SYBASE®

Developer Reference for BlackBerry

Sybase Unwired Platform 1.5.2

Contents

Contents

Introduction to Developer Reference for BlackBerry

This developer reference 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*
- *Sybase Unwired WorkSpace – Device Application Development*
- *Tutorial: BlackBerry Device Application Development (Device Application Designer)*, where you create the SUP101 sample project referenced in this guide
- *Tutorial: BlackBerry Device Application Development (Custom Development)*
  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>\Servers\UnwiredServer\ClientAPI\apidoc`. There are subdirectories for `\j2me` and `\j2se`.

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

Documentation Road Map for Unwired Platform

Learn more about Sybase® Unwired Platform documentation.

Table 1. Unwired Platform documentation

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

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

Document	Description
Getting started tutorials	Tutorials for trying out basic development functionality.
	Audience: new developers, or any interested user.
	Use: after installation.

	• Learn mobile business object (MBO) basics, and create a mobile device application:
	   • *Tutorial: Mobile Business Object Development*
	   • *Tutorial: BlackBerry Application Development using Device Application Designer*
	   • *Tutorial: Windows Mobile Device Application Development using Device Application Designer*
	• Create native mobile device applications:
	   • *Tutorial: BlackBerry Application Development using Custom Development*
	   • *Tutorial: iPhone Application Development using Custom Development*
	   • *Tutorial: Windows Mobile Application Development using Custom Development*
	• Create a mobile workflow package:
	   • *Tutorial: Mobile Workflow Package Development*

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

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

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 using either:
  - The Device Application Designer – an Unwired Platform graphical design and development tool.
  - A native IDE – an API approach that uses your native IDE's Client Object API and Device Framework 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.
  - A combination of both development environments – begin device development with the Device Application Designer, and further customize with an IDE.

This guide provides reference material for the last two options.

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. See *Sybase Unwired WorkSpace - Device Application Development* for information about device application features and appearance.

Development Task Flows

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

This diagram illustrates the two approaches by which you can develop a device application: directly from mobile business objects (MBOs), using the custom UI approach, or by extending Device Application Designer code by using the Designer Generated UI approach.
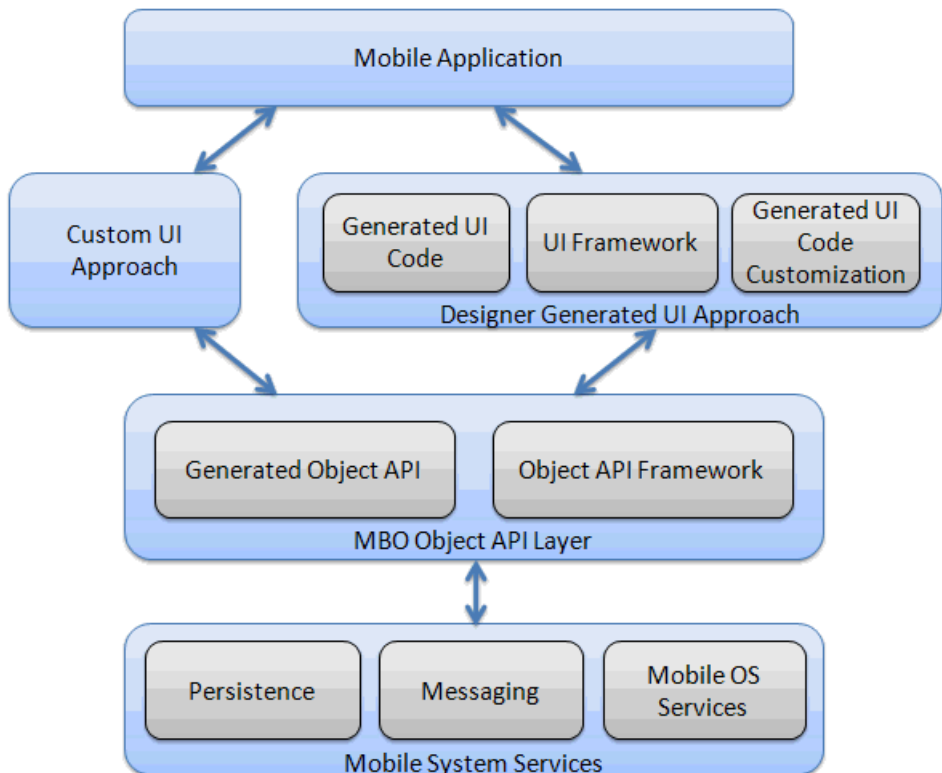
Task Flow for BlackBerry JDE Development

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:

Task Flow for Device Application Designer and BlackBerry JDE Development

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

Highlevel steps:

Configuring Your BlackBerry Development Environment

This section describes how to set up your BlackBerry development environment, and provides the location of required JAR files, Javadocs, 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, but since Unwired WorkSpace and the Device Application Designer both run in Eclipse, Sybase recommends that you use the BlackBerry Java plug-in for Eclipse for a more integrated development environment.

