAP request`
	`* has executed successfully.`
	`* @return a single Map.Entry. The boolean "key" value should be set to true if the`
	`* SOAP request is deemed to have succeeded. Normal result processing will ensue.<P>`
	`* If the String value is not empty/null, that value will be treated as a warning message,`
	`* which will be logged on the server,`
	`* and returned as a warning in transaction logs to the client.<P>`
	`* Set the key value to false if it is deemed that SOAP has failed. The String value will`
	`* be thrown in the body of an exception. The error will be logged on the server, and the`
	`* client will receive a transaction log indicating failure, including the string value.`
	`*/`
	`Map.Entry<Boolean, String> checkReturn(javax.xml.soap.SOAPEnvelope response);`
	`}`

Data source	Interface

RESTful Web service	`package com.sybase.sup.ws.rest;`
	
	`import java.util.List;`
	`import java.util.Map;`
	
	`public interface RestResultChecker`
	`{`
	`/**`
	`* REST Result Checker.`
	`*`
	`* @param responseBody HTTP response body.`
	`*`
	`* @param responseHeaders HTTP response headers in the form`
	`* {{header1,value1}, {header2,value2}, ...}.`
	`*`
	`* @param httpStatusCode HTTP status code.`
	`*`
	`* @return Single Map.Entry whose boolean "key" value is true if the`
	`* HTTP request succeeded, after which normal result processing will`
	`* ensue.<P>`
	`*`
	`* If the String value is not empty/null, that value will be treated`
	`* as a warning message which will be logged on the server and returned`
	`* as a warning in the transaction log sent to the client.<P>`
	`*`
	`* Set the key value to false if it is deemed that the service has failed.`
	`* The String value will be thrown in the body of an exception. The error`
	`* will be logged on the server, and the client will receive a transaction`
	`* log indicating failure, including the string value.`
	`**/`
	`Map.Entry<Boolean, String> checkReturn( String responseBody,`
	`List<List<String>> responseHeaders, int httpStatusCode );`
	`}`

Result checkers depend on the `sup-ds.jar` file, in `com.sybase.uep.tooling.api/lib` subdirectory. For example, `C:\Sybase\UnwiredPlatform-1_5\Unwired_WorkSpace\Eclipse`

```
\sybase_workspace\mobile\eclipse\plugins
\com.sybase.uep.tooling.api_1.5.0.200909281740\lib
```

2. Save any classes you create to an accessible Unwired WorkSpace location. This allows you to select the class when you configure the result checker for your mobile business object.

See also

- *Adding a Result Checker* on page 19

Adding a Result Checker

Add a result checker when you edit Attribute or Operation properties for a mobile business object derived from a data source. Add a result checker after you have either written a custom one or use a predefined one in Unwired WorkSpace (the latter of which can be configured when you create an object).

1. In the New Attributes or New Operation wizard, in the Result checker section, select from these options:

Option	Description

Default	The result checker depends on the data source type:
	• SAP – `com.sybase.sup.sap.DefaultSAPResultCheck`. If a RETURN parameter is found in the selected operation, this option is automatically selected.
	• Web service (SOAP) – `com.sybase.sup.ws.soap.DefaultWSResultCheck`. The default checker always returns the status as successful.
	`DefaultWSResultCheck Passed.`
	• Web service (RESTful) – `com.sybase.sup.ws.rest.DefaultRestResultCheck`. The default checker always returns the status as successful.
	`DefaultRestResultCheck Passed.`

None	Return the status as successful with no message. The result checker used depends on the data source type: • SAP – `com.sybase.sup.sap.NoOpSAPResultCheck` • Web service (SOAP) – `com.sybase.sup.ws.soap.NoOpWSResultCheck` • Web service (RESTful) – `com.sybase.sup.ws.rest.NoOpRestResultCheck`
Custom	Specify a custom result checker.

2. (Optional) If you have not yet created the result checker classes, select **Custom** in the Result checker area of the New Attributes or New Operation dialog, and click **Create** to run the New Java Class wizard.
3. If prompted, add a Java nature.
   a) (Recommended) Click **Yes** to add a Java nature. In Eclipse, a Java nature adds Java-specific behavior to projects.

   In the New Java Class wizard, enter:

Option	Description
Source folder	By default, this is the `Filters` folder from your project. Click **Browse** to locate the source folder for the Java class.
Package	Click **Browse** to locate the package for the new Java class. **Note:** Sybase recommends that you do not leave this field blank. Otherwise, the JDK 1.4 Java class in the default package cannot be resolved in other packages.
Enclosing type	Choose a type in which to enclose the new class. You can select either this option or the Package option, above. Enter a valid name or click **Browse**.
Name	Enter a name for the result checker class.
Modifiers	Select the Java class modifiers. The default modifier is public.
Superclass	1. Click **Browse**. 2. In the Superclass Selection dialog, enter: • Choose a Type • Matching Items 3. Click **OK**.

