SYBASE®
An SAP® Company

Developer Guide for BlackBerry

Sybase Unwired Platform 2.0

DOCUMENT ID: DC01215-01-0200-04
LAST REVISED: January 2013

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

Contents

Introduction to Developer Guide for BlackBerry

This developer guide provides information about using advanced Sybase® Unwired Platform features to create applications for RIM BlackBerry devices. The audience is advanced developers who are familiar working with APIs, but who may be new to Sybase Unwired Platform.

This guide describes requirements for developing a device application for the platform, how to generate application code, and how to customize the generated code using the client object API. Also included are task flows for the development options, procedures for setting up the development environment, and client object API documentation.

Companion guides include:

- *Sybase Unwired WorkSpace – Mobile Business Object* Development
- *Tutorial: BlackBerry Application Development*, where you create the SUP101 sample project referenced in this guide.

  Complete the tutorials to gain a better understanding of Unwired Platform components and the development process.
- *Troubleshooting for Sybase Unwired Platform*
- *Javadocs, which provide a complete reference to the APIs, are available from:*
  - Client Object API – the Unwired Platform installation directory `<UnwiredPlatform_InstallDir>\ClientAPI\apidoc`. There is a subdirectory for `rim`.

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

Documentation Roadmap for Unwired Platform

Learn more about Sybase® Unwired Platform documentation.

Table 1. Sybase 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** > **Programs** > **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

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*
	• Create native mobile device applications:
	  • *Tutorial: BlackBerry Application Development*
	  • *Tutorial: iOS Application 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 – Mobile Workflow Package Development	Online help for developing mobile workflow applications. Audience: new and experienced developers. Use: after system installation.
Developer guides 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 Guide for BlackBerry* • *Developer Guide for iOS* • *Developer Guide for Mobile Workflow Packages* • *Developer Guide for Windows and Windows Mobile*

Document	Description
Developer guide for Unwired Server side customization – *Developer Guide 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 guide for system administration customization – *Developer Guide for Unwired Server Management API*	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*.

Introduction to Developing Device Applications with Sybase Unwired Platform

A device application includes both business logic (the data itself and associated metadata that defines data flow and availability), and device-resident presentation and logic.

Within Sybase Unwired Platform, development tools enable both aspects of mobile application development:

- The data aspects of the mobile application are called mobile business objects (MBO), and "MBO development" refers to defining object data models with back-end enterprise information system (EIS) connections, attributes, operations, and relationships that allow segmented data sets to be synchronized to the device. Applications can reference one or more MBOs and can include synchronization keys, load parameters, personalization, and error handling.
- Once you have developed MBOs and deployed them to Unwired Server, develop device-resident presentation and logic for your device application by generating code to use as a base in a native IDE. Follow an API approach that uses your native IDE's Client Object API. Unwired WorkSpace provides MBO code generation options targeted for specific development environments, for example, BlackBerry JDE for BlackBerry device applications, or Visual Studio for Windows Mobile device applications.

  The Client Object API uses the data persistence library to access and store object data in the database on the device. Code generation takes place in Unwired WorkSpace. You can

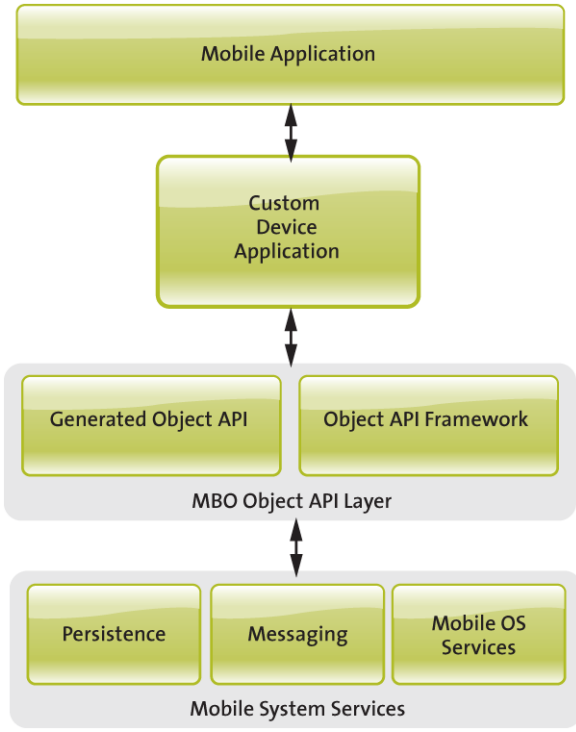
generate code manually, or by using scripts. The code generation engine applies the correct templates based on options and the MBO model, and outputs client objects.

Note: See *Sybase Unwired WorkSpace – Mobile Business Object Development* for procedures and information about creating and deploying MBOs.

Development Task Flow

This section describes the overall development task flow, and provides information and procedures for setting up the development environment, and developing device applications.

This diagram illustrates how you can develop a device application directly from mobile business objects (MBOs), using the Object API and custom device application coding, as shown on the left. This is how you create device applications with sophisticated UI interaction, validation, business logic, and performance.



Task Flow for BlackBerry JDE Development

This describes a typical task flow for creating a device application using the BlackBerry JDE or the BlackBerry Java plug-in for Eclipse (eJDE).

Highlevel steps:

1. Configuring the BlackBerry development environment:

   a. Installing the BlackBerry Java Plug-in for Eclipse or Downloading the BlackBerry JDE and MDS Simulator.
   b. Client API JAR File Locations.
2. Generating Object API code for Mobile Business Objects.
3. Creating a BlackBerry Device Application Project .
4. Adding Required .jar and .cod Files .
5. Developing, Debugging, and Customizing BlackBerry Applications .
6. Deploying Applications to Devices .

Configuring Your BlackBerry Development Environment

This section describes how to set up your BlackBerry development environment, and provides the location of required JAR files and client object APIs.

Installing the BlackBerry Development Environment

Download and install either the BlackBerry JDE or the BlackBerry Java plug-in for Eclipse (eJDE).

You can develop device applications with either the BlackBerry JDE or the BlackBerry Java plug-in for Eclipse,

For information on transitioning from the BlackBerry JDE to the eJDE, view the video at the Research In Motion Developer Video Library Web site: *http://supportforums.blackberry.com/t5/Java-Development/tkb-p/java_dev%40tkb?labels=video*

Installing the BlackBerry Java Plug-in for Eclipse

The BlackBerry Java Plug-in for Eclipse is an IDE for developing BlackBerry applications that leverage the Object API code generated from MBOs.

Prerequisites

You must have a BlackBerry developer account to download the BlackBerry Java Plug-in for Eclipse. You may be required to register if you do not already have an account.

Task

1. Shut down Unwired WorkSpace.
2. Go to *http://us.blackberry.com/developers/javaappdev/* and download the BlackBerry Java Plug-in for Eclipse (full installer) to a temporary folder.
3. Double-click the setup application file.
4. On the Introduction page, click **Next**.
5. Accept or decline the terms of the license agreement and click **Next**.

6. Create and select a new, empty folder for the installation directory and click **Next**.
7. Review the information on the Pre-installation Summary screen and click **Install**.
8. Click **Done**.
   The installation is complete.
9. (Optional). Copy the `plugin` and `features` folders from the installation to `<UnwiredPlatform_InstallDir>\Unwired_WorkSpace\Eclipse\sybase_workspace\mobile\eclipse`.
   This step ensures that Sybase Unwired WorkSpace contains the BlackBerry Java Plug-in for Eclipse, and that users can directly use it from Sybase Unwired WorkSpace instead of opening another instance of Eclipse to work with the BlackBerry Java Plug-in for Eclipse.

Downloading the BlackBerry JDE and MDS Simulator

To generate and distribute BlackBerry device applications, download the MDS simulator and the BlackBerry JDE and its prerequisites from the BlackBerry Web site.

Prerequisites

You must have a BlackBerry developer account to download the BlackBerry JDE. You may be required to register if you do not already have an account. Before you download the JDE, ensure the 32-bit JDK has already been installed, even for 64-bit operating systems; otherwise, MDS will not start.

Task

1. Go to the BlackBerry Web site at *http://us.blackberry.com/developers/javaappdev/javadevenv.jsp* to download and install the BlackBerry JDE.
2. Go to *http://us.blackberry.com/developers/browserdev/devtoolsdownloads.jsp* to download and install the MDS simulator.

Client API JAR File Locations

The client API library JAR files and dependencies are installed in the Sybase Unwired Platform installation directory. JAR files are used for compilation and COD files for runtime. Make sure COD files are deployed to the simulator/device along with the device application.

The contents and location of the client API are:

- Client database (UltraLite®J) libraries – `<UnwiredPlatform_InstallDir>\ClientAPI\RBS\BB\UltraliteJ.jar`.
- Object API libraries - `<UnwiredPlatform_InstallDir>\ClientAPI\RBS\BB\sup_client_rim.jar`.
- Single sign-on libraries - `<UnwiredPlatform_InstallDir>\ClientAPI\RBS\BB\CommonClient.jar`.

Using Object API to Develop a Device Application

Generate object API code on which to build your application.

Unwired Platform provides the Code Generation wizard for generating object API code. Code generation creates the business logic, attributes, and operations for your Mobile Business Object. You can generate code for these platforms:

- RIM BlackBerry

See the guidelines for generating code for each platform type.

Replication-based Synchronization

Generate code for a replication-based synchronization application:

You can deploy a package of replication-based mobile business objects.

Replication-based synchronization

- Supported on BlackBerry and Windows Mobile devices.
- Data flow follows an upload/download pattern.
- Data is synchronous (supports background synchronization).
- Uses the "poke-pull" model of push, where a notification is pushed to the device (poke), and the device fetches the content (pull).

Generating Java Object API Code

Generate object API code containing mobile business object (MBO) references, which allows you to use APIs to develop device applications for BlackBerry devices.

Prerequisites

Before generating device client code, develop the MBOs that will be referenced in the device applications you are developing. A mobile application project must contain at least one non-online MBO. You must have an active connection to the data sources to which the MBOs are bound.

Task

1. Launch the **Code Generation** wizard.

From	Action
The Mobile Application Diagram	Right-click within the Mobile Application Diagram and select **Generate Code**.

From	Action
WorkSpace Navigator	Right-click the Mobile Application project folder that contains the mobile objects for which you are generating API code, and select **Generate Code**.

2. (Optional) Enter the information for these options:

Note: This page of the code generation wizard is seen only if you are using the Advanced developer profile.

Option	Description

Select code generation configuration	Select either an existing configuration that contains code generation settings, or generate device client code without using a configuration:
	• Continue without a configuration – select this option to generate device code without using a configuration.
	• Select an existing configuration – select this option to either select an existing configuration from which you generate device client code, or create a new configuration. Selecting this option enables:
	  • Select code generation configuration – lists any existing configurations, from which you can select and use for this session. You can also delete any and all existing saved configurations.
	  • Create new configuration – enter the **Name** of the new configuration and click **Create** to save the configuration for future sessions. Select an existing configuration as a starting point for this session and click **Clone** to modify the configuration.

3. Click **Next**.
4. In Select Mobile Objects, select all the MBOs in the mobile application project or select MBOs under a specific synchronization group, whose references, metadata, and dependencies (referenced MBOs) are included in the generated device code.

   Dependent MBOs are automatically added (or removed) from the Dependencies section depending on your selections.

Note: Code generation fails if the server-side (run-time) enterprise information system (EIS) data sources referenced by the MBOs in the project are not running and available to connect to when you generate object API code.

5. Click **Next**.
6. Enter the information for these configuration options:

Option	Description
Language	Select **Java**.

Option	Description
Platform	Select the platform (target device) from the drop-down list for which the device client code is intended. • Java ME for BlackBerry **Note:** When generating code into a plain Java project with language 'Java' and platform 'Java Me for BlackBerry', compilation errors are generated because of code references to RIM API's. To avoid errors, generate code into a BlackBerry project.
Unwired Server	Specify a default Unwired Server connection profile to which the generated code connects at runtime.
Server domain	Choose the domain to which the generated code will connect. If you specified an Unwired Server to which you previously connected successfully, the first domain in the list is chosen by default. You can enter a different domain manually. **Note:** This field is only enabled when an Unwired Server is selected.

Option	Description
Page size	Optionally, select the page size for the generated client code. If the page size is not set, the default page size is 4KB at runtime. The default is a proposed page size based on the selected MBO's attributes. The page size should be larger than the sum of all attribute lengths (a binary length greater than 32767 is converted to a Binary Large Object (BLOB), and is not included in the sum; a string greater than 8191 is converted to a Character Large Object (CLOB), and is also not included) for any MBO that is included with all the selected MBOs. If an MBO attribute's length sum is greater than the page size, some attributes are automatically converted to BLOB or CLOB, and therefore, these attributes cannot be put into a `where` clause. **Note:** This field is only enabled when an Unwired Server is selected.
Package	Enter a unique name for the Java package. **Note:** Do not use "java" in package names. The Java package name along with the class name makes the fully qualified class name that must be unique into one RIM JVM. If there is a class with the same fully qualified name, the application may fail on real device.
Destination	Specify the destination of the generated device client files. Enter (or **Browse**) to a **Project path** or **File system path** (Mobile Application project) location, and select `Generated Code` as the destination. JAR files are automatically added to the destination project. Select **Clean up destination before code generation** to clean up the destination folder before generating the device client files.
Replication-based	Select to use replication-based synchronization for the application.

Option	Description
Message-based	This option is not supported for Java applications.

7. Select **Generate metadata classes** to generate metadata for the attributes and operations of each generated client object.
8. Select **Generate metadata and object manager classes** to generate both the metadata for the attributes and operations of each generated client object and an object manager for the generated metadata.

   The object manager allows you to invoke MBOs using metadata instead of the object instances.
9. (Optional) Select **Generate JavaDoc** to generate API documentation from the source code.
10. Click **Finish**.