For information on transitioning from the BlackBerry JDE to the eJDE, view the video *http://www.blackberry.com/DevMediaLibrary/view.do?name=JDEtoeclipse* at the Research In Motion Developer Video Library Web site.

Installing the BlackBerry Java Plug-in for Eclipse

The Device Application Designer supports the BlackBerry Java Plug-in for Eclipse, which allows you to generate the device application code using the Device Application Designer code generation wizard, then debug the generated code.

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. Start Sybase Unwired WorkSpace.
2. Select **Help > Install New Software** from the main menu.
3. In the Available Software dialog box, click **Add**.
4. In Name, enter `BlackBerry Update Site`.
5. In the Location field, enter *http://www.blackberry.com/go/eclipseUpdate/3.5/java*.
6. Click **OK**.
7. From the "Work with" list, select the **BlackBerry Update Site**.
8. Select **BlackBerry Java Plug-in Category**, and click **Next**.
9. On the Review Licenses page, indicate whether or not you accept the terms of the license.
10. Click **Finish**.
11. (Optional) If you are prompted, enter the user name and password for your BlackBerry developer account.
12. Click **Yes** to restart Sybase Unwired WorkSpace.

BlackBerry Java Plug-in for Eclipse Integration

The Device Application Designer Code Generation wizard is integrated with the BlackBerry Java Plug-in for Eclipse.

You can launch a BlackBerry project directly from Eclipse after code generation. This allows you to debug generated device application code that contains the user interface framework within a BlackBerry project.

Note: To use this feature, you must first install the BlackBerry Java Plug-in for Eclipse v1.1.

See the documentation on the BlackBerry developer Web site *http://docs.blackberry.com/en/developers/?userType=21* for more information about the BlackBerry Java Plug-in for Eclipse.

Downloading the BlackBerry JDE and MDS Simulator

To generate and distribute BlackBerry device applications built with the Unwired WorkSpace Device Application Designer, 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.

Note: The BlackBerry JDE is a standalone development environment. The BlackBerry Java Plug-in for Eclipse v1.1 is recommended.

Task

1. Go to the BlackBerry Web site at *http://na.blackberry.com/eng/developers/javaappdev/javadevenv.jsp* to download and install the BlackBerry JDE.

   Note: BlackBerry JDE 4.7 supports the touch screen features of the BlackBerry Storm device.

2. Go to *http://na.blackberry.com/eng/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 (UltraLiteJ™) libraries – `<UnwiredPlatform_InstallDir>\Servers\UnwiredServer\ClientAPI\UltraliteJ`.
- Framework classes that are used by generated classes (J2ME, J2SE and RIM BlackBerry) – `<UnwiredPlatform_InstallDir>\Servers\UnwiredServer\ClientAPI\java`

Mobile Business Object Code or Device Application Designer Code

Determine whether to develop a device application directly from mobile business object (MBO) generated code, or from Device Application Designer generated code, then generate the code according to your decision.

Note: Do not modify generated MBO API or Device Application Designer generated code directly. For Device Application Designer Code, use the customization pattern documented in

this guide by either adding event handlers or customization classes. For MBO generated code, create a layer on top of the MBOs using patterns native to the mobile operating system development to extend and add functionality.

To avoid errors or inconsistent behavior, client applications must be regenerated whenever a mobile application package has been redeployed.

Generating BlackBerry Mobile Application Project Code

After developing the mobile business objects (MBOs), generate the Java files that implement the business logic and are required for BlackBerry device application development.

1. From Unwired WorkSpace, right-click in the Mobile Application Diagram of the project for which you are generating code and select **Generate Code**.