Option	Description
Interfaces	By default, this is populated with the corresponding interface: • SAP – `com.sybase.sup.sap.SAPResultChecker` • Web service (SOAP) – `com.sybase.sup.ws.soap.WSResultChecker` • RESTful Web services – `com.sybase.sup.ws.rest.ResultChecker` Click **Add** to select interfaces implemented by the new class.
Which Method Stubs Would You Like to Create	• **Public Static Void Main** • **Constructors From Superclass** • (Default) **Inherited Abstract Methods**
Do You Want to Add Comments	Select **Generate Comments** to add comments. From here, you can modify the preferences of the code templates by clicking **Configure templates and default values**.

b) Click **No** if you do not want to add the Java nature to the selected mobile application project.

c) Click **Finish** to compile the java skeleton source file and add the skeleton Java checker class to the MBO.
The result checker appears next to the Custom option.

4. In the Result checker section, next to the Custom option, click **Browse** to find an existing result checker class.

   a) In the Select Result Checker Class dialog, select the result checker class and click **OK**.

   The result checker class appears next to the Custom option.

5. Validate the result checker:

   a) To reuse input values you have already saved for previous previews, select **Existing Configuration**. Otherwise, load defaults, or create a new set of input values expressly for this preview instance.

   b) Click **Preview**.

   If the data runs successfully, `Execution Succeeded` appears at the top of the Preview dialog and data appears in the **Preview Result** window.

See also

- *Writing a Custom Result Checker* on page 16

Default Result Checker Code

This result checker is a default result checker and is used to check results in SAP data sources.

```
package com.sybase.sap;

import java.util.AbstractMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

import com.sap.mw.jco.JCO;
import com.sybase.sup.sap.SAPResultChecker;
import com.sybase.vader.utils.logging.SybLogger;

public class DefaultSAPResultCheck implements SAPResultChecker
{
    private static Set<String>              nonErrorMessages;
    static
    {
        nonErrorMessages = new HashSet<String>();
        nonErrorMessages.add("No data found");
        nonErrorMessages.add("Data was not found for the document");
    }

    public Map.Entry<Boolean, String> checkReturn(JCO.Function f)
    {
        JCO.Record returnStructure = null;
        JCO.ParameterList jpl = f.getExportParameterList();
        String msg = null;
        boolean success = true;
        if ( jpl != null )
        {
            try
            {
                returnStructure = jpl.getStructure("RETURN");
                if ( returnStructure != null )
                {
                    String type = returnStructure.getString("TYPE");
                    // generally TYPE is S for success, I for
informational,
                    // or empty
                    if ( !(type.equals("") || type.equals("S") ||
type.equals("I")) )
                    {
                        String message =
returnStructure.getString("MESSAGE");
                        /*UWPLogger.LogWarning*/
                      SybLogger.warn("SapUtils.execute: TYPE: <<" +
type + ">>, MESSAGE: <<" + message + ">>");
```

```
                        if ( !type.equals("W") && !
nonErrorMessages.contains(message) )
                        {
                            success = false;
                        msg =  "TYPE: <<" + type + ">>, MESSAGE: <<" +
message + ">>";
                        }
                        else
                        {
                        msg =  "TYPE: <<" + type + ">>, MESSAGE: <<" +
message + ">>";
                        }
                    }
                    else
                    {
                        if (SybLogger.isDebugEnabled())
                        {
                            String message =
returnStructure.getString("MESSAGE");
                        SybLogger.debug("SapUtils.execute: TYPE: <<"
+ type + ">>, MESSAGE: <<" + message + ">>");
                        }
                    }

                }
            }
            catch (Exception e)
            {
                /*
                if (UWPLogger.isTrace())
                    UWPLogger.LogTrace
                */
              SybLogger.debug("SapUtils::execute: Unable to retrieve
RETURN structure - Will try to retrieve RETURN table next.", e);
            }
        }
        // if there is no RETURN structure, look for RETURN table
        if ( returnStructure == null )
        {
            jpl = f.getTableParameterList();
            if ( jpl != null )
            {
                try
                {
                    StringBuilder retMessage = new StringBuilder();
                    JCO.Table returnTable = jpl.getTable("RETURN");
                  for (int i = 0; i < returnTable.getNumRows(); i++)
                    {
                        returnTable.setRow(i);
                        String type = returnTable.getString("TYPE");
                        // generally TYPE is S for success, I for
                        // informational, or empty
                      if ( !(type.equals("") || type.equals("S") ||
type.equals("I")) )
                        {
                            String message =
```

```
returnTable.getString("MESSAGE");
                            /*UWPLogger.LogWarning*/
SybLogger.warn("SapUtils.execute[" + i + "]: TYPE: <<" + type + ">>,
MESSAGE: <<"
                                    + message + ">>");
                            if ( !type.equals("W") && !
nonErrorMessages.contains(message) )
                            {
                                success = false;
                                retMessage
                                    .append("[" + i + "]TYPE: <<" +
type + ">>, MESSAGE: <<" + message + ">>");
                            }
                            else
                            {
                                retMessage
                                    .append("[" + i + "]TYPE: <<" +
type + ">>, MESSAGE: <<" + message + ">>");
                            }
                        }
                        else
                        {
                            if (SybLogger.isDebugEnabled())
                            {
                                String message =
returnTable.getString("MESSAGE");
                               SybLogger.debug("SapUtils.execute[" + i +
"]: TYPE: <<" + type + ">>, MESSAGE: <<" + message + ">>");
                            }
                        }

                    }
                    if( retMessage.length() > 0 )
                    {
                        msg = retMessage.toString();
                    }
                }
                catch (Exception e)
                {
                    /*UWPLogger.LogWarning*/
SybLogger.warn("SapUtils::execute: error in execution while
retrieving RETURN table: ", e);
                    success = false;
                    msg = e.toString();
                }
            }
        }

        return new CheckReturnMapEntry<Boolean, String>(success,
msg);
    }

    class CheckReturnMapEntry<Boolean, String> extends
java.util.AbstractMap.SimpleImmutableEntry<Boolean, String> {

        public CheckReturnMapEntry(Boolean success, String msg) {
```

```
            super(success, msg);
        }
    }
}
```