Generated Code Location and Contents

Generated object API code is stored in the project's Generated Code sub-folder by default, for example, `C:\Documents and Settings\administrator\workspace\<Unwired Platform project name>\Generated Code\src`. Language, platform, and whether or not you select the Generate metadata classes option determines the class files generated in this folder.

Assuming you generate code in the default location, you can access it from WorkSpace Navigator by expanding the Mobile Application project folder for which the code is generated, and expand the Generated Code folder.

The contents of the folder is determined by the options you selected from the Generate Code wizard, and include generated class (.java) files that contain:

- MBO - the business logic of your MBO.
- Synchronization parameters - any synchronization parameters for the MBOs.
- Personalization - personalization and personalization synchronization parameters used by the MBOs.
- Metadata - if you selected **Generate metadata classes**, the metadata classes which allow you to use code completion and compile-time checking to ensure that run-time references to the metadata are correct.

Validating Generated Code

Validation rules are enforced when generating client code for C# and Java. Define prefix names in the Mobile Business Object Preferences page to correct validation errors.

Sybase Unwired WorkSpace validates and enforces identifier rules and checks for key word conflicts in generated Java and C# code. For example, by displaying error messages in the Properties view or in the wizard. Other than the known name conversion rules (converting '.' to

'_', removing white space from names, and so on), there is no other language specific name conversion. For example, cust_id is not changed to custId.

You can specify the prefix string for mobile business object, attribute, parameter, or operation names from the Mobile Business Object Preferences page. This allows you to decide what prefix to use to correct any errors generated from the name validation.

1. Select **Window > Preferences**.
2. Expand **Sybase, Inc > Mobile Development**.
3. Select **Mobile Business Object**.
4. Add or modify the **Naming Prefix** settings as needed.

The defined prefixes are added to the names (object, attribute, operation, and parameter) whenever these are auto-generated. For example, when you drag-and-drop a data source onto the Mobile Application Diagram.

Creating Projects and Importing Files into the BlackBerry Development Environment

Set up the BlackBerry project, add required libraries, and import mobile business object (MBO) generated files. Use these procedures if you are developing a device application using the BlackBerry JDE or the BlackBerry Java plug-in for Eclipse.

Mobile Business Object Required Files

Develop a device application directly from mobile business object (MBO) generated code.

The main characteristics are:

- Mobile business objects – contain only MBO business logic. You must:
  - Include libraries and JAR files in the BlackBerry project that support the BlackBerry Client Object API.
  - Add the Java files from the MBO Generated Code folder to the BlackBerry project.

Differences Between the BlackBerry Java Plug-in and BlackBerry JDE

To develop a device application using the BlackBerry Java plug-in for Eclipse, use the same procedure as developing with the BlackBerry JDE, but note the differences.

- Libraries cannot be located inside BlackBerry projects developed using the BlackBerry Java plug-in for Eclipse, due to a RIM limitation. The libraries must be outside the projects and referred to with an absolute path.

Creating a Project in the BlackBerry JDE

Create the BlackBerry project and add the generated mobile business object (MBO) Java files to the BlackBerry JDE.

1. Launch the BlackBerry JDE and create a new workspace.
2. Create a BlackBerry project and name it `supClients`.
3. Right-click the project and select **Properties**.
4. In the properties dialog, select the **Application** tab, specify `Application` for **Project type** and select **Always make project active** in the General tab of the properties for the project.
5. Select the **Build** tab, and click **Add** next to "Imported jar files." Add files as described in *Developer Guide for BlackBerry > Development Tasks Flows > Creating Projects and Importing Files into the BlackBerry Development Environment > Adding Required .jar and .cod Files*.
6. Click **OK**.
7. Copy the MBO generated Java code from the generated location to the project location.
   - MBO generated code – references the Client object API and contains the Java files that implements the business logic of your project. Navigate to the `src` subdirectory where you generated the Java code from your Unwired WorkSpace mobile application. This location is dependent on the workspace that you used.

     For example, if your workspace is in the `C:\myBBapplications` directory and the name of the mobile application project is `test`, navigate to `C:\myBBapplications\test\Generated Code\src\test` and copy all of the `.java` files to your project.

Creating a Project in the BlackBerry Java Plug-in for Eclipse

Create a new BlackBerry project in the BlackBerry Java Plug-in for Eclipse..

1. Start the BlackBerry Java Plug-in for Eclipse.
2. From the toolbar, select **New > BlackBerry Project**.
3. In the New BlackBerry Project wizard, use these values and click **Next**.
   - Name – enter `SUPClient`
   - Use a project specific JRE – select **BlackBerry JRE 6.0.0**

Adding Required .jar and .cod Files

Add the following Unwired Platform .jar and .cod file references to the BlackBerry project's Java build path.

- `sup_client_rim.jar` – from `<UnwiredPlatform_InstallDir>\ClientAPI\RBS\BB` for the Blackberry client.
- `UltraLiteJ.jar` from `<UnwiredPlatform_InstallDir>\ClientAPI\RBS\BB` for the BlackBerry client.
- `CommonClient.jar` from `<UnwiredPlatform_InstallDir>\ClientAPI\RBS\BB` for the BlackBerry client.

Copy required .cod files to the BlackBerry simulator directory:

- `UltraLiteJ.cod` from `<UnwiredPlatform_InstallDir>\ClientAPI\RBS\BB` for the BlackBerry client.
- `sup_client_rim.cod` and `CommonClient.cod` from `<UnwiredPlatform_InstallDir>\ClientAPI\RBS\BB` for the BlackBerry client.
- `CommonClient.cod` and `CommonClient.cod` from `<UnwiredPlatform_InstallDir>\ClientAPI\RBS\BB` for the BlackBerry client.

Developing, Debugging, and Customizing BlackBerry Applications

Use the BlackBerry Client Object API, as well as native Research in Motion (RIM) APIs to create or customize your device applications.

To learn more about the BlackBerry JDE, BlackBerry Java plug-in for Eclipse, or RIM BlackBerry APIs, go to the BlackBerry Java application development Web site at *http://na.blackberry.com/eng/developers/javaappdev/*.

Note: Do not modify generated MBO code directly. Create a layer on top of the MBOs using patterns native to the mobile operating system development to extend and add functionality.

Configuring an Application to Synchronize and Retrieve MBO Data

This example illustrates the basic code requirements for connecting to Unwired Server, updating mobile business object (MBO) data, and synchronizing the device application from a Client Object API based device application.

1. Configure a synchronization profile to point to your host and port.

   ```
   TestDB.getSynchronizationProfile().setServerName( "localhost" );
   TestDB.getSynchronizationProfile().setPortNumber(2480);
   ```

2. Log in to Unwired Server using a user name and password. This step is required for application initialization.

   ```
   TestDB.loginToSync("supAdmin", "s3pAdmin");
   ```

3. Subscribe to Unwired Server. Unwired Server creates a subscription for this particular application.

```
TestDB.subscribe();
```

4. Synchronize with Unwired Server. Synchronization uploads all the local changes and downloads new data with related subscriptions.

```
ObjectList sgs = new ObjectList();
sgs.add(TestDB.getSynchronizationGroup("default"));
TestDB.beginSynchronize(sgs, "mycontext");
```

5. List all customer MBO instances from the local database using an object query. `FindAll` is a pre-defined object query.

```
ObjectList customers = Customer.findAll();
for(int i = 0; i < customers.size(); i++)
{
  Customer customer = (Customer)(customers.elementAt(i));
  System.out.println("customer: " + customer.getFname() + " " +
customer.getLname()
  + " " + customer.getId()  + customer.getCity());
}
```

6. Find and update a particular MBO instance, and save to the local database.

```
Customer cust = Customer.findByPrimaryKey(100);
cust.setAddress("1 Sybase Dr.");
cust.setPhone("9252360000");
cust.save();//or cust.update();
```

7. Submit the pending changes. The changes are ready for upload, but have not yet been uploaded to the Unwired Server.

```
Customer.submitPendingOperations();
```

8. Upload the pending changes to the Unwired Server and get the replay results and all the changed MBO instances.

```
<PkgName>DB.beginSynchronize(sgs, "mycontext");
```

9. Unsubscribe the device application if the application is no longer used.

```
TestDB.unsubscribe();
```

Device Application Example Code

Example code for an object API-based client application.

```
package test;

import com.sybase.persistence.*;
import com.sybase.collections.*;

class MyCallbackHandler extends DefaultCallbackHandler
{
  public int onSynchronize(ObjectList groups,
    com.sybase.persistence.SynchronizationContext context)
  {
    System.out.println("OnSynchronize is called with user context: "
+ context.getUserContext());
     return SynchronizationAction.CONTINUE;
  }
}
```

```
public class JavaMain extends net.rim.device.api.ui.UiApplication{
  public JavaMain() {    }

  public static void main(String [] args) throws Exception
  {
    //Configure synchronization profile to point to your host and
port
    TestDB.getSynchronizationProfile().setServerName( "localhost" );
    TestDB.getSynchronizationProfile().setPortNumber(2480);

    //If the application requires a callback (for example, to allow
the client framework to provide
    //notification of synchronization results),register the callback
    //function after setting up the connection profile, by executing
    MyCallbackHandler callback = new MyCallbackHandler();
            TestDB.registerCallbackHandler(callback);

   //Login to Unwired Server. This step is required for application
initialization.
    TestDB.loginToSync("supAdmin", "s3pAdmin");

    //Subscribe to Unwired Server. Unwired Server creates a
subscription for this particular application.
    TestDB.subscribe();

    //Synchronize with Unwired Server. Synchronization uploads all
the local changes and downloads new data with related subscriptions.
    ObjectList sgs = new ObjectList();
    sgs.add(TestDB.getSynchronizationGroup("default"));
    TestDB.beginSynchronize(sgs, "mycontext");

   //Wait for synchronization to complete. May not be required for a
typical UI application.
    Thread.sleep(1000*10);

   //List all customer MBO instances from the local database using a
named query. FindAll is a pre-defined object query.
    //Alternatively you can use Dynamic Query to MBO instances too.
    ObjectList customers = Customer.findAll();
    for(int i = 0; i < customers.size(); i++)
    {
      Customer customer = (Customer)(customers.elementAt(i));
      System.out.println("customer: " + customer.getFname() + " " +
customer.getLname()
      + " " + customer.getId()  + customer.getCity());
     }

     //Find a particular MBO instance.
     Customer cust = Customer.findByPrimaryKey(441);
     cust.setAddress( "1 Sybase Dr.");
     cust.setPhone( "9252360000");

     //Update the customer MBO instance. Save to the local database.
     cust.save();
     //Submit the pending changes. The changes are ready for upload,
```

```
but have not yet been uploaded to the Unwired Server.
     cust.submitPending();

     //Upload the pending changes to the Unwired Server and get the
replay results and all the changed MBO instances.
     TestDB.beginSynchronize(sgs, "mycontext");

      //Unsubscribe the device application if the application is no
longer used.
      TestDB.unsubscribe();
  }
}
```

Adding a Device Application Entry Point

Add a main file to the BlackBerry device application.

1. From the BlackBerry Application project that contains your generated MBO code, for example `supClient`, add a new file by right-clicking the project and selecting **Create new file in project**.
2. Name the file, for example, `BBMain`. Click **OK**.

   This file is the main entry point to the device application.
3. Import the common BlackBerry device application development packages as well as the package that contains your MBOs (for example, com.custom.MBO.*).

   You can now create the code to connect to Unwired Server, access and synchronize your MBOs, and perform other functions.

Configuring Unwired Server to Use HTTPS for RBS

Enable SSL encryption by configuring the replication-based synchronization HTTPS port.

1. In the left navigation pane of Sybase Control Center for Unwired Platform, expand the **Servers** node and click the server name.
2. Click **Server Configuration**.
3. In the right administration pane, on the **Replication** tab, click **Synchronization Listener**.
4. Select `Secure synchronization port` as the protocol used for synchronization and configure the certificate properties, then in the optional properties section, specify the `myserver_identity.crt` certificate file using the fully qualified path to the file, along with the password you entered during certificate creation.

   Note: In a clustered environment, this fully qualified path must work for all nodes in the cluster. You can do this via a a shared disk, or distribute this file manually to all nodes.

   See *Configuring Replication-Based Synchronization Properties.* in the Sybase Control Center online help.

Developing the BlackBerry Device Application

This section provides procedures and compares the differences between creating a BlackBerry application from mobile business object generated code in the BlackBerry JDE versus the Blackberry Eclipse plug-in (eJDE).

Prerequisites

The following procedures requires you to create, deploy, and generate code from the mobile business objects (MBOs) used in *Tutorial: BlackBerry Application Development*, which creates the business logic and generates the Java files required for the application. Sybase recommends that you complete the tutorial.

For either development approach:

1. Since KeywordFilterField is employed in this sample, which is available since JDE 4.5.0, make sure this sample is used in the proper BlackBerry operating system.
2. The generated code SUP101.Customer is modified to override the `toString()` method so that the KeywordFilterField displays the data properly.

Task

Developing a BlackBerry Device Application using the BlackBerry Eclipse Plug-in

Follow these procedures to run the SUP101 project in the BlackBerry® Java® Plug-in for Eclipse®.

1. Modify the build path to point to the correct location for the `sup_client_rim.jar`, `CommonClient.jar`, and `UltraLiteJ.jar` files.

   The files cannot be located in the current project due to a RIM BlackBerry Plug-in restriction.
2. Copy `sup_client_rim.cod`, `CommonClient.cod`, and `UltraLiteJ.cod` files to the simulator directory.
3. Deploy the SUP101 project to the Unwired Server to which the sample refers.
4. Modify `SUP101DB.java` to include your Unwired Server information(lines 47-51). For example:

   ```
   getSynchronizationProfile().setServerName("<UnwiredServerHost>");
   getSynchronizationProfile().setPortNumber(2480);
   getSynchronizationProfile().setNetworkProtocol("http");
   getSynchronizationProfile().setNetworkStreamParams
   ("trusted_certificates=;url_suffix=");
   getSynchronizationProfile().setDomainName("default");
   ```
5. Run the project.