Data Change Notification Interface

Data change notification (DCN) provides an HTTP interface by which enterprise information system (EIS) changes can be immediately propagated to Unwired Server.

Sybase Unwired Platform provides the `gson-1.4.jar` library you use to construct a DCN URL located in the `<UnwiredPlatform_InstallDir>\Servers\UnwiredServer\lib\ext` directory. All DCN commands support both GET and POST methods. The EIS developer creates and sends a DCN to Unwired Server through HTTP GET or POST operations. The portion of the DCN command parameters that come after `http://host:8000/dcn/DCNServlet`, can all be in POST; any *var=name* can be in either the URL (GET) or in the POST. The HTTP POST method is more secure than HTTP GET methods; therefore, Sybase recommends that you include the authenticate.password parameter in the POST method, as well as any sensitive data provided for attributes and parameters.

Note: Enter the HTTP request on a single line.

You must be familiar with the EIS data source from which the DCN is issued. You can create and send DCNs that are based on:

- Database triggers
- EIS system events
- External integration processes

You can use DCN with payload to instruct Unwired Server to refresh data:

- DCN with payload – calls only the two direct cache-affecting operations (**:upsert** or **:delete**), which always exist for an MBO, and are not related to user-defined MBO operations.
  - :upsert – the message must contain name/value pairs for every required attribute, and the name must exactly match the MBO attribute name.
  - :delete – provide only the name/value pairs for the primary key column(s).

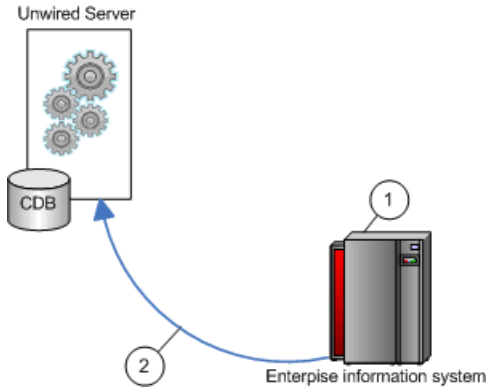
  These operations respectively insert or update, or delete a row in the CDB. Calling either of these operations does not trigger any other refresh action:
  1. Some event initiates the DCN request (a database trigger for example).
  2. The Unwired Server cache is updated directly from the EIS. The actual data (payload) is applied to the cache, through either an **:upsert** (update or insert) or a **:delete** operation.
  3. Unwired Server returns a DCN status message to the requester.

Data Change Notification Data Flow

Data change notifications (DCNs) refresh data when a change to the enterprise information system (EIS) occurs.

DCN requests are sent to Unwired Server as HTTP GET or POST operations. Each DCN can instruct Unwired Server to modify cached MBO data.

A DCN can be invoked by a database trigger, an EIS event, or an external process. DCNs are more complex to implement than other data refresh methods, but ensure that changes are immediately reflected in the cache.



1. An event initiates the DCN.
2. The DCN (HTTP POST or GET) is issued to Unwired Server.

Invoking upsert and delete Operations Using Data Change Notification

Data change notifications (DCNs) with payload directly update the Unwired Server cache, either with the built-in, direct cache-affecting operations **:upsert** (update or insert), or with **:delete**.

Syntax

DCN with payload requires a JavaScript Object Notation (JSON) string (dcn_request) that contains one or more **:upsert** and **:delete** operations that are applied to the Unwired Server cache (CDB).

```
http://unwired_server_host:unwired_server_port(default 8000)/dcn/
DCNServlet
? cmd=dcn
&username=userName
```

```
&password=password
&domain=domainName
&package=unwired_server_PackageName
&dcn_request={"pkg":"dummy","messages":
[{"id":"1","mbo":"CustomerWithParam","op":":upsert","cols":
{"id":"10001","fname":"Adam"}}]}
&dcn_filter=fully_qualified_name_of_dcn_filter
```

Parameters

- **unwired_server –** Unwired Server host name to which the DCN is issued.
- **unwired_server_port –** Unwired Server port number. The default port is 8000.
- **username –** authorized Unwired Server user with permission to modify the MBO.
- **password –** authorized user's password.
- **domain –** Unwired Server domain that contains the package.
- **package –** Unwired Server package that contains the MBO. The format is package:version. For example, e2e_package:1.0.
- **dcn_request –** the JSON string that contains operation name and parameters, which must include:
  - Package name (pkg) – this package name is required to support backwards compatibility but ignored. The package value supplied in the header is the package value used by DCN.
  - A list of messages (messages). Each message includes:
    - A unique message ID (id) used to report back the status. The values provided for the "id" element of each DCN statement within a DCN request message are used only to identify the corresponding status message in the DCN response, which means you can select any value, including nonnumeric characters. Use unique values, so that responses to the correlated requests can be clearly identified.
    - Mobile business object name (mbo).
    - Operation name (op): an operation name of the specified MBO.
    - Bindings (cols): operation parameter names and values covering at least all primary key attributes of the MBO and additionally the required operation parameters.

      Note: For DCN with payload, parameter names must correspond to the attributes of the MBO.
- **dcn_filter –** (optional) the custom filter used to preprocess the DCN request and postprocess the DCN status message any JSON strings. By default, Unwired Server expects all DCN requests to be a valid JSON string. A DCN filter can be used to convert client specific DCN request strings to a valid JSON string as governed by the filter implementation.

- **ppm –** personalization parameters (for either the server or client side) that need to be explicitly defined in the DCN request. The format must conform to the JSON messaging synchronization format, which is a Base64-encoded map of personalization parameters.

Examples

- **Upsert example with header –** this DCN contains a single `:upsert` operation that updates or inserts (upserts) data in the Unwired Server cache for the Department MBO.

```
http://dsqavm5:8000/dcn/DCNServlet?cmd=dcn&username=
supAdmin&password=s3pAdmin&package=dept:
1.0&domain=default&dcn_request=
{"pkg":"dummy","messages":
[{"id":"1","mbo":"Department","op":":upsert",
"cols":{"dept_id":"2","dept_name":"D2","dept_head_id":"501"}}]}
```

- **Upsert example without header –** this JSON string included in a DCN contains a single `:upsert` operation that updates or inserts (upserts) data in the Unwired Server cache for the Department MBO.

```
dcn_request={"pkg":“TestPackage",
"messages":
    [{"id":"1","mbo":"Department",
        "op":":upsert",
        "cols":{"DepartmentID":"3333",
                 "DepartmentName":"Test Value",
                                  "DepartmentHeadID":"501"}}]
}
```

- **Delete example with header –** this DCN example deletes a row of data from the Unwired Server cache for the Department MBO:

```
http://dspevm5:8000/dcn/DCNServlet?cmd=dcn&username=
supAdmin&password=s3pAdmin&package=dept:
1.0&domain=default&dcn_request=
{"pkg":"dummy","messages":
[{"id":"1","mbo":"Department","op":":delete",
"cols":{"dept_id":"2"}}]}
```

- **Delete example without header –** this example JSON string included in the DCN sent to Unwired Server, deletes a row of data from the Unwired Server cache for the Department MBO:

```
dcn_request={"pkg":“TestPackage",
    "messages":[{"id":"1","mbo":"Department",
"op":":delete",
    "cols":{"DepartmentID":"3333"}}]}
```

Usage

Follow these guidelines when constructing a DCN:

- The format of non string data is the same as parameter default values in Unwired WorkSpace. For example, specify timestamp values in a format similar to 2009-03-04T17:03:00+05:30.
- The **:upsert** operation requires:
  - All MBO primary key attributes to be present in the payload.
  - Any other MBO attributes used in the upsert.
  - All columns in the operation use attribute names (not the column names to which they are mapped).
- The **:delete** operation requires:
  - The MBO primary key attribute be present in the payload.
  - All columns in the operation use attribute names (not the column names to which they are mapped).

Data Change Notification Requirements

Use these data change notification (DCN) requirements to familiarize yourself with possible implementation scenarios.

Personalization parameters in DCN

Server and client personalization parameters of the MBO need to be specified separately in the **ppm** parameter. The required ppm parameter in the `dcn_request` has to be a `string` which should be a Base64-encoded map of personalization parameters. This example shows how you must use `ppmString` to define the value for **ppm** parameter in the `dcn_request`:

```
Map<String, String> ppm = new HashMap<String, String>();
        ppm.put("myCompany", "Sybase");
        String ppmString =
Base64Binary.toString(gson.toJson(ppm).getBytes());
```

DCN upsert operations and MBO relationships

When using the DCN payload mode to upsert rows to MBOs where there is a relationship between rows of data, you must provide the data in the correct order so Unwired Server can properly create the metadata in the cache (CDB) to reflect the data relationship. The correct order is to send the upserts for the rows for the child MBO before upserting the related parent rows. However, when you are using DCN to insert data into the cache, the concept of child and parent may be different from what is reflected in the package definition seen in Unwired WorkSpace.

When using DCN to upsert rows to both the parent and child MBOs in a relationship, the order for the upserts can change depending on the nature of the relationship. This is due to the implementation details of the cache metadata. In these examples, the Department MBO is the parent MBO in both relationships, but notice the order of the upsert operations:

- For a one-to-one relationship between:
  ```
  Dept.dept_head_id - > Employee.emp_id
  ```
  (from a department to the department head) the order in which you upsert a new department and new department head is:
  1. Employee
  2. Department

  The foreign surrogate key reference is contained in the Department table in the cache.
- For a one-to-many relationship between:
  ```
  Dept.dept_id - > Employee.dept_id
  ```
  (from a department to all of the employees in the department) the order in which you upsert a new department and a new employee is:
  1. Department
  2. Employee

  The foreign surrogate key reference is contained in the Employee table in the cache.