Developing a BlackBerry Device Application using the BlackBerry JDE

Follow these procedures to run the SUP101 project in the BlackBerry JDE.

1. Open the BlackBerry JDE and create a new workspace.
2. Create a new project in the new workspace.
3. Change the Project Type to be CLDC Application or BlackBerry Application (depending on the JDE you are using).
4. Add an image file to Icon files.
5. Add `sup_client_rim.jar`, `CommonClient.jar`, and `UltraLiteJ.jar` files to the Build import jar files.
6. Copy `sup_client_rim.cod`, `CommonClient.cod`, and `UltraLiteJ.cod` files to the simulator directory.
7. Deploy the SUP101 project to the Unwired Server to which the sample refers.
8. Modify `SUP101DB.java` to include your Unwired Server information (lines 47-51). For example:

```
getSynchronizationProfile().setServerName("<UnwiredServerHost>");
getSynchronizationProfile().setPortNumber(2480);
getSynchronizationProfile().setNetworkProtocol("http");
getSynchronizationProfile().setNetworkStreamParams
("trusted_certificates=;url_suffix=");
getSynchronizationProfile().setDomainName("default");
```

9. Run the project.

Debugging BlackBerry Device Development

Device client and Unwired Server troubleshooting tools for diagnosing RIM® BlackBerry® development problems.

Client-side debugging

Problems on the device client side that may cause client application problems:

- Unwired Server connection failed.
- Data does not appear on the client device.
- Physical device problems, such as low battery or low memory.

To find out more information on the device client side:

- If you have implemented debugging in your generated or custom code (which Sybase recommends), turn on debugging, and review the debugging information. See *Developer Guide for BlackBerry* about using the MBOLogger class to add log levels to messages reported to the console.
- Check the log record on the device. Use the **<PkgName>DB.getLogRecords** (**com.sybase.persistence.Query**) or **Entity.getLogRecords()** methods. Use this method for logs corresponding to MBO classes.

This is the log format:

```
level,code,eisCode,message,component,entityKey,operation,requestI
d,timestamp
```

This is a log sample:

```
5,500,'','java.lang.SecurityException:Authorization failed:
Domain = default Package = end2end.rdb:1.0 mboName =
simpleCustomer action =
delete','simpleCustomer','100001','delete','100014','2010-05-11
14:45:59.710'
```

- `level` – the log level currently set. Values include: 1 = TRACE, 2 = DEBUG, 3 = INFO, 4 = WARN, 5 = ERROR, 6 = FATAL, 7 = OFF.
- `code` – replication-based synchronization, Unwired Server administration codes:
  - 200 – success.
  - 500 – failure.
- `eisCode` – not currently used.
- `message` – the message content.
- `component` – Mobile Business Object (MBO) name.
- `entityKey` – MBO surrogate key, used to identify and track MBO instances and data.
- `operation` – operation name.
- `requestId` – operation replay request ID or messaging-based synchronization message request ID.
- `timestamp` – message logged time, or operation execution time.

- Check the Storm event log:
  1. On the Home screen, press Hold.
  2. Click the upper-left corner and upper-right corner twice.
  3. Review the event log.
- Check the BlackBerry event log:
  1. On the device, press ALT+lglg; or, for touch-screen devices, hold the ESC key, tap (no click) top-left, top-right, top-left, then top-right.
  2. Review the event log, and see the RIM BlackBerry documentation for information about debugging and optimizing.*http://na.blackberry.com/eng/developers/resources/A50_How_to_Debug_and_Optimize_V2.pdf*

Server-side debugging

Problems on the Unwired Server side that may cause device client problems:

- The domain or package does not exist.
- Authentication failed for the synchronizing user.
- The operation role check failed for the synchronizing user.
- Back-end authentication failed.

- An operation failed on the remote, replication database back end, for example, a table or foreign key does not exist. Detailed messages can be found in the Log Record.
- An operation failed on the Web service, REST, or SAP® back end. You can find detailed messages in the log record.

To find out more information on the Unwired Server side:

- Check the MMS server log files. See the *Sybase Control Center* documentation for more information.

Debugging the BlackBerry Device Application

Debug your device application by setting breakpoints and stepping through code.

1. From the BlackBerry JDE, select **Debug > Go** to build and execute the application, and launch the simulator.

   You can view build results in the JDE output window.
2. Add breakpoints to the code:
   a) Place your cursor in the code where you want to add a breakpoint and select **Debug > Breakpoint > Set Breakpoint at Cursor**.
   b) You can also set breakpoints for a given event from the same menu, for example, **On startup**, **When an exception is thrown**, **Before garbage collection**, and so on.
3. Run the application from the simulator. The application stops based upon the breakpoint you set.
4. Once stopped, you can step through the code using any of the step icons (step over, step into, step out, and so on) located in the JDE toolbar:

   For more information about the various views available for debugging, including determining memory usage, code coverage, and so on, refer to the BlackBerry JDE documentation. To view a video on how to debug your BlackBerry device application in the BlackBerry JDE, go to the Research In Motion Developer Video Library Web site at: *http://supportforums.blackberry.com/t5/Java-Development/tkb-p/java_dev%40tkb?labels=video*.

Localizing a BlackBerry Application

To localize a BlackBerry application, you must create a resource header file, a resource content file for the global locale, and a resource content file for any specific locales that you require.

Adding a Resource File to the Application

Add a resource file to define the descriptive keys for each localized string.

1. Open the BlackBerry application using the Java Perspective in Eclipse.

2. Focus on the `res` folder, and right-click and select **New** > **Package**.
3. In the New Java Package dialog, in the Name field, enter the same package name as the `src` package name, for example, "com.sybase.sup.samples.objectapi."
4. Add the resource file under `res` > `<package-name>`.
   - Focus on `res` > `<package-name>` and right-click and select **New** > **Other**.
5. In the New dialog, select **BlackBerry** > **BlackBerry Resource File** and click **Next**.
6. In the New BlackBerry Resource File dialog, under the `res` package, enter the a file name for the `rrh` (resource header file) in the File name field. Name it by the project name.

   When you create a new resource header file, the BlackBerry® Java® Plug-in for Eclipse® creates the associated .rrc resource content file. For example, entering `SUP101Sample.rrh` creates `SUP101Sample.rrh` and `SUP101Sample.rrc` files.

   You can create additional resource content files as required for specific locales. These files must have the same name as the resource header file, followed by an underscore (_) and the language code, and then, optionally, by a single underscore (_) and a country code. Language and country codes are specified in ISO-639 and ISO-3166, respectively.

Adding Resource Keys and Values

Localize a BlackBerry application by adding a resource files to the application, and adding localization code to the application source file.

1. Focus on the `rrh` (resource header) file and double-click it to open the Resource Editor.
2. Add resource keys to the resource header file by selecting **Add Key** from the Root tab. The resource keys are added in the Root tab, indicating that these resource keys have been added to the resource header file. The keys are also automatically created in each of the resource content files.
3. Enter resource values in each of the resource content files.

Adding Localization Code

Add localization code into the application file. The following example is from the SUP101 project.

1. Open the `CustomerSampleScreen.java` file in the SUP101Sample project. Add the following code:

```
//import resource bundle interface. SUP101SampleResource is the
resource bundle interface created automatically
import com.sybase.sup.samples.objectapi.SUP101SampleResource;
```

2. Add the following code to the concrete screen code:

```
implements SUP101SampleResource
```

```
private static ResourceBundle _resources =
ResourceBundle.getBundle(BUNDLE_ID, BUNDLE_NAME);
```

3. Call the resource bundles string to display user interface text, and change the string to call the resource bundles to display. Add the following code:

```
InfoScreen(CustomerSampleScreen sampleScreen, Customer customer)
{
  _sampleScreen = sampleScreen;
  _customer = customer;

  // Set up and display UI elements. Use resource bundle string to
display.
  setTitle(_resources.getString(UPDATE_TITLE));
   _fnameField = new
BasicEditField(_resources.getString(FIELD_FNAME),
customer.getFname(),
BasicEditField.DEFAULT_MAXCHARS,Field.FOCUSABLE);
   _lnameField = new
BasicEditField(_resources.getString(FIELD_LNAME),
customer.getLname(),
BasicEditField.DEFAULT_MAXCHARS,Field.FOCUSABLE);
   _companyField = new
BasicEditField(_resources.getString(FIELD_COMPANY),
customer.getCompany_name(), BasicEditField.DEFAULT_MAXCHARS,
Field.FOCUSABLE);
   _addressField = new
BasicEditField(_resources.getString(FIELD_ADDRESS),
customer.getAddress(), BasicEditField.DEFAULT_MAXCHARS,
Field.FOCUSABLE);
   _stateField = new
BasicEditField(_resources.getString(FIELD_STATE),
customer.getState(), BasicEditField.DEFAULT_MAXCHARS,
Field.FOCUSABLE);
   _cityField = new
BasicEditField(_resources.getString(FIELD_CITY),
customer.getCity(), BasicEditField.DEFAULT_MAXCHARS,
Field.FOCUSABLE);
   _phoneField = new
BasicEditField(_resources.getString(FIELD_PHONE),
customer.getPhone(), BasicEditField.DEFAULT_MAXCHARS,
Field.FOCUSABLE);
  _zipField = new BasicEditField(_resources.getString(FIELD_ZIP),
customer.getZip(), BasicEditField.DEFAULT_MAXCHARS,
Field.FOCUSABLE);
```

Validating the Localization Changes

Test that your changes appear in your application.

1. Launch the BlackBerry simulator then launch the application.
2. Select **Options**.
3. Select the language you want to test.
   The simulator restarts in the new language.

4. Launch your application and verify that it is localized.

Deploying Applications to Devices

This section describes how to deploy customized mobile applications to devices.

Device Registration

Replication devices are used exclusively with replication-based synchronization (RBS) mobile business objects that rely on RBS data cached in the consolidated database. RBS device users are automatically registered when they first synchronize data. There is no device configuration required; the only tasks an administrator performs are monitoring RBS device activity and deleting RBS devices.

Note: For more information on device registration, see *Sybase Unwired Platform System Administration > Device and User Management > Replication Devices* and *Sybase Unwired Platform System Administration > Device and User Management > Device Provisioning.*

Signing

Code signing is required for applications to run on physical devices.

You can implement code signing from the BlackBerry JDE:

- BlackBerry JDE – download the Signing Authority Tool from the BlackBerry Web site at *http://na.blackberry.com/eng/developers/javaappdev/signingauthority.jsp*. View Deploying and Signing Applications in the BlackBerry JDE plug-in for Eclipse at the Research In Motion Developer Video Library Web site: *http://supportforums.blackberry.com/t5/Java-Development/tkb-p/java_dev%40tkb?labels=video*.

Provisioning Options for BlackBerry Devices

To provision the application to BlackBerry devices, you can automatically push the application to the device or send a link to device users so they can install it when desired. For small deployments or evaluation purposes, device users can install the application using BlackBerry Desktop Manager.

Once installed on the device, the application appears in Downloads. However, device users can move it to a different location. If device users reinstall the application from a link or URL, or using Desktop Manager, the BlackBerry device remembers the installation location.

Provisioning Method	Purpose	Description
BlackBerry Enterprise Server (BES) Over-the-Air (OTA)	Enterprise installations	When the BlackBerry device activates, it automatically pairs with the BES and downloads the application. See *http://www.blackberry.com/btsc/search.do?cmd=displayKC&docType=kc&externalId=KB03748* for step-by-step instructions.
OTA: URL/link to installation files	Enterprise installations	The administrator stages the OTA files in a Web-accessible location and notifies BlackBerry device users via an e-mail message with a link to the JAD file.
Desktop Manager	Personal installation	Installs the application when the BlackBerry device is synced via a computer.

BES and BIS Provisioning for BlackBerry

BlackBerry devices that are connected to a production environment using relay server can use either BlackBerry Enterprise Server (BES) or BlackBerry Internet Server (BIS) to provision supported device types.

See the following sections in *System Administration* for details on how to perform BlackBerry provisioning and deployment:

- *System Administration > Device Provisioning > Afaria Provisioning and Mobile Device Management*.
- *System Administration > Device Provisioning > BES and BIS Provisioning for BlackBerry*
  - *Provisioning Prerequisites for BlackBerry*
- *System Administration > Device Provisioning > Setting up Push Synchronization for Replication Synchronization Devices*

BlackBerry Desktop Manager Provisioning

You can deploy BlackBerry applications to physical devices through BlackBerry Desktop Manager.

The generated code is compiled against the BlackBerry RAPC compiler to output the following COD (.cod), Application Loader Files (.alx), and Java Application Descriptor (.jad) files. File requirements depend on application and installation type:

Required files include:

- UltralLiteJ.cod/.alx/.jad files

- sup_client_rim files
- CommonClient files

Reference

This section describes the Client Object API. Classes are defined and sample code is provided.

BlackBerry Client Object API

The Sybase Unwired Platform BlackBerry Client Object API consists of generated business object classes that represent the mobile business object model built and designed in the Unwired WorkSpace development environment. The BlackBerry Client Object API is used by device applications to retrieve data and invoke mobile business object operations.

To generate Client Object API Javadoc, select the **Generate JavaDoc** option when generating mobile business object (MBO) code.

Client Object API Javadocs

Use the Sybase Client Object API Javadocs as a Client Object API reference.

Review the reference details in the Client Object API Javadocs, located in the Unwired Platform installation directory `<UnwiredPlatform_InstallDir>\ClientAPI\apidoc`. There is a subdirectory for `rim`.

From the `index.html` file, the top left navigation pane lists all packages installed with Unwired Platform. The applicable documentation is available with each package. Click this link and navigate through the Javadoc as required.

Connection APIs

The Connection APIs contain methods for managing local database information, establishing a connection with the Unwired Server, and authenticating.

ConnectionProfile

The `ConnectionProfile` class manages local database information. You must set its properties before creating a local database.