Message autonomy

Unwired Server expects serialized DCN message updates to MBO instances. That is, do not send two concurrent threads of the same MBO instance to Unwired Server.

Unwired Server expects an entire graph when sending updates to MBOs within a composite relationship.

DCN upsert operations and binary data

When using DCN to upsert binary data to the cache (CDB), the string used for the value of the binary type attribute of the MBO in the request message must conform to a very specific encoding for the DCN request to be processed correctly. Read the binary data into a byte array, then use the following code to obtain it in the correctly encoded format:

```
byte[] picByteArray = < < user code to read binary data into byte[] >
>
String picStringBase64Encoded =
com.sybase.djc.util.Base64Binary.toString(picByteArray);
String picStringUrlEncoded =
java.net.URLEncoder.encode(picStringBase64Encoded, "UTF-8");
```

Use the **picStringUrlEncoded** string as the value for the binary attribute in the DCN request message.

DCN and date, time, and datetime datatypes

DCN accepts date, time, and datetime attribute and parameter values using this format:

- `date` – yyyy-MM-dd
- `time` – HH:mm:ss

- `datetime` – yyyy-MM-dd'T'HH:mm:ss

For example, Unwired Server parses `string` or `long` values and upserts a valid timestamp object:

```
http://localhost:8000/dcn/DCNServlet?
cmd=dcn&username=supAdmin&password=
s3pAdmin&package=testdatetime:1.0&domain=default&dcn_request=
{"pkg":"testdatetime","messages":
[{"id":"1","mbo":"TestDateTimeStamp","op":":upsert",
"ppm":null,"cols":
{"testTimestamp":"2009-08-09T12:04:05","testDate":"2009-08-09","c_i
nt":"0",
"testDateTime":"2009-08-09T12:04:05","testSmalldt":"2009-08-09T12:0
4:05","testTime":"12:04:05"},
}]}
```

MBOs with complex data types must be handled specifically, depending on whether you use Unwired WorkSpace or code entirely written by a developer:

- Manually writing the code – if a package uses a complex type, and it defines MBOs from that returned type , you can use a DCN to update that complex type. For example, PurchaseOrder is a complex type. The MBO is defined with the returned POHeader and POLineItem because of a DCN written to update that PurchaseOrder. The DCN code parses the PurchaseOrder data structure, and then constructs separate DCN upsert requests for each row from the POHeader and POLineItem MBOs that are derived from that PurchaseOrder.
- Unwired WorkSpace – the DCN with payload requires the MBO attribute name-value pairs as payload data. Because the DCN payload data disregards the EIS schema, the developer needs to be aware of how the EIS schema and the MBO attributes are mapped. The most important consideration is the logic used by the developer to flatten either the complex type, database tables, or the Web service internal schema, or how the developer maps to the backend as MBO attributes.

Send DCN messages only to MBOs with load operations that do not take parameters

You cannot use DCNs with MBOs that define more than one partition (that is, a load operation mapped to synchronization parameters in Unwired WorkSpace). An MBO load operation must be managed completely by DCN should not return any data. If the DCN initializes and maintains the cached MBO, the MBO load operation must not return any rows.

If the load operation initializes the MBO, and you use DCNs to maintain the MBO, then associate the MBO with a cache group that implements an infinite schedule. Do not send DCN messages until the cache is initialized.

Cache update policies and DCN

Do not use an cache update policy that invalidates the cache if you use a DCN to populate the MBO.

DCN and deadlocks

The requirements described above (*Message autonomy* and *Send DCN messages only to MBOs with load operations that do not take parameters*) are designed to prevent deadlock situations. However, if you do not define an order of operation execution, deadlocks might occur depending on the DCN implementation or the locking mechanism used by the enterprise information system (EIS). In a deadlock situation, the entire transaction is rolled back (if there are multiple operations in a single DCN) and a `replayFailed` result is returned.

Data Change Notification Results

Each binding in a data change notification (DCN) request is associated with an ID. The result status of the DCN request is returned in JavaScript Object Notation (JSON) format, and includes a list of IDs followed by a Boolean success field and status message, in case of error.

In response to payload and MBO operation DCNs, Unwired Server sends the requester a JSON string containing details about the success and or failure of the operations. This example shows the JSON format for a DCN result for a request of three IDs (recID1, recID2, recID3). The example has been formatted using new lines, and indentations, which are not present in an actual response:

```
[
 {
 "recordIDs":
  [
  "recID1",
  "recID2"
  ],
 "success":true,
 "statusMessage":""
 },
 {
 "recordIDs":
  [
  "recID3"
  ],
 "success":false,
 "statusMessage":"bad msg2
 "}
]
```

This example is unformatted:

- Successful operation:

  ```
  [{"recordIDs“:["1"],"success“:true,"statusMessage“:""}]
  ```

- Failed operation using tildas instead of colons:

  ```
  [{"recordIDs"~["1"],
      "success"~false,"statusMessage"~"Error inferring attribute
  bindings from EIS bindings {DepartmentID\u003d10000,
  ```

```
   DepartmentAlias\u003dTest,
   DepartmentHeadID\u003d501}"}]
```

Data Change Notification Filters

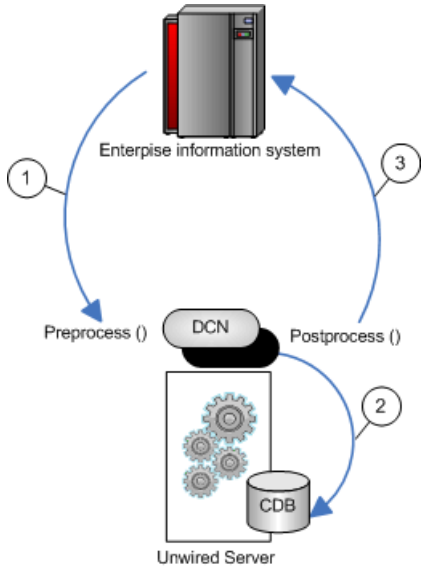
Data change notification (DCN) requests need not always be in the format Unwired Server expects.

You can deploy a DCN filter to Unwired Server and reference it in the DCN request. Unwired Server allows the filter to preprocess the submitted DCN. The filter converts raw data in the DCN request to the required JavaScript Object Notation (JSON) format. The filter can also postprocess the JSON response returned by the Unwired Server into the format preferred by the back end (which is governed by the implementation in the filter class).

The filter interface `DCNFilter` is in the `com.sybase.sup.server.dcn` package in the `sup-server-rt.jar` file. All classes that implement a DCN filter should implement this interface. The functions available in the interface are:

- **String preprocess(String blobDCNRequest, Map<String, String requestHeaders> requestHeaders);** – takes the DCN request as a binary large object (BLOB), converts it into a valid JSON DCN request format, and returns the same.
- **String postprocess(String jsonDCNResult, Map<String, String responseHeaders> responseHeaders);** – takes the DCN result in a valid JSON format, converts it to the EIS-specific format, and returns the same.

Figure 1: DCN filter flow



1. Changed data is sent from the EIS to Unwired Server via a DCN request, where any data preprocessing occurs. For example, the EIS data could be sent to Unwired Server as XML where the preprocess filter converts the data to JSON.
2. The DCN executes. For example, apply data changes directly to the Unwired Server cache.
3. Postprocessed DCN response is sent to the originating EIS as an HTTP response to the original DCN request. For example, the JSON response is converted to XML.

Implementing a Data Change Notification Filter

Write and deploy preprocess and postprocess DCN filters to Unwired Server.

When specifying filters, add a dcn_filter parameter to the base URL, and to the parameters specified in the DCN request section. The **dcn_filter** parameter specifies the fully qualified name of the filter class, which must be in a valid CLASSPATH location so Unwired Server can locate it using its fully qualified name.

JSON requires colons to define the object structure, but since colons have a special function in HTTP URLs, use the tilda character "~" instead of colons ":" when implementing the DCN filter, so the JSON `dcn_request` string can be passed as an HTTP GET or POST parameter:

```
dcn_request={"pkg"~“TestPackage",
    "messages"~[{"id"~"1","mbo"~"Department","op"~"~upsert",
    "cols"~{"DepartmentID"~"3333",
          "DepartmentName"~“My Department",
             "DepartmentHeadID"~"501"}}]}
```

The `dcn_request` is in a format that is specific to the back end. The filter class can preprocess to the JSON format expected by Unwired Server.

1. Write the filter. For example:

```
import java.util.Map;
import com.onepage.fw.uwp.shared.uwp.UWPLogger;
import com.sybase.sup.dcn.DCNFilter;

public class CustomDCNFilter implements DCNFilter
{
  String preprocess(String blobDCNRequest, Map<String,String>
headers) {
         String result = blobDCNRequest.replace(‘~’,’:’);
         return result;
  }

  String postprocess(String jsonDCNResult, Map<String,String>
responseHeaders) {
         String result = jsonDCNResult.replace(‘:’,’~’);
         return result;
  }

    public static void main( String[] args ) { }
}
```

2. Package your DCN filter class in a JAR file.
3. Deploy the JAR file to Unwired Server by using the Deployment wizard from Unwired WorkSpace:
   a) Invoke the deployment wizard. For example, right-click in the Mobile Application Diagram and select **Deploy Project**.
   b) Select the JAR file that contains your DCN filter class files to deploy to Unwired Server in the third screen of the wizard (Package User-defined Classes).
   c) Click **Finish** after selecting the target Unwired Server.
4. Restart Unwired Server.

Custom XSLT Transforms

If you are using data from a SOAP or REST Web service, you may need to use XSLT (Extensible Stylesheet Language Transformations) to modify the structure of the message data generated by the service, so it can be used by an Unwired Platform MBO. Unwired Workspace can create XSLT transforms automatically, however sometimes these generated transforms are not sufficient and do not yield the results you require.

MBOs typically require a flat and tabular message structure from a Web service. This tabular structure corresponds to the rows and columns that eventually materialize the MBO's instances and attributes, respectively. Therefore the message structure used by a Web service must align correctly. Transformation must be precise to avoid unexpected results in an MBO.