By default, the database class name is generated as "packageName"+"DB".

```
ConnectionProfile profile = <PkgName>DB.getConnectionProfile();
profile.setPageSize( 4*1024 );
profile.setEncryptionKey(”Your key”);
```

Improving Device Application Performance with Multiple Database Reader Threads

The `maxDbConnections` property improves device application performance by allowing multiple threads to read data concurrently from the same local database.

Note: Replication based synchronization clients support a single write thread concurrently with multiple read threads.

In a typical device application such as Sybase Mobile CRM, a list view lists all the entities of a selected type. When pagination is used, background threads load subsequent pages. When the device application user selects an entry from the list, the detail view of that entry displays, and loads the details for that entry.

Prior to the implementation of `maxDbConnections`, access to the package on the local database was serialized. That is, an MBO database operation, such as, create, read, update, or delete (CRUD) operation waits for any previous operation to finish before the next is allowed to proceed. In the list view to detail view example, when the background thread is loading the entire list, and a user selects the details of one entry to display, the loading of details for that entry must wait until the entire list is loaded, which can be a long while, depending on the size of the list.

You can specify the amount of reader threads using `maxDbConnections`. The default value is 2.

Implementing maxDbConnections

The ConnectionProfile class in the persistence package includes the `maxDbConnections` property, that you set before performing any operation in the application. The default value (maximum number of concurrent read threads) is two.

```
ConnectionProfile connectionProfile =
MyPackageDB.getConnectionProfile();
```

To allow 6 concurrent read threads, set the maxDbConnections property to 6 in ConnectionProfile before accessing the package database at the beginning of the application.

```
    connectionProfile.setMaxDbConnections(6);
```

SynchronizationProfile

Before synchronizing with Unwired Server, you must configure a client with information for establishing a connection with the Unwired Server where the mobile application has been deployed. The ConnectionProfile class manages that information.

You can configure the synchronization connection profile using the package database class:

```
ConnectionProfile profile = <PkgName>DB.getSynchronizationProfile();
profile.setServerName( "sup.sybase.com" );
profile.setPortNumber( 2480 );
profile.setNetworkProtocol( “http” );
profile.setDomainName( “default” );
```

Connect through a Relay Server

To enable your client application to connect through a Relay Server you must make manual configuration changes in the object API code to provide the Relay Server properties.

Edit <package-name>DB by modifying the values of the Relay Server properties for your Relay Server environment.

To update properties for Relay Server installed on Apache on Linux:

```
getSynchronizationProfile().setServerName("examplexp-vm1");
getSynchronizationProfile().setPortNumber(2480);
getSynchronizationProfile().setNetworkProtocol("http");
getSynchronizationProfile().setNetworkStreamParams("trusted
certificates=;url_suffix=/cli/iarelayserver/<FarmName>");
getSynchronizationProfile().setDomainName("default");
```

To update properties for Relay Server installed on Internet Information Services (IIS) on Microsoft Windows:

```
getSynchronizationProfile().setServerName("examplexp-vm1");
getSynchronizationProfile().setPortNumber(2480);
getSynchronizationProfile().setNetworkProtocol("http");
getSynchronizationProfile().setNetworkStreamParams("trusted
certificates=;url_suffix=ias_relay_server/client/rs_client.dll/
<FarmName>");
getSynchronizationProfile().setDomainName("default");
```

For more information on Relay Server configuration, see *System Administration* and *Sybase Control Center for Unwired Server*.

Authentication

The generated package database class already provides a valid synchronization connection profile. You can log in to the Unwired Server with your user name and credentials.

The package database class provides the following methods for logging in to the Unwired Server:

- `public static void onlineLogin(String username, String password);`
- `public static bool offlineLogin(String username, String password);`
- `public static void loginToSync(String username, String password);`

`onlineLogin` authenticates the credentials against the Unwired Server.

`offlineLogin` authenticates against the last successfully authenticated credentials. There is no communication with Unwired Server in this method.

`loginToSync` tries `offlineLogin` first. If `offlineLogin` fails, it will try `onlineLogin`. This is the recommended login method. `loginToSync` brings the

KeyGenerator to the client from the Unwired Server. The KeyGenerator is an MBO for storing key values that are known to both the server and the client. On `loginToSync` from the client, the server sends down a value that the client can use when creating new records (by using the method [`KeyGenerator generateId`] to create key values that the server will accept).

The KeyGenerator is set up so that the value increments each time the `generateId` method is called. A periodic call to `submitPending` by the KeyGenerator generateId MBO sends the most recently used value to the Unwired Server, to let the Unwired Server know what keys have been used on the client side.

```
<PkgName>DB.loginToSync("username", "password");
```

Note: Call `loginToSync` at least once before using the specific Sybase Unwired Platform package.

Synchronization APIs

You can synchronize mobile business objects (MBOs) based on synchronization parameters, for individual MBOs, or as a group, based on the group's synchronization policy.

Changing Synchronization Parameters

Synchronization parameters let you change the parameters used to retrieve data from an MBO during a synchronization session.

The primary purpose of synchronization parameters is to partition data. Change the synchronization parameters to affect the data you are working with (including searches), and synchronization.

```
CustomerSynchronizationParameters sp =
Customer.getSynchronizationParameters();
sp.setMyid(10001);
sp.save();
```

Performing Mobile Business Object Synchronization

A synchronization group is a group of related MBOs. A mobile application can have predefined synchronization groups. An implicit default synchronization group includes all the MBOs that are not in any other synchronization group.

Two synchronization methods are provided in the package database class. You can synchronize a specified group of MBOs using the synchronization group name:

```
<PkgName>DB.synchronize(“sync_group”);
```

Or, you can synchronize all synchronization groups:

```
<PkgName>DB.synchronize();
```

Query APIs

The Query APIs allow you to retrieve data from mobile business objects, to retrieve relationship and paging data, and to retrieve and filter a query result set.

Retrieving Data from Mobile Business Objects

You can retrieve data from mobile business objects through a variety of queries including object queries, arbitrary find, and through filtering query result sets.

Object Query

To retrieve data from the local database, use one of the static Object Query methods in the MBO class.

Object Query methods are generated based on Object Queries defined in Unwired WorkSpace by the modeler. Object Query methods have whatever query name, parameters and return type that were defined in Unwired WorkSpace. Object Query methods return one object, or a collection of objects that match the specified search criteria defined in the Object Query.

The following examples demonstrate how to use the Object Query methods of the Customer MBO to retrieve data.

The following method retrieves all customers.

```
public static com.sybase.collections.ObjectList findAll()

com.sybase.collections.ObjectList customers = Customer.findAll();
```

The following method retrieves all customers in a certain page.

```
public static com.sybase.collections.ObjectList findAll(int skip,
int take)

com.sybase.collections.ObjectList customers = Customer.findAll(10,
5);
```

Suppose the modeler defined the following Object Query:

- name: findByFirstName
- parameter: String firstName
- query definition: SELECT x.* FROM Customer x WHERE x.fname = :firstName
- return type: List

The preceding Object Query results in this generated method:

```
public static com.sybase.collections.ObjectList
findByFirstName(String firstName)

com.sybase.collections.ObjectList customers =
Customer.findByFirstName(“fname”);
```

Query and Related Classes

The following classes define arbitrary search methods and filter conditions, and provide methods for combining test criteria and dynamically querying result sets.

Table 2. Query and Related Classes

Class	Description
Query	Defines arbitrary search methods and can be composed of search conditions, object/row state filter conditions, and data ordering information.
AttributeTest	Defines filter conditions for MBO attributes.
CompositeTest	Contains a method to combine test criteria using the logical operators AND, OR, and NOT to create a compound filter.
QueryResultSet	Provides for querying a result set for the dynamic query API.

In addition queries support select, where, and join statements.

Arbitrary Find

The arbitrary find method provides custom device applications the ability to dynamically build queries based on user input.

In addition to allowing for arbitrary search criteria, the arbitrary find method lets the user specify the ordering of the results and object state criteria. A `Query` class is included in the client object API's core classes. The `Query` class is the single object passed to the arbitrary search methods and consists of search conditions, object/row state filter conditions, and data ordering information.

Define these conditions by setting properties in a query:

- **TestCriteria –** criteria used to filter returned data.
- **SortCriteria –** criteria used to order returned data.
- **Skip –** an integer specifying how many rows to skip. Used for paging.
- **Take –** an integer specifying the maximum number of rows to return. Used for paging.

`TestCriteria` can be an `AttributeTest` or a `CompositeTest`.

Dynamic Query

You can construct a query SQL statement to query data from a local database. This query may across multiple tables (MBOs).

```
Query query2 = new Query();
query2.select("c.fname,c.lname,s.order_date,s.region");
```

```
query2.from("Customer", "c");
//
// Convenience method for adding a join to the query
// Detailed construction of the join criteria
query2.join("Sales_order", "s", "c.id", "s.cust_id");
AttributeTest ts = new AttributeTest();
ts.setAttribute("fname");
ts.setTestValue("Beth");
query2.where(ts);
QueryResultSet qrs = SampleAppDB.executeQuery(query2);
```

Note: A wildcard is not allowed in the `select` clause. You must use explicit column names.

SortCriteria

`SortCriteria` defines a list of `SortOrder`, which contains an attribute name and an order type (ASCENDING or DESCENDING).

For example, locate all Customer objects based on the following criteria:

- FirstName = 'John' AND LastName = 'Doe' AND (State = 'CA' or State = 'NY')
- Customer is New or Updated
- Ordered by: LastName ASC, FirstName ASC, Credit DESC
- Skip the first 10 and take 5

Use code similar to:

```
 Query props = new Query();
        //define the attribute based conditions
        CompositeTest innerCompTest = new CompositeTest();
        innerCompTest.setCompositionType(TestType.OR);
        innerCompTest.add (
        new AttributeTest ("state", "CA", AttributeTest.EQUAL));
        innerCompTest.add (
        new AttributeTest ("state", "NY", AttributeTest.EQUAL));
        CompositeTest outerCompTest = new CompositeTest();
        outerCompTest.setCompositionType(CompositeTest.AND);
        outerCompTest.add (
        new AttributeTest("fname", "John", AttributeTest.EQUAL));
        outerCompTest.add (
        new AttributeTest("lname", "Doe" ,AttributeTest.EQUAL));
        outerCompTest.add (innerCompTest);
        //define the ordering
        SortCriteria sort = new SortCriteria();
        sort.add ("lname", SortOrderType.ASCENDING);
        sort.add ("fname", SortOrderType.ASCENDING);
        sort.add ("id", SortOrderType.DESCENDING);
        //set the Query object
        props.setTestCriteria(outerCompTest);
        props.setSortCriteria(sort);
        props.setSkip(10);
        props.setTake(5);
        props.setStateCriteria(ObjectState.NEW |
ObjectState.UPDATED);
        com.sybase.collections.ObjectList customers =
Customer.findWithQuery(props);
```

Paging Data

On low-memory devices, retrieving up to 30,000 records from the database may cause the custom client to fail and throw an OutOfMemoryException.

Consider using the Query object to limit the result set:

```
Query props = new Query();
props.setSkip(10);
props.setTake(5);

com.sybase.collections.ObjectList customers =
Customer.findWithQuery(props);
```

AttributeTest

An `AttributeTest` defines a filter condition using an MBO attribute, and supports conditions.

- IS_NULL
- NOT_NULL
- EQUAL
- NOT_EQUAL
- LIKE
- NOT_LIKE
- MATCH
- NOT_MATCH
- LESS_THAN
- LESS_EQUAL
- GREATER_THAN
- GREATER_EQUAL
- CONTAINS
- STARTS_WITH
- ENDS_WITH
- DOES_NOT_START_WITH
- DOES_NOT_END_WITH
- DOES_NOT_CONTAIN

CompositeTest

A `CompositeTest` combines multiple `TestCriteria` using the logical operators AND, OR and NOT to create a compound filter.

The following example retrieves all log records where mboName=entityName and key=idString:

```
String entityName = "Customer";
        String idString = "12345";
        com.sybase.persistence.Query query = new
```

```
        com.sybase.persistence.Query();
        com.sybase.persistence.CompositeTest ct = new
        com.sybase.persistence.CompositeTest();
        ct.setOperator(com.sybase.persistence.CompositeTest.AND);

ct.add(com.sybase.persistence.AttributeTest.equal("component",
        entityName));

ct.add(com.sybase.persistence.AttributeTest.equal("entityKey",idStr
ing));
        query.setTestCriteria(ct);
        com.sybase.collections.ObjectList logList =
        LogRecordImpl.findWithQuery(query);
```

QueryResultSet

The `QueryResultSet` class provides for querying a result set for the dynamic query API. `QueryResultSet` is returned as a result of executing a query.

Example

The following example shows how to execute a query on multiple MBOs using a join:

```
com.sybase.persistence.Query query = new
com.sybase.persistence.Query();

query.select("c.fname,c.lname,s.order_date,s.region");
query.from(" Customer ", "c");
query.join(" SalesOrder ", "s", " s.cust_id ", "c.id");
AttributeTest ts = new AttributeTest();
ts.setAttribute("lname");
ts.setTestValue(" Devlin");
ts.setOperator(AttributeTest.EQUALS)
query.setTestCriteria(ts);
QueryResultSet qrs = <MyPkg>DB.executeQuery(query);
while(qrs.next())
{
     System.out.println(qrs.getString(columnIndex));
     System.out.println(qrs.getStringByName(columnName));
}
```

Retrieving Relationship Data

A relationship between two MBOs allows the parent MBO to access the associated MBO.

Assume there are two MBOs defined in Unwired Server. One MBO is called Customer and contains a list of customer data records. The second MBO is called SalesOrder and contains order information. Additionally, assume there is an association between Customers and Orders on the customer ID column. The Orders application is parameterized to return order information for the customer ID.

```
Customer customer = Customer.findById (101);
com.sybase.collections.ObjectList orders =
customer.getSalesOrders();
```

You can also use the Query class to filter the return MBO list data.

```
Query props = new Query();
// set query parameters
......
com.sybase.collections.ObjectList orders =
customer.getSalesOrdersFilterBy(props);
```

Operations APIs

Mobile business object operations are performed on an MBO instance. Operations in the model that are marked as create, update, or delete (CUD) operations create instances (non-static) of operations in the generated client-side objects.