Therefore, always validate the transform before deploying it to a production environment.

Custom XSLT Use Cases

In most cases, the XSLT that is generated by Unwired WorkSpace is sufficient. However, in some cases, you may need to modify the generated XSLT file, or to create a new one manually.

Some of these cases include:

- Web service response messages do not precisely conform to the schema required by the WSDL schema.
  For example, the schema indicated that an integer field is not nullable, but the Web service response message failed to return a valid integer value. This omission triggers an error on the device application. even though the root issue is the data from the Web service, not Unwired Platform.
  In this scenario, it is simpler to modify the generated XSLT slightly, by changing the single op_nullable field from `false` to `true`.

Implementing Custom Transforms

When the generated transform does not yield expected results in the MBO, you need to either modify the generated transform or create a custom transform outside of Unwired WorkSpace.

1. Make changes to an existing transform or write a new one.
2. Save the changes and overwrite the file that already exists. This ensures that the binding remains intact for the MBO. See *Unwired WorkSpace > Develop > Developing Mobile Business Objects > Binding Mobile Business Objects to Data Sources*.
3. Redeploy the MBO so changes implemented to Unwired Server, and include the transform in the deployment package.

See *Unwired WorkSpace > Develop > Developing Mobile Business Objects > Packaging and Deploying Mobile Business Objects*.

Note: If you are redeploying to a production environment, ensure the administrator redeploys the MBO with the modified transform.

XSLT Stylesheet Syntax

XSLT stylesheet must follow Unwired Platform stylesheet syntax requirements so that the Web service response message is formatted correctly for MBOs bound to this data source.

The stylesheet is applied to different parts of the Web service response message, depending on the type:

- For SOAP web service response messages, the stylesheet is applied to the contents of the SOAP body.
- For REST web service messages, the stylesheet is applied to the contents of the HTTP response body.

Table 2. Stylesheet elements

Element	Description	Contains
`Data`	The root element of the stylesheet.	One or more `Record` elements.
`Record`	The element that corresponds to a row in the tabular MBO data structure. The first `Record` element resulting from the transformat describes the column using metadata (that is, names, data types, nullability, and so on). The `Record` element has no attributes, except when it is a metadata element. The contents of the `Field` elements should match the corresponding `op_label` values. The `Record` or `Field` values from this first Record element will not appear in the resulting tabular data structure.	One or more `Field` elements.
`Field`	The element that corresponds to the column value. The `Field` element has a number of attributes that can be used.	One or more attributes. See the *Attributes* table.

Table 3. Attributes

Attribute	Applicability	Description
`op_label`	Required by the stylesheet and the resulting transformed structure's metadata. Ignored by the data field elements.	The column name.
`op_bindpath`	Required by the stylesheet and the resulting transformed structure's metadata. Ignored by the data field elements.	Binds the MBO column to the corresponding definition in the XML schema description (that is, either the WSDL or REST description).
`op_position`	Required by all.	The attribute's position in the tabular structure. The first attribute is at position 1
`op_nullable`	Required by the stylesheet and the resulting transformed structure's metadata. Ignored by the data field elements.	Whether (TRUE) or not (FALSE) the attribute is nullable.
`op_datatype`	Required by the stylesheet and the resulting transformed structure's metadata. Ignored by the data field elements.	The data type. Supported values inlclude STRING, INT, LONG, BOOLEAN, DECIMAL, BINARY, FLOAT, DOUBLE, DATE, TIME, DATETIME, CHAR, BYTE, SHORT, INTEGER. See *Sybase Unwired WorkSpace - Mobile Business Object Development > Develop > Working with Mobile Business Objects > Mobile Business Object Data Properties > Datatype Support.*
`op_xsdtype`	Required by the stylesheet and the resulting transformed structure's metadata. Ignored by the data field elements.	The XML schema primitive type name corresponding to this attribute.

XSLT Stylesheet Example

Use the example XSLT stylesheet to understand the structure required by Unwired Platfrom.

The bolded elements are required. The <xsl:stylesheet> needs a <xsl:template> element. The first child element of <xsl:template> must be the <data> that also requires the a metadata <Record> element.

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/
Transform"; xmlns:ns1="urn:Sample_Enrollments" exclude-result-
prefixes="ns1">
        <xsl:template match="//ns1:OpGetListResponse">
                <data>
                        <Record>
                                <Field op_label="Class_Cost"
op_bindpath="getListValues[].Class_Cost" op_position="1"
op_datatype="DECIMAL" op_nullable="false">Class_Cost</Field>
                                <Field op_label="Class_ID"
op_bindpath="getListValues[].Class_ID" op_position="2"
op_datatype="STRING" op_nullable="false">Class_ID</Field>
                                <Field op_label="Class_Start_Date___Time"
op_bindpath="getListValues[].Class_Start_Date___Time"
op_position="3" op_datatype="DATETIME"
op_nullable="false">Class_Start_Date___Time</Field>
                                <Field op_label="Class_Title"
op_bindpath="getListValues[].Class_Title" op_position="4"
op_datatype="STRING" op_nullable="false">Class_Title</Field>
                                <Field op_label="Enrollee_Login"
op_bindpath="getListValues[].Enrollee_Login" op_position="5"
op_datatype="STRING" op_nullable="false">Enrollee_Login</Field>
                                <Field op_label="Temp_Number"
op_bindpath="getListValues[].Temp_Number" op_position="6"
op_datatype="INT" op_nullable="true">Temp_Number</Field>
                        </Record>
                        <xsl:for-each select="ns1:getListValues">
                                <Record>
                                        <Field>
                                                <xsl:attribute
name="op_position">1</xsl:attribute>
                                                <xsl:value-of
select="ns1:Class_Cost"/>
                                        </Field>
                                        <Field>
                                                <xsl:attribute
name="op_position">2</xsl:attribute>
                                                <xsl:value-of
select="ns1:Class_ID"/>
                                        </Field>
                                        <Field>
                                                <xsl:attribute
name="op_position">3</xsl:attribute>
                                                <xsl:value-of
```