Any parameters in the create, update, or delete operation that are mapped to the object's attributes are handled internally by the Client Object API, and are not exposed. Any parameters not mapped to the object's attributes are left as parameters in the Generated Object API. The code examples for create, update and delete operations are based on the **fill from attribute** being set. Different MBO settings will effect operation methods.

Note: If the Sybase Unwired Platform object model defines one instance of a create operation and one instance of an update operation, and all operation parameters are mapped to the object's attributes, then a `Save` method can be automatically generated which, when called internally, determines whether to insert or update data to the local client-side database. In other situations, where there are multiple instances of create or update operations, it is not possible to automatically generate such a Save method.

Create Operation

To execute a create operation on an MBO, create a new MBO instance, set the MBO attributes, then call the `save()` or `create()` operation.

```
Customer cust = new Customer();
cust.setFname ( "supAdmin" );
cust.setCompany_name( "Sybase" );
cust.setPhone( "777-8888" );
cust.create();// or cust.save();
cust.submitPending();
<PkgName>DB.synchronize();
// or <PkgName>DB.synchronize (String synchronizationGroup)
```

Update Operation

To execute update operations on an MBO, get an instance of the MBO, set the MBO attributes, and then call either the `save()` or `update()` operations.

```
Customer cust = Customer.findById(101);
cust.setFname("supAdmin");
cust.setCompany_name("Sybase");
cust.setPhone("777-8888");
cust.save();
cust.submitPending();
<PkgName>DB.synchronize();
// or <PkgName>DB.synchronize (String synchronizationGroup)
```

To update multiple MBOs in a relationship, call `submitPending()` on the parent MBO, or call `submitPending()` on the changed child MBO:

```
Customer cust = Customer.findById(101);
com.sybase.collections.ObjectList orders = cust.getSalesOrders();
SalesOrder order = (SalesOrder)orders.getByIndex(0);
order.setOrder_date(new java.util.Date());
order.save();
cust.submitPending();
```

Delete Operation

To execute delete operations on an MBO, get an instance of the MBO, set the MBO attributes, then call the `delete()` operation.

```
Customer cust = Customer.findById(101);
cust.delete();
```

For MBOs in a relationship, perform a delete as follows:

```
 Customer cust = Customer.findById(101);
        com.sybase.collections.ObjectList orders =
cust.getSalesOrders();
        SalesOrder order = (SalesOrder)orders.getByIndex(0);
        order.delete();
        cust.submitPending();
<PkgName>DB.synchronize();
// or <PkgName>DB.synchronize (String synchronizationGroup)
```

Save Operation

When called, the Save method determines internally if it should insert or update data to the client database.

```
//Update an existing customer
Customer cust = Customer.findById(101);
cust.save();

//Insert a new customer
Customer cust = new Customer();
cust.save();
```

Other Operation

Operations that are not create, update, or delete operations are called “Other” operations. An Other operation class is generated for each operation in the MBO that is not a create, update or delete operation.

Suppose the Customer MBO has an Other operation “other”, with parameters “P1” (string), “P2” (int) and “P3” (date). This results in a `CustomerOtherOperation` class being generated, with “P1”, “P2” and “P3” as its attributes.

To invoke the Other operation, create an instance of `CustomerOtherOperation`, and set the correct operation parameters for its attributes. This code provides an example:

```
CustomerOtherOperation other = new CustomerOtherOperation();
other.setP1("somevalue");
other.setP2(2);
other.setP3(new Date());
other.save(); // or other.create()
other.submitPending();
<PkgName>DB.synchronize(); // or <PkgName>DB.synchronize (String
synchronizationGroup)
```

Multilevel Insert

Multilevel insert allows a single synchronization to execute a chain of related insert operations.

Consider creating a Customer and a new Customer order at the same time on the client side, where the SalesOrder has a reference to the new Customer identifier. The following example demonstrates a multilevel insert:

```
Customer customer = new Customer();
customer.setFname("firstName");
customer.setLname("lastName");
customer.setPhone("777-8888");
customer.save();
SalesOrder order = new SalesOrder();
order.setCustomer(customer);
order.setOrder_date(new java.util.Date());
order.setRegion("Eastern");
order.setSales_rep(102);
customer.getOrders().add(order);
//Both the child and parent MBO must call save()
order.save();
//Must submit parent
...
```

To insert an order for an existing customer, first find the customer, then create a sales order with the customer ID retrieved:

```
Customer customer = Customer.findById(101);
SalesOrder order = new SalesOrder();
order.setCustomer(customer);
order.setOrder_date(new java.util.Date());
order.setRegion("Eastern");
order.setSales_rep(102);
customer.getSalesOrders().add(order);
order.save();
customer.submitPending();
```

See the Sybase Unwired Platform online documentation for specific multilevel insert requirements.

Pending Operation

You can manage pending operations using these methods:

- **cancelPending –** cancels the previous create, update, or delete operations on the MBO. It cannot cancel submitted operations.

- **submitPending –** submits the operation so that it can be replayed on the Unwired Server. A request is sent to the Unwired Server during a synchronization.
- **submitPendingOperations –** submits all the pending records for the entity to the Unwired Server. This method internally invokes the `submitPending` method on each of the pending records.
- **cancelPendingOperations –** cancels all the pending records for the entity. This method internally invokes the `cancelPending` method on each of the pending records.

Note: Use the `SubmitPendingOperations` and `CancelPendingOperations` methods only when there are multiple pending entities on the same MBO type. Otherwise, use the MBO instance's `SubmitPending` or `CancelPending` methods, which are more efficient if the MBO instance is already available in memory.

```
Customer customer = Customer.findById(101);
if (errorHappened) {
    customer.cancelPending();
}
else {
    customer.submitPending();
}
```

Passing Structures to Operations

Structures hold complex datatypes (for example a string list, class or MBO object, or a list of objects) that enhance interactions with certain enterprise information systems (EIS) data sources, such as SAP and Web services, where the mobile business object (MBO) requires complex operation parameters.

An Unwired WorkSpace project includes an example MBO that is bound to a Web service data source that includes a create operation that takes a structure as an operation parameter. MBOs differ depending on the data source, configuration, and so on, but the principles are similar.

The SimpleCaseList MBO contains a create operation that has a number of parameters, including a parameter named _HEADER_ that is a structure datatype named AuthenticationInfo, defined as:

```
AuthenticationInfo
   userName: String
   password: String
   authentication: String
   locale: String
   timeZone: String
```

Structures are implemented as classes, so the parameter _HEADER_ is an instance of the AuthenticationInfo class. The generated Java code for the create operation is:

```
public  void create(complex.AuthenticationInfo
_HEADER_,java.lang.String escalated,java.lang.String
hotlist,java.lang.String orig_Submitter,java.lang.String
pending,java.lang.String workLog)
```

This example demonstrates how to initialize the AuthenticationInfo class instance and pass them, along with the other operation parameters, to the create operation:

```
AuthenticationInfo authen = new AuthenticationInfo();
        authen.setUserName("Demo");
        authen.setPassword("");
        authen.setAuthentication("");
        authen.setLocale("EN_US");
        authen.setTimeZone("GMT");

        SimpleCaseList newCase = new SimpleCaseList();
        newCase.setCase_Type("Incident");
        newCase.setCategory("Networking");
        newCase.setDepartment("Marketing");
        newCase.setDescription("A new help desk case.");
        newCase.setItem("Configuration");
        newCase.setOffice("#3 Sybase Drive");
        newCase.setSubmitted_By("Demo");
        newCase.setPhone_Number("#0861023242526");
        newCase.setPriority("High");
        newCase.setRegion("USA");
        newCase.setRequest_Urgency("High");
        newCase.setRequester_Login_Name("Demo");
        newCase.setRequester_Name("Demo");
        newCase.setSite("25 Bay St, Mountain View, CA");
        newCase.setSource("Requester");
        newCase.setStatus("Assigned");
        newCase.setSummary("MarkHellous was here Fix it.");
        newCase.setType("Access to Files/Drives");
        newCase.setCreate_Time(new
        java.sql.Timestamp(System.currentTimeMillis()));

        newCase.create(authen, "Other", "Other", "Demo", “false”,
“worklog”);
        newCase.submitPending();
```

Mobile and Local Business Objects

A business object can be either local or mobile. A local business object is a client only object, and is represented by the `LocalBusinessObject` interface. A mobile business object can be synchronized with the Unwired Server, and is represented by the `MobileBusinessObject` interface.

Both `LocalBusinessObject` and `MobileBusinessObject` extend `BusinessObject`. `MobileBusinessObject` provides the following additional methods:

```
public interface MobileBusinessObject extends BusinessObject
{
    void cancelPending();
    LogRecord[] getLogRecords();
    boolean isCreated();
    boolean isPending();
    boolean isUpdated();
    void submitPending();
}
```

`getLogRecords` returns operation logs as `LogRecord` instances. See the LogRecord API.

`submitPending` submits a pending record to the Unwired Server. A pending record is one that has been updated in the client database, but not sent to the Unwired Server.

`cancelPending` cancels a pending record.

Personalization APIs

Personalization keys allow the application to define certain input parameter values that differ (are personalized) for each mobile user. The Personalization APIs allow you to manage personalization keys, and get and set personalization key values.

Type of Personalization Keys

There are three types of personalization keys: client, server, and transient (or session). Client personalization keys are persisted in the local database. Server personalization keys are persisted on the Unwired Server. Session personalization keys are not persisted and are lost after the device application terminates.

A personalization parameter can be a primitive or complex type. This is shown in the code example.

Get or Set Personalization Key Values

The `PersonalizationParameters` class is generated automatically for managing personalization keys. Personalization keys allow the application to define certain input parameter values that are different (personalized) for each mobile user.

The following code provides an example on how to set a personalization key, and pass an array of values and array of objects:

```
PersonalizationParameters pp =
<PkgName>DB.getPersonalizationParameters();
pp.setMyIntPK(10002);
pp.save();
IntList il = new IntList(2);
il.add(10001);
il.add(10002);
pp.setMyIntListPK(il);
pp.save();

MyDataList dl = new MyDataList();
//MyData is a structure type defined in tooling
MyData md = new MyData();
md.setIntMember( ... );
md.setStringMember2( ... );
dl.add(md);
pp.setMyDataList( dl );
pp.save();
```

If a synchronization parameter is personalized, you can overwrite the value of that parameter with the personalization value.

Object State APIs

The object state APIs provide methods for returning information about the state of an entity in an application.

Entity State Management

The object state APIs provide methods for returning information about entities in the database. All entities that support pending state have the following attributes:

Name	Java Type	Description
isNew	boolean	Returns true if this entity is new (but has not been created in the client database).
isCreated	boolean	Returns true if this entity has been newly created in the client database, and one the following is true: • The entity has not yet been submitted to the server with a replay request. • The entity has been submitted to the server, but the server has not finished processing the request. • The server rejected the replay request (replayFailure message received).
isDirty	boolean	Returns true if this entity has been changed in memory, but the change has not yet been saved to the client database.
isDeleted	boolean	Returns true if this entity was loaded from the database and was subsequently deleted.
isUpdated	boolean	Returns true if this entity has been updated or changed in the database, and one of the following is true: • The entity has not yet been submitted to the server with a replay request. • The entity has been submitted to the server, but the server has not finished processing the request. • The server rejected the replay request (replayFailure message received).
pending	boolean	Returns true for any row that represents a pending create, update, or delete operation, or a row that has cascading children with a pending operation.

Name	Java Type	Description
pendingChange	char	If pending is true, then 'C' (create), 'U' (update), 'D' (delete), 'P' (to indicate that this MBO is a parent in a cascading relationship for one or more pending child objects, but this MBO itself has no pending create, update or delete operations). If pending is false, then 'N'.
replayCounter	long	Returns a long value which is updated each time a row is created or modified by the client. This value is derived from the time in seconds since an epoch, and increases each time a row is changed.
replayPending	long	Returns a long value. When a pending row is submitted to the server, the value of `replayCounter` is copied to `replayPending`. This allows the client code to detect if a row has been changed since it was submitted to the server (that is, if the value of`replayCounter` is greater than `replayPending`).
replayFailure	long	Returns a long value. When the server responds with a `replayFailure` message for a row that was submitted to the server, the value of `replayCounter` is copied to `replayFailure`, and `replayPending` is set to 0.

Pending State Pattern

When a create, update, delete, or save operation is called on an entity in a replication-based synchronization application, the requested change becomes pending. To apply the pending change, call `submitPending` on the entity, or `submitPendingOperations` on the MBO class:

```
  Customer e = new Customer();
e.setFname("Fred");
e.setAddress("123 Four St.");
e.create(); // create as pending
e.submitPending(); // submit to server
Customer.submitPendingOperations(); // submit all pending Customer
rows to server
```

`submitPendingOperations` submits all the pending records for the entity to the Unwired Server. This method internally invokes the `submitPending` method on each of the pending records.

The call to `submitPending` causes the operations to be marked for replay by Unwired Server. On the next synchronization, Unwired Server processes the operations and creates log records for each operation with code indicating the status of the operation. The `LogRecord` interface is defined as follows:

Method Name	Java Type	Description
component	string	Name of the MBO for the row for which this log record was written.
entityKey	string	String representation of the primary key of the row for which this log record was written.
code	int	One of several possible HTTP error codes: • 200 indicates success. • 401 indicates that the client request had invalid credentials, or that authentication failed for some other reason. • 403 indicates that the client request had valid credentials, but that the user does not have permission to access the requested resource (package, MBO, or operation). • 404 indicates that the client tried to access a nonexistent package or MBO. • 405 indicates that there is no valid license to check out for the client. • 500 to indicate an unexpected (unspecified) server failure.
message	String	Descriptive message from the server with the reason for the log record.
operation	String	The operation (create, update, or delete) that caused the log record to be written.
requestId	String	The id of the replay message sent by the client that caused this log record to be written.
timestamp	Date	Date and time of the log record.

If a rejection is received, the application can use the entity method `getLogRecords` to access the log records and get the reason:

```
  com.sybase.collections.ObjectList logs = e.getLogRecords();
for(int i=0; i<logs.count(); i++)
{
com.sybase.persistence.LogRecord log =
(com.sybase.persistence.LogRecord)logs.getByIndex(i);
System.out.println("Entity has a log record:");
System.out.println("Code = " + log.getCode());
System.out.println("Component = " + log.getComponent());
System.out.println("EntityKey = " + log.getEntityKey());
System.out.println("Level = " + log.getLevel());
System.out.println("Message = " + log.getMessage());
System.out.println("Operation = " + log.getOperation());
```

```
System.out.println("RequestId = " + log.getRequestId());
System.out.println("Timestamp = " + log.getTimestamp());
}
```

`cancelPendingOperations` cancels all the pending records for an entity. This method internally invokes the `cancelPending` method on each of the pending records.

Mobile Business Object States

A mobile business object can be in one of three states:

- Original state, the state before any CUD operation.
- Downloaded state, the state downloaded from the Unwired Server.
- Current state, the state after any CUD operation.

The Mobile Business Object class provides properties for querying the original state and the downloaded state:

```
public Customer getOriginalState();
public Customer getDownloadState();
```

The original state is valid only before the application synchronizes with the Unwired Server. After synchronization has completed successfully, the original state is cleared and set to null.

```
Customer cust = Customer.findById(101);        // state 1
cust.setFname("firstName");
cust.setCompany_name("Sybase");
cust.setPhone("777-8888");
cust.save();                                   // state 2
Customer org = cust.getOriginalState();        // state 1
//suppose there is new download for Customer 101 here
Customer download = cust.getDownloadState();   // state 3
cust.cancelPending();                          // state 3
```

Using all three states, the application can resolve most conflicts that may occur.

Refresh Operation

The refresh operation of an MBO allows you to refresh the MBO state from the client database.

The following code provides an example:

```
Customer cust = Customer.findById(101);
cust.setFname("newName");
cust.refresh();// newName is discarded
```

Clear Relationship Objects

The `clearRelationshipObjects` method releases relationship attributes and sets them to null. Attributes get filled from the client database on the next getter method call or property reference. You can use this method to conserve memory if an MBO has large child attributes that are not needed at all times.

clearRelationshipObjects

Common APIs

In addition to Object State APIs these APIs are available with each mobile business object.

- **save –** save a record to the local database, In the case of an existing record, save calls update. In the case of a new record, save calls create.
- **refresh –** client refreshes the entity from the local database.
- **cancelPending –** cancels a pending record.
- **submitPending –** submits a pending record to the server.
- **getPendingChange –** if pending is true, then 'C' (create), 'U' (update), 'D' (delete), 'P' (to indicate that this row is a parent in a cascading relationship for one or more pending child objects, but this row itself has no pending create, update or delete operations). If pending is false, then 'N'.
- **getReplayCounter –** updated each time a row is created or modified by the client. This value is derived from the time in seconds since an epoch, so it always increases each time the row is changed.
- **getReplayPending –** when a pending row is submitted to the server, the value of replayCounter is copied to replayPending. This allows client code to detect if a row has been changed since it was submitted to the server --the test to look for : replayCounter > replayPending. On receiving a successful response (replayResult) from the server, this is reset to 0.
- **getReplayFailure –** when the server responds with a replayFailure message for a row that was submitted to the server, the replayCounter value is copied to replayFailure, and replayPending is set to 0.

Security APIs

The security APIs allow you to customize some aspects of connection and database security.

Connect Using a Certificate

You can set certificate information in `ConnectionProfile`.

```
ConnectionProfile profile = <PkgName>DB.getSynchronizationProfile();
profile.setDomainName( "default" );
profile.setServerName( "host-name" );
profile.setPortNumber( 2481 );
profile.setNetworkProtocol( "https" );
```

Install the certificate to BlackBerry:

- Simulator: copy the certificate to the simulator directory.
- Physical device: use the Desktop Manager Certificate Synchronization tool to import an HTTPS public certificate from the PC to the device. Then perform a synchronization with Unwired Server by HTTPS.

Encrypt the Database

You can use `ConnectionProfile.EncryptionKey` to set the encryption key of a local database. Set the key during application initialization, and before creating or accessing the client database.

The length of the encyption key must not be less than 16 characters.

```
ConnectionProfile profile = <PkgName>DB.getConnectionProfile();
profile.setEncryptionKey("Your key");
```

Installing X.509 Certificates on BlackBerry Simulators and Devices

Install the .p12 certificate on the BlackBerry device or simulator and select it during authentication.

1. Install the certificate on a device:
   a) Connect to the device with a USB cable.
   b) Browse to the SD Card folder on the computer to which the device is connected.
   c) Navigate to and select the certificate. Enter the password.
   d) Import the certificate.

   You can also use the BlackBerry Desktop Manager to intstall the certificate on the device, but you may need to perform a custom installation to access the Synchronize Certificates option.
2. Install the certificate on a simulator:
   a) From the simulator, select **Simulate > Change SD Card**.
   b) Add/or select the directory that contains the certificate.
   c) Open the media application on the device, and select **Menu > Explore**.
   d) Navigate to and select the certificate. Enter the password.
   e) Check the certificate and select **Menu > Import Certificate**.

BlackBerry Sample Code

This sample code illustrates importing the certificate and setting up login credentials, as well as other APIs related to certificate handling:

```
/// End2EndDB is a generated RBS class
///First install certificates on your simulator, for example
"Sybase101.p12"


//Test getting certificate from certificat store
CertificateStore myStore =
CertificateStore.getDefault();
String filter1 = "Sybase";
StringList labels = myStore.certificateLabels(filter1, null);
String aLabel = labels.item(0);
LoginCertificate lc = myStore.getSignedCertificate(aLabel,
"password");
```

```
// Save the login certificate to your synchronization profile
End2EndDB.getSynchronizationProfile().setCertificate(lc);

// Login to and synchronize with Unwired Server
End2EndDB.loginToSync();
End2EndDB.synchronize();

// Save the login certificate to your data vault
// The vault must be unlocked before saving
String vaultName = "myVault";
DataVault vault = null;
if(!DataVault.vaultExists(vaultName))
{
    vault = DataVault.createVault(vaultName, "password", "salt");
}
else
{
    vault = DataVault.getVault(vaultName);
}
vault.unlock("password", "salt");
lc.save("myLabel", vault);

//test loading and deleting certificate
LoginCertificate newLc = LoginCertificate.load("myLabel", vault);
LoginCertificate.delete("myLabel", vault);
```

Single Sign-On With X.509 Certificate Related Object API

Use these classes and attributes when developing mobile applications that require X.509 certificate authentication.

- CertificateStore class - wraps platform-specific key/certificate store class, or file directory
- LoginCertificate class - wraps platform-specific X.509 distinguished name and signed certificate
- ConnectionProfile class - includes the certificate attribute used for Unwired Server synchronization.

Refer to the Javadocs that describe implementation details.

Importing a Certificate Into the Data Vault

Obtain a certificate reference and store it in a password protected data vault to use for X.509 certificate authentication.

The difference between importing a certificate from a system store or a file directory is determined by how you obtain the `CertificateStore` object. In either case, only a label and password are required to import a certificate.

```
// Obtain a reference to the certificate store
CertificateStore certStore = CertificateStore.getDefault();

// Obtain a list of certificates
StringList labels = certStore.certificateLabels();
```

```
// Import a certificate from store (into memory)
String label = ...; // ask user to select a label
String password = ...; // ask the user for a password
LoginCertificate cert = certStore.getSignedCertificate(label,
password);

// Lookup or create data vault
String vaultPassword = ...; // ask user or from O/S protected storage
String vaultName = "..."; // e.g. "SAP.CRM.CertificateVault"
String vaultSalt = "..."; // e.g. a hard-coded random GUID
DataVault vault;
try
{
    vault = DataVault.getVault(vaultName);
    vault.unlock(vaultPassword, vaultSalt);
}
catch (DataVaultException ex)
{
    vault = DataVault.createVault(vaultName, vaultPassword,
vaultSalt);
}

// Save certificate into data vault
cert.save("myCert", vault);
```

Selecting a Certificate for Unwired Server Connections

Select the X.509 certificate from the data vault for Unwired Server authentication.

```
LoginCertificate cert = LoginCertificate.load("myCert", vault);
ConnectionProfile syncProfile =
MyDatabase.getSynchronizationProfile();
syncProfile.setCertificate(cert);
```

Connecting to Unwired Server With a Certificate

Once the certificate property is set, use the `onlineLogin()` API with no parameters (do not use the `onlineLogin()` API with username and password).

```
MyPackage_MyPackageDB onlineLogin();

// Handle login response

MyPackage_MyPackageDB subscribe;
```

Utility APIs

The Utility APIs allow you to customize aspects of logging, callback handling, and generated code.

LogRecord API

`LogRecord` is used to store two types of logs.

- Operation logs on the Unwired Server. These logs can be downloaded to the device.
- Client logs. These logs can be uploaded to the Unwired Server.

The following example code executes an update operation and examines the log records for the Customer MBO:

```
int id = 101;
Customer result = Customer.findById(id);
result.setFname("newFname");
result.save();
result.submitPending();
<PkgName>DB.synchronize();
result = Customer.findById(id);
com.sybase.collections.ObjectList logs = result.getLogRecords();
for( iint i=0 ; i<logs.count(); i++)
{
com.sybase.persistence.LogRecord log = logs.getByIndex(i);
System.out.println("Message: " + log.getMessage());
System.out.println("Component: " + log.getComponent());
System.out.println("Operation: " + log.getOperation());
System.out.println("Timestamp: " + log.getTimestamp());
...
}
```

Viewing Error Codes in Log Records

You can view any EIS error codes and the logically mapped HTTP error codes in the log record.

For example, you could observe a "Backend down" or "Backend login failure" after the following sequence of events:

1. Deploying packages to Unwired Server.
2. Performing an initial synchronization.
3. Switching off the backend or change the login credentials at the backend.
4. Invoking a create operation by sending a JSON message.

```
JsonHeader
{"id":"684cbe16f6b740eb930d08fd626e1551","cid":"111#My1:1","ppm":
"eyJ1c2VybmFtZSI6InN1cEFkbWluIiwicGFzc3dvcmQiOiJzM3BBZG1pbiJ9","p
id":"moca://
Emulator17128142","method":"replay","pbi":"true","upa":"c3VwQWRta
W46czNwQWRtaW4=","mbo":"Bi","app":"My1:1","pkg":"imot1:1.0"}

JsonContent
{"c2":null,"c1":1,"createCalled":true,"_op":"C"}
```

The Unwired Server returns a response. The code is included in the `ResponseHeader`.

```
ResponseHeader
{"id":"684cbe16f6b740eb930d08fd626e1551","cid":"111#My1:1","loginFa
```

```
iled":false,"method":"replayFailed","log":
[{"message":"com.sybase.jdbc3.jdbc.SybSQLException:SQL Anywhere
Error -193: Primary key for table 'bi' is not unique : Primary key
value ('1')","replayPending":
0,"eisCode":"","component":"Bi","entityKey":"0","code":
500,"pending":false,"disableSubmit":false,"?":"imot1.server.LogReco
rdImpl","timestamp":"2010-08-26
14:05:32.97","requestId":"684cbe16f6b740eb930d08fd626e1551","operat
ion":"create","_op":"N","replayFailure":
0,"level":"ERROR","pendingChange":"N","messageId":200001,"_rc":
0}],"mbo":"Bi","app":"My1:1","pkg":"imot1:1.0"}

ResponseContent
{"id":100001}
```

Logging APIs

Retrieve client log records.

```
//To fill out the deleted and submitted log records
            AttributeTest attributeTestNotDeleted = new
AttributeTest(LogConfig.ReplayPending/*"replayPending"*/,
LogConfig.DefaultReplayPendingValue/*"0"*/, AttributeTest.EQUAL);

q.setTestCriteria(AttributeTest.isNull("operation").and(attributeTe
stNotDeleted));
```

```
package com.sybase.persistence;

/**
  * The interface for the logger. Used to create log record.
  */
public interface Logger
{
    /**
     * Get current log level
     */
    public int getLogLevel();
    /**
     * Set current log level
     */

    public void setLogLevel(int newLevel);

    /**
     * Create a new log record
     * @param level The log level of the new log record
     * @param message The log message of the new log record
     */
    public LogRecord newLogRecord(int level, String message);

    /**
     * Create a fatal log
     * @param message The log message of the new log record
     */
    public void fatal(String message);
```

```
    /**
     * Create an error log
     * @param message The log message of the new log record
     */
    public void error(String message);

    /**
     * Create a warn log
     * @param message The log message of the new log record
     */
    public void warn(String message);

    /**
     * Create an info log
     * @param message The log message of the new log record
     */
    public void info(String message);

    /**
     * Create a debug log
     * @param message The log message of the new log record
     */
    public void debug(String message);

    /**
     * Create a trace log
     * @param message The log message of the new log record
     */
    public void trace(String message);
}
```

Callback Handlers

To receive callbacks, you must register a `CallBackHandler` with the generated database class, the entity class, or both. You can create a handler by extending the `DefaultCallbackHandler` class.

In your handler, override the particular callback that you are interested in (for example, `OnReplayFailure`). The callback is executed in the thread that is performing the action (for example, replay). When you receive the callback, the particular activity is already complete. The `CallbackHandler` interface consists of the following callbacks:

Table 3. Callbacks in the CallbackHandler Interface

Callback	Description
void onReplayFailure(java.lang.Object entity)	Replay failure response notification. *entity* is a client MBO instance.
void onReplaySuccess(java.lang.Object entity)	Replay success response notification. *entity* is a client MBO instance.

Callback	Description
int onSynchronize(com.sybase.collections.ObjectList groups,SynchronizationContext context)	This method will be invoked at the specified status of the synchronization.*groups* is a list of synchronization group names. *context* is the synchronization context.