```
select="ns1:Class_Start_Date___Time"/>
                                        </Field>
                                        <Field>
                                                <xsl:attribute
name="op_position">4</xsl:attribute>
                                                <xsl:value-of
select="ns1:Class_Title"/>
                                        </Field>
                                        <Field>
                                                <xsl:attribute
name="op_position">5</xsl:attribute>
                                                <xsl:value-of
select="ns1:Enrollee_Login"/>
                                        </Field>
                                        <Field>
                                                <xsl:attribute
name="op_position">6</xsl:attribute>
                                                <xsl:value-of
select="ns1:Temp_Number"/>
                                        </Field>
                                </Record>
                        </xsl:for-each>
                </data>
        </xsl:template>
</xsl:stylesheet>
```

If you use this style sheet, the output generated by this transform would be:

```
<data>
        <Record>
                <Field op_label="Class_Cost"
op_bindpath="getListValues[].Class_Cost" op_position="1"
op_datatype="DECIMAL" op_nullable="false">Class_Cost</Field>
                <Field op_label="Class_ID"
op_bindpath="getListValues[].Class_ID" op_position="2"
op_datatype="STRING" op_nullable="false">Class_ID</Field>
                <Field op_label="Class_Start_Date___Time"
op_bindpath="getListValues[].Class_Start_Date___Time"
op_position="3" op_datatype="DATETIME"
op_nullable="false">Class_Start_Date___Time</Field>
                <Field op_label="Class_Title"
op_bindpath="getListValues[].Class_Title" op_position="4"
op_datatype="STRING" op_nullable="false">Class_Title</Field>
                <Field op_label="Enrollee_Login"
op_bindpath="getListValues[].Enrollee_Login" op_position="5"
op_datatype="STRING" op_nullable="false">Enrollee_Login</Field>
                <Field op_label="Temp_Number"
op_bindpath="getListValues[].Temp_Number" op_position="6"
op_datatype="INT" op_nullable="true">Temp_Number</Field>
        </Record>
        <Record>
                <Field op_position="1">100.00</Field>
                <Field op_position="2">00001</Field>
                <Field op_position="3">2010-07-02T10:27:35-07:00</
Field>
             <Field op_position="4">Managing Within the Law</Field>
```

```
                <Field op_position="5">Demo</Field>
                <Field op_position="6"/>
        </Record>
        <Record>
                <Field op_position="1">150.00</Field>
                <Field op_position="2">00005</Field>
                <Field op_position="3">2005-11-17T08:00:00-08:00</
Field>
               <Field op_position="4">Microsoft Word for Beginners</
Field>
                <Field op_position="5">Demo</Field>
                <Field op_position="6"/>
        </Record>
        <Record>
                <Field op_position="1">299.00</Field>
                <Field op_position="2">00006</Field>
                <Field op_position="3">2005-11-15T08:00:00-08:00</
Field>
                <Field op_position="4">Meeting Planning and
Facilitation</Field>
                <Field op_position="5">Demo</Field>
                <Field op_position="6"/>
        </Record>
</data>
```

Index

C

companion documentation 1
custom development features 1
custom filters
    See result set filters

D

data change notification
    filters 35
data change notification filter
    example 36
    implementing 36
data change notification interface 27
data change notification parameters 28
data change notification syntax 28
data change notification with payload 28
data change notification, results 34
documentation roadmap
    document descriptions 3

F

filters
    data change notification 35
    result set 13

G

guide, introducing 1

H

HTTP interface for data change notification 27

I

interfaces 2
introduction 1

J

Javadocs 2
JPDA
    enabling 13

M

messages, transforming 40

P

parameters, data change notification
    dcn_request 28
    domain 28
    package 28
    password 28
    unwired_server 28
    unwired_server_port 28
    username 28

R

response messages, transforming 40
REST
    transforming data
        See Transforms
result checker 19
    default SAP code 22
result checker, customizing 16
result checker, implementing 15
result checkers 15
result set filters 8, 11
    debugging 13
    deploying 11

S

SAP result checker 15
server API features 1
SOAP
    transforming data
        See Transforms
stylesheet syntax, XSLT 40
syntax, XSLT 40

T

transforms
custom, introducing 39
implementing 39
stylesheet example 42
stylesheet syntax 40
when to use custom files 39
W
Web services
transforming data
See Transforms
X
XSLTs
See Transforms