The following code example shows how to create and register a handler to receive callbacks:

```
public class MyCallbackHandler extends DefaultCallbackHandler
{
    // implementation
}

CallbackHandler handler = new MyCallbackHandler();
<PkgName>DB.registerCallbackHandler(handler);
//or Customer.registerCallbackHandler(handler);
```

SyncStatusListener API

You can implement a synchronization status listener to track the progress of synchronization.

Create a listener that implements the SyncStatusListener interface as follows:

```
public interface SyncStatusListener
{
    boolean objectSyncStatus(ObjectSyncStatusData statusData);
}

public class MySyncListener extends SyncStatusListener
{
// implementation
}
```

Pass an instance of the listener to the synchronize methods as follows:

```
MySyncListener listener = new MySyncListener();
<PkgName>DB.synchronize("sync_group", listener);
// or <PkgName>DB.synchronize(listener); if we want to synchronize
all
// synchronization groups
```

As the application synchronization progresses, the `objectSyncStatus` method defined by the `SyncStatusListener` interface is called and is passed an `ObjectSyncStatusData` object. The `ObjectSyncStatusData` object contains information about the MBO being synchronized, the connection to which it is related, and the current state of the synchronization process. By testing the `State` property of the `ObjectSyncStatusData` object and comparing it to the possible values in the `SyncStatusState` enumeration, the application can react accordingly to the state of the synchronization.

Possible uses of `objectSyncStatus` method include changing form elements on the client screen to show synchronization progress, such as a green image when the

synchronization is in progress, a red image if the synchronization fails, and a gray image when the synchronization has completed successfully and disconnected from the server.

Note: The `objectSyncStatus` method of `SyncStatusListener` is called and executed in the data synchronization thread. If a client runs synchronizations in a thread that is not the primary user interface thread, the client cannot update its screen as the status changes. In that case, the client must instruct the primary user interface thread to update the screen regarding the current synchronization status.

The following is an example of `syncStatusListener` implementation:

```
public class SyncListener extends syncStatusListener
{
  public boolean objectSyncStatus(ObjectSyncStatusData data)
  {
    switch (data.getSyncStatusState()) {
    case SyncStatusState.APPLICATION_SYNC_DONE:
      //implement your own UI indicator bar
      break;
    case SyncStatusState.APPLICATION_SYNC_ERROR:
      //implement your own UI indicator bar
      break;
    case SyncStatusState.SYNC_DONE:
      //implement your own UI indicator bar
      break;
    case SyncStatusState.SYNC_STARTING:
      //implement your own UI indicator bar
      break;
    ...
    }
    return false;
  }
}
```

isSynchronized() and getLastSynchronizationTime()

The package database class provides the following methods for querying the synchronized state and the last synchronization time of a synchronization group:

```
// Returns if the synchronizationGroup was synchronized
public static boolean isSynchronized(String synchronizationGroup)

// Returns the last synchronization time of the synchronizationGroup
public static java.util.Date getLastSynchronizationTime(String
synchronizationGroup)
```

generateId

You can use the `generateId` methods in the `LocalKeyGenerator` and `KeyGenerator` classes to generate an ID when creating a new object for which you require a primary key.

This method in the `LocalKeyGenerator` class generates a unique ID for the package on the local device:

```
public static long generateId()
```

This method in the `KeyGenerator` class generates a unique ID for the same package across all devices:

```
public static long generateId()
```

Client Database APIs

The generated package database class provides methods for managing the client database.

```
public static void createDatabase()
public static void deleteDatabase()
```

Typically, `createDatabase` does not need to be called since it is called internally when necessary. An application may use `deleteDatabase` when the client database contains corrupted data and needs to be cleared.

Exceptions

Reviewing exceptions allows you to identify where an error has occurred during application execution.

Handling Exceptions

The Client Object API defines server-side and client-side exceptions.

Server-Side Exceptions

Exceptions thrown on the Unwired Server are logged in both the server log and in `LogRecord`. For `LogRecord`, the exception gets downloaded to the device automatically during synchronization.

HTTP Error Codes

Unwired Server examines the EIS code received in a server response message and maps it to a logical HTTP error code, if a corresponding error code exists. If no corresponding code exists, the 500 code is assigned to signify either a Sybase Unwired Platform internal error, or an unrecognized EIS error. The EIS code and HTTP error code values are stored in log records.

The following is a list of recoverable and non-recoverable error codes. Beginning with Unwired Platform version 1.5.5, all error codes that are not explicitly considered recoverable are now considered unrecoverable.

Table 4. Recoverable Error Codes

Error Code	Probable Cause
409	Backend EIS is deadlocked.
503	Backend EIS down or the connection is terminated.

Table 5. Non-recoverable Error Codes

Error Code	Probable Cause	Manual Recovery Action
401	Backend EIS credentials wrong.	Change the connection information, or backend user password.
403	User authorization failed on Unwired Server due to role constraints (applicable only for MBS).	N/A
404	Resource (table/webservice/BAPI) not found on Backend EIS.	Restore the EIS configuration.
405	Invalid license for the client (applicable only for MBS).	N/A
412	Backend EIS threw a constraint exception.	Delete the conflicting entry in the EIS.
500	SUP internal error in modifying the CDB cache.	N/A

Beginning with Unwired Platform version 1.5.5, error code 401 is no longer treated as a simple recoverable error. If the `SupThrowCredentialRequestOn401Error` context variable is set to true (which is the default), error code 401 throws a `CredentialRequestException`, which sends a credential request notification to the user's inbox. You can change this default behavior by modifying the value of the `SupThrowCredentialRequestOn401Error` context variable in Sybase Control Center. If `SupThrowCredentialRequestOn401Error` is set to false, error code 401 is treated as a normal recoverable exception.

Mapping of EIS Codes to Logical HTTP Error Codes

The following is a list of SAP® error codes mapped to HTTP error codes. SAP error codes which are not listed map by default to HTTP error code 500.

Table 6. Mapping of SAP error codes to HTTP error codes

Constant	Description	HTTP Error Code
JCO_ERROR_COMMUNICATION	Exception caused by network problems, such as connection breakdowns, gateway problems, or inavailability of the remote SAP system.	503

Constant	Description	HTTP Error Code
JCO_ERROR_LOGON_FAILURE	Authorization failures during the logon phase usually caused by unknown username, wrong password, or invalid certificates.	401
JCO_ERROR_RESOURCE	Indicates that JCO has run out of resources such as connections in a connection pool	503
JCO_ERROR_STATE_BUSY	The remote SAP system is busy. Try again later	503

Client-Side Exceptions

Device applications are responsible for catching and handling exceptions thrown by the client object API.

Note: See *Callback Handlers*.

Exception Classes

The Client Object API supports exception classes for queries and for the messaging client.

- **SynchronizeException –** this exception is thrown when an error occurs during synchronization.
- **ObjectNotFoundException –** this exception is thrown when trying to load an MBO that is inside the local database.
- **NoSuchOperationException –** this exception is thrown when trying to call a method (using the Object Manager API) but the method is not defined for the MBO.
- **NoSuchAttributeException –** this exception is thrown when trying to access an attribute (using the Object Manager API) but the attribute is not defined for the MBO.

MetaData and Object Manager API

The MetaData and Object Manager API allows you to access metadata for database, classes, entities, attributes, operations, and parameters.

MetaData and Object Manager API

Some applications or frameworks can operate against MBOs generically by invoking MBO operations without prior knowledge of MBO classes. This can be achieved by using the MetaData and Object Manager APIs.

These APIs allow retrieving the metadata of packages, MBOs, attributes, operations and parameters during runtime. The APIs are especially useful for a runtime environment without a reflection mechanism such as J2ME.

You can generate metadata classes using the `-md` code generation option. You can use the `-rm` option to generate the object manager class.

The following code synchronizes and retrieves MBO data:

```
<PkgName>DB.loginToSync("username", "password");
<PkgName>DB.synchronize();
Customer cust = Customer.findById(123);
```

The following code gets the same result by using the reflection mechanism:

```
ObjectManager om = new <PkgName>DB_RM();
DatabaseMetaData dbmd = <PkgName>DB.getMetaData();
ObjectList params = new ObjectList(2);
params.add("username");
params.add("password");
om.invoke(dbmd, dbmd.getOperation("loginToSync"), params);
om.invoke(dbmd, dbmd.getOperation("synchronize"), null);
ObjectList syncParams = new ObjectList(1);
syncParams.add("default");
om.invoke(dbmd, dbmd.getOperation("synchronize",new
String[] {"string"}),syncParams);
```

ObjectManager

The `ObjectManager` class allows an application to call the Object API in a reflection style.

```
Customer object = Customer.findById(123);
ObjectManager rm = new <PkgName>DB_RM();
ClassMetaData customer =
<PkgName>DB.getMetaData().getClass("Customer");
AttributeMetaData lname = customer.getAttribute("lname");
OperationMetaData save = customer.getOperation("save");
Object myMBO = rm.newObject(customer);
rm.setValue(myMBO, lname, "Steve");
rm.invoke(object, save, new ObjectList());
```

DatabaseMetaData

The `DatabaseMetaData` class holds package level metadata. You can use it to retrieve data such as synchronization groups, default database file, and MBO metadata.

```
DatabaseMetaData dmd = <PkgName>DB.getMetaData();
com.sybase.collections.StringList syncGroups =
dmd.getSynchronizationGroups();
for(int i=0; i<syncGroups.size(); i++)
{
String syncGroup = syncGroups.getByIndex(i);
System.out.println(syncGroup);
}
```

ClassMetaData

The `ClassMetaData` class holds metadata for the MBO, including attributes and operations.

```
AttributeMetaData lname = customerMetaData.getAttribute("lname");
OperationMetaData save = customerMetaData.getOperation("save");
...
```

AttributeMetaData

The `AttributeMetaData` class holds metadata for an attribute such as attribute name, column name, type, and maxlength.

```
System.out.println(lname.getName());
System.out.println(lname.getColumn());
System.out.println(lname.getMaxLength());
```

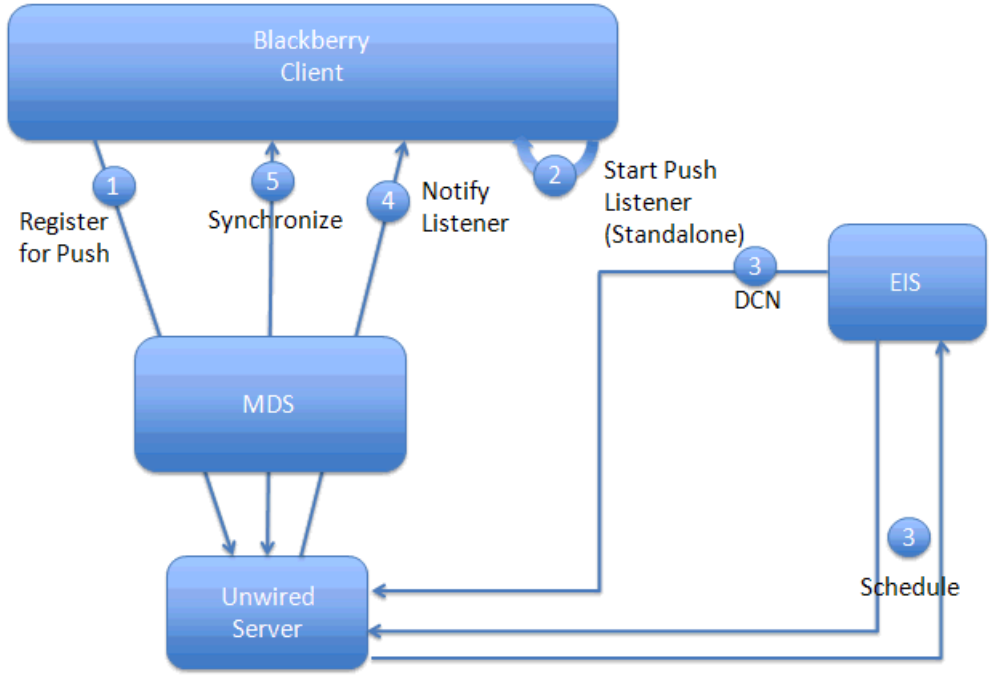
Replication-Based Push Synchronization Applications

BlackBerry devices support sending push requests through HTTP. Sybase Unwired Platform supports push configuration and notification handling APIs for BlackBerry HTTP push.

HTTP Push Gateway

Blackberry has an HTTP push feature for sending messages to occasionally connected devices. For Blackberry devices paired with BlackBerry Enterprise Server (BES), the HTTP push gateway contains an address that points to the HTTP listener of the BES server. The POST to the BES server has a query parameter that contains the device ID of the target devices (for example, 2100000a for an emulator). The BES server holds the message for a configurable amount of time, and delivers it to the device when the device becomes reachable.

The push listener runs in the background, and listens for server-initiated synchronization notifications, for example, based on a schedule or triggered by a Data Change Notification (DCN):



The HTTP push gateway can also be used for network-connected Sybase Unwired Platform applications (for example the Java desktop). The address of the subscription contains an HTTP URL to an HTTP listener which the application creates. The URL contains a query parameter such as:

```
&mode=direct
```

When the HTTP push gateway sees a query parameter without a device ID, the gateway understands that the message is not going through the BES server. For the `mode=direct` notifications to work, the application must be running and have the listener open. If the application is not running, the HTTPPush gateway reports a `ConnectionRefused` error in the log files, and the notification is not delivered.

Push Configuration APIs

The following APIs support push notification in the generated database class.

The following code example starts an HTTP push listener thread.

```
// Start the http push listener thread
pushThread = new Thread(new PushListener());
pushThread.start();
```

The client sets the SIS push configuration parameters using `SynchronizationGroup`.

```
SynchronizationGroup sg =
MyDatabase.getSynchronizationGroup("PushEnabled");
sg.setEnableSIS(true);
sg.setInterval(3);
sg.save(); // this will update the local db
```

The `synchronize()` method synchronizes the SIS subscription to the Unwired Server.

```
MyDatabase.synchronize();
System.out.println("++++ Synchronization succeeded ++++++");
```

The following code example creates the HTTP URL for push, when MDS is running on the localhost.

```
public static String getHTTPPushAddress(String deviceid)
{
  String mdsServer = MDSSERVER;

  String mdsPort = MDSSERVERPORT;

  StringBuffer result = new StringBuffer("http://");
  result.append(mdsServer);
  result.append(":");
  result.append(mdsPort);
  result.append("/push?DESTINATION=");
  result.append(deviceid);
  result.append("&PORT=");
  result.append(PUSH_HTTP_DEFAULT_DEVICE_PORT);
  return result.toString();
```

The `setPushConnectionProfile` method configures push settings for the specified package's synchronization profile.

```
private boolean setPushConnectionProfile(String packageName, String
deviceId, ConnectionProfile syncProfile, String appId)
```

Creating a Replication Based Push Application

The device application must meet these requirements to utilize the Replication-Based Push Synchronization APIs described in this section.

Develop the push application directly from generated mobile business object (MBO) code.

1. Properly configure and deploy the mobile business objects (MBOs).
   a) Create a Cache Group (or use the default) and set the cache policy to **Scheduled** and set some value for the **Cache interval**, 30 seconds for example.
   b) Create a Synchronization Group and set some value for the **Change detection level**, one minute for example.
   c) Place all Mobile Application project MBOs in the same Cache Group and Synchronization Group.
   d) Deploy the Mobile Application Project as **Replication-based** in the Deployment wizard.
2. Develop the push application.

- Develop the application directly from MBO code:
  a. Generate the Object API code.
  b. Write a push listener to listen to SIS notification sent from server

```
public class PushListener
implements Runnable
{
  Connection conn = null;

  private static String url  = "http://:
100;deviceside=false";

  /**
   * Constructor
   */
  public PushListener()
  {
  }

  public void run()
  {
     System.out.println("++++++  Started Push Listener +++++
+++");
        try
        {
           conn = Connector.open(url);
           while (true)
           {
              String syncRequestStr = null;
              try
              {
                 if ( conn instanceof
StreamConnectionNotifier )
                 {
             // Open an InputStream.
                  StreamConnectionNotifier scn =
                  (StreamConnectionNotifier) conn;
                  StreamConnection sc = scn.acceptAndOpen();
                  InputStream input = sc.openInputStream();
                  // Extract the data from the InputStream.
                  StringBuffer sb = new StringBuffer();
                  byte[] data = new byte[256];
                  int chunk = 0;
                  while (-1 != (chunk = input.read(data)))
                  {
                  sb.append(new String(data, 0, chunk));
                  }

              // Close the InputStream and StreamConnection.
                  input.close();
                  String s = sb.toString();
              // Display the received data.
                  syncRequestStr = s.trim();
                  System.out.println(">>Received: " +
syncRequestStr);
```

```
                    }
                }
                catch (Exception ex)
                {
                    System.out.println(ex);
                }

// Clients can parse the syncRequestStr to find client
application
// name, package name, sync group name(publication), launch
client
//application and perform sync.

// format of the push message sent by the server:
// notification_timestamp=<datetime>;app=<client app name>;
// device_id=<device id>;package=<sup package name with
version>;
// publication=<comma separated list of syncGroup names>

   TestDB.registerCallbackHandler(new MyCallbackHandler());
   com.sybase.collections.ObjectList sgs = new com.sybase.
   collections.ObjectList()
// Assume you have notification to sync two
syncGroups(publications),
// sg1 and sg2:
        sgs.add(TestDB.getSynchronizationGroup("sg1"));
        sgs.add(TestDB.getSynchronizationGroup("sg2"));
        TestDB.beginSynchronize(sgs, new Object());

          }
      }
      catch (Exception ex)
      {
        System.out.println("HttpPushListener - ERROR : " +
ex);
      }
  }

    /*
     * Define callback handler for handling SIS notifications
     */
    public class MyCallbackHandler extends com.sybase.
    persistence.DefaultCallbackHandler
    {
     public int onSynchronize(ObjectList arg0,
SynchronizationContext arg1)
      {
        System.out.println("Called on Synchronize");
        return SynchronizationAction.CONTINUE;
       // returns SynchronizationAction.CONTINUE to proceed
this sync
        }

        public void onSynchronizeFailure(ObjectList arg0)
        {
            System.out.println("Called
```

```
onSynchronizeFailure");
        }

        public void onSynchronizeSuccess(ObjectList arg0)
        {
            System.out.println("Called
onSynchronizeSuccess");
        }
    }
}
```

c. In the application, start the push listener, set up the connection profile for SIS and synchronize SIS subscription to server:

```
public class PushClientApp extends Application
{
 public static String MDSSERVER                = "localhost";
 public static String MDSSERVERPORT              = "8080";

  static String   PROFILE_HTTP_PUSH_PROTOCOL    = "HTTPPUSH";
  static String   PROFILE_KEY_ADDRESS           = "address";
  static String   PROFILE_KEY_PROTOCOL          = "protocol";
  static String   PROFILE_KEY_APPNAME           = "appname";
  static String   PROFILE_KEY_DEVICE_ID         = "deviceId";
  static String   PUSH_HTTP_DEFAULT_DEVICE_PORT = "100";
  static String   DEVICE_ID                     = "2100000a";

    public static void main(String[] args)
    {
      PushClientApp app = new PushClientApp();
      app.enterEventDispatcher();
    }

    Thread pushThread;

    PushClientApp()
    {
      // Set the connection profile information
      System.out.println("++++++++  Starting the client  +++
+++++++");
      ConnectionProfile syncprofile =
      TestDB.getSynchronizationProfile();
      syncprofile.setServerName("kpatilxp");
      syncprofile.setPortNumber(2480);
      syncprofile.save();

        // Login to the SUP server
        TestDB.loginToSync("supAdmin", "s3pAdmin");

        // Start the http push listener thread
        pushThread = new Thread(new PushListener());
        pushThread.start();

        setPushConnectionProfile("Test:1.0", DEVICE_ID,
        syncprofile, "PushClientApp");

        // Enable SIS on the synchronization group
```

```
        SynchronizationGroup sg =
        TestDB.getSynchronizationGroup("PushEnabled");
        sg.setEnableSIS(true);
        sg.setInterval(3);
        sg.save(); // this will update the local db

       // This will synchronize the SIS subscription to the
server
        TestDB.synchronize();
        System.out.println("++++ Synchronization succeeded +
+++++");
    }

 /*
 * For now this assumes MDS is running on localhost
 * Creates the URL for PUSH
 *
 * @param deviceid for SUP client
 */
    public static String getHTTPPushAddress(String deviceid)
    {
        String mdsServer = MDSSERVER;

        String mdsPort = MDSSERVERPORT;

        StringBuffer result = new StringBuffer("http://");
        result.append(mdsServer);
        result.append(":");
        result.append(mdsPort);
        result.append("/push?DESTINATION=");
        result.append(deviceid);
        result.append("&PORT=");
        result.append(PUSH_HTTP_DEFAULT_DEVICE_PORT);
        return result.toString();
    }

    /**
     * Sets up push settings for specified package's
     * synchronization profile.
     *
     * @param packageName
     *            the specified package name
     * @return true if set up succesfully.
     */
    private boolean setPushConnectionProfile(String
packageName,
    String deviceId, ConnectionProfile syncProfile,
            String appId)
    {

      try
      {
        String httpPushAddress =
getHTTPPushAddress(deviceId);

        syncProfile.setProperty(PROFILE_KEY_ADDRESS,
```

```
        httpPushAddress);

        syncProfile.setProperty(PROFILE_KEY_PROTOCOL,
        PROFILE_HTTP_PUSH_PROTOCOL);

          syncProfile.setProperty(PROFILE_KEY_APPNAME,
appId);

          syncProfile.setProperty(PROFILE_KEY_DEVICE_ID,
deviceId);

          syncProfile.save();
    }
    catch (Exception e)
    {
     System.out.println(">> setPushConnectionProfile -
     Exception e : " + e);
     return false;
     }

     return true;
}
```

Best Practices for Developing Applications

Observe best practices to help improve the success of software development for Sybase Unwired Platform.

Set up your development environment and develop your application using the procedures in the *Developer Guide for BlackBerry*.

Check Network Connection Before Login

Use `offlineLogin` to test the wireless connection before a login attempt is made. If the wireless connection option has been switched off, the `loginToSync` call takes a long time to fail when a wrong password is entered, with an 'Invalid Password' error message appearing only after a timeout.

```
import net.rim.device.api.system.RadioInfo;
……
 public class Test {
    public static void main(String[] args) {
    if (isWirelessConnected ())
   {
     XXDB.loginToSync(username, password);
   }
   else
   {
     boolean result = XXDB. offlineLogin(username, password);
     if (result == false)
       {
         throw new Exception("Offline login failed");
```

```
        }
    }
}

    public static boolean isWirelessConnected()
   {
     boolean isWirelessConnected = true;
     int state = RadioInfo.getState();
     int signal = RadioInfo.getSignalLevel();
     if (state == RadioInfo.STATE_OFF || state ==
RadioInfo.STATE_LOWBATT ||
       signal == RadioInfo.LEVEL_NO_COVERAGE)
       {
         isWirelessConnected = false;
        }
       return isWirelessConnected;
        }
}
```

If your application uses BES or BIS connectivity to connect to an Unwired Server from a BlackBerry device, the application can automatically switch to use Wi-Fi when it becomes available. If using a direct TCP connection, the application must explicitly specify a Wi-Fi transport type.

```
DatabaseClass.getSynchronizationProfile().setString("transport",
"WIFI")
```

Constructing Synchronization Parameters

When constructing synchronization parameters to filter rows to be download to a device, if the SQL statement involves two mobile business objects, you must use an "in" clause rather than a "join" clause. Otherwise, when there is a joined SQL statement, all rows of the subsequent mobile business object are filtered out.

For example, you would change this statement:

```
SELECT x.* FROM So_company x ,So_user y where x.company_id =
y.company_id and y.uname='test'
```

To:

```
SELECT x.* FROM So_company x where x.company_id in (select
y.company_id from So_user y where y.uname='test')
```

Clear Synchronization Parameters

When using synchronization parameters to retrieve data from an MBO during a synchronization session, you may want to clear the previous synchronization parameter values.

```
MBOSynchronizationParameters param =
<MBO>.SynchronizationParameters;
param.delete();
param = <MBO>.SynchronizationParameters; //must re-get the sync
parameter instance
```

```
params.Param1 = value1; //set new sync parameter value
params.Param2 = value2; //set new sync parameter value
param.save();
```

Clear the Local Database

Each time you redeploy a package on the Unwired Server, the client application should clear the local database. After clearing the database, login again so that the local database is reconstructed.

```
XXDB.deleteDatabase();
XXDB.loginToSync(); //Don't forget to login again so that the local
database will be re-constructed.
```

Process Synchonized Data

When performing data synchronization, apply logic to the data that is synchronized.

```
import com.sybase.persistence.SyncStatusListener;
……
public class Test {
    public static void main(String[] args) {
        XXDB.loginToSync();
       MySyncStatusListener myListener = new MySyncStatusListener();
        XXDB.synchronize(myListener);
        int receivedRowCount = myListener.getReceivedRowCount();
        if (receivedRowCount > 0)
        {
    // handle the logic only if there is data
synchronized.
        }
        …
        myListener.setReceivedRowCount(0);  // reset row count
        XXDB.synchronize(myListener);
        …….
    }
}
class MySyncStatusListener implements SyncStatusListener {
    private int                    _receivedRowCount = 0;
    @Override
    public boolean objectSyncStatus(ObjectSyncStatusData data) {
   if (data.getReceivedRowCount() > 0)
   {
     _receivedRowCount = data.getReceivedRowCount();
   }
        return false;
    }
    public int getReceivedRowCount() {
        return _receivedRowCount;
    }
    public void setReceivedRowCount(int receivedRowCount) {
        this._receivedRowCount = receivedRowCount;
    }
}
```

Create a Custom Callback Handler

Create a custom callback handler if the application requires a callback (for example, to allow the client framework to provide notification of synchronization results).

```
import com.sybase.persistence.DefaultCallbackHandler;
……
public class Test {
  public static void main(String[] args) {
    XXDB.loginToSync();
    XXDB.registerCallbackHandler(new MyCallbackHandler());
    GenericList sgs = new GenericList( 2 );
    sgs.add(SortDB.getSynchronizationGroup("sg1"));
    sgs.add(SortDB.getSynchronizationGroup("sg2"));
    XXDB.beginSynchronize(sgs, new SynchronizationNotification());
  }
}
class MyCallbackHandler extends DefaultCallbackHandler
{
  public int onSynchronize(GenericList groups,
SynchronizationContext context)
  {
    if ( context == null )
      {
        return SynchronizationAction.CANCEL;
      }
    if (!(context.getUserContext() instanceof
SynchronizationNotification))
      {
        return super.onSynchronize(groups, context);
      }
    switch (context.getStatus())
      {
        case SynchronizationStatus.STARTING:
          return beforeSynchronize(groups);
        case SynchronizationStatus.FINISHING:
          return afterSynchronize(groups);
        default:
          return SynchronizationAction.CONTINUE;
      }
  }
  private int beforeSynchronize(GenericList groups)
  {
    // ..... logic before sync
    if ( groups == null || groups.size() == 0 )
    {
      return SynchronizationAction.CANCEL;
    }

    return SynchronizationAction.CONTINUE;
  }
  private int afterSynchronize(GenericList groups)
  {
    // ..... logic after sync
```

```
    if ( groups == null || groups.size() == 0 )
    {
      return SynchronizationAction.CANCEL;
    }
    return SynchronizationAction.CONTINUE;
  }
}
```

Turn Off API Logger

In production environments, turn off the API logger to improve performance.

```
XXDB.getLogger().setLogLevel(LogLevel.OFF);
```

Index

L

M

N

O

P

Q

R

S

U