2. Follow the Code Generation wizard instructions to generate code appropriate for the BlackBerry JDE environment, by selecting **Java** as the language and, in this case, **Java ME for BlackBerry** as the platform.
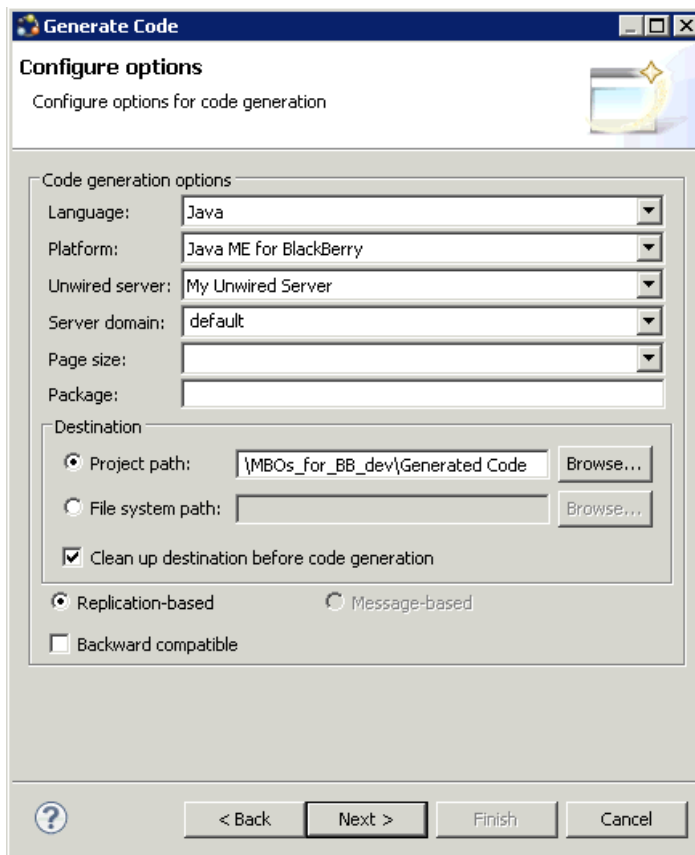
Other selections affect generated output as well. For example, if you include an Unwired Server entry, it generates a default connection to Unwired Server.

3. Click **Next**. Select the MBOs for which you are generating code and any additional options you require.
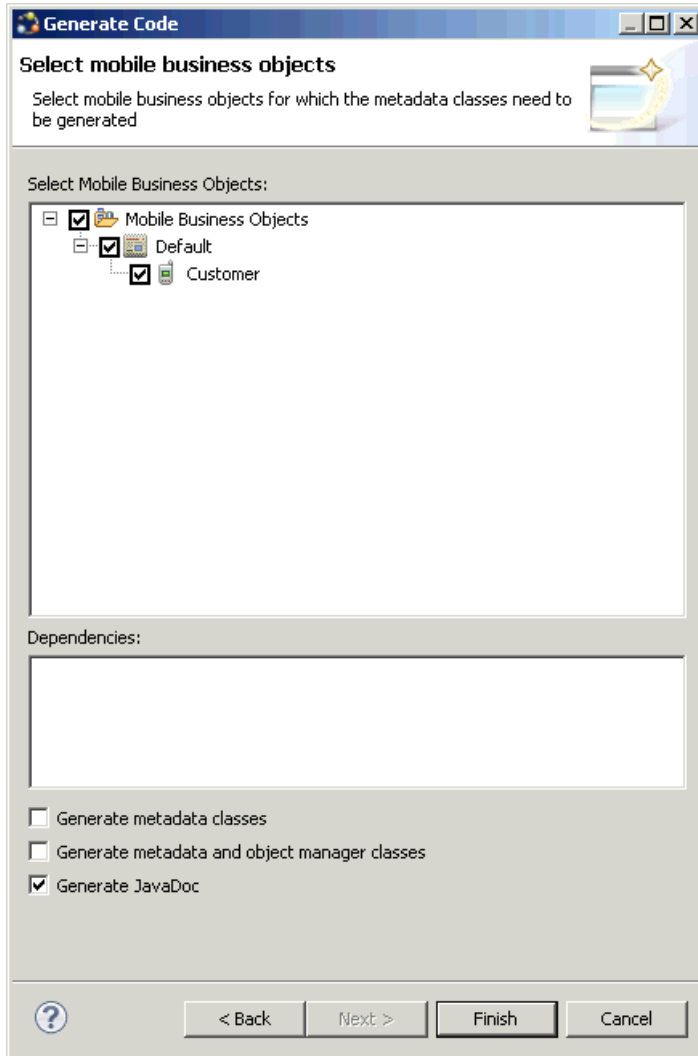
See *Generating Object API Code* for details of all options.

4. Click **Finish**.

Ignore errors that reference the sup-rim-afaria.jar file since it has been removed and not required. The class files include all methods required to create connections,

synchronize deployed MBOs with the device, query objects, and so on, as defined in your MBOs.

By default, the MBO source code and supporting documentation are generated in the project's Generated Code folder. The generated Java files are located in the <MBO_project_name> folder under the `src` folder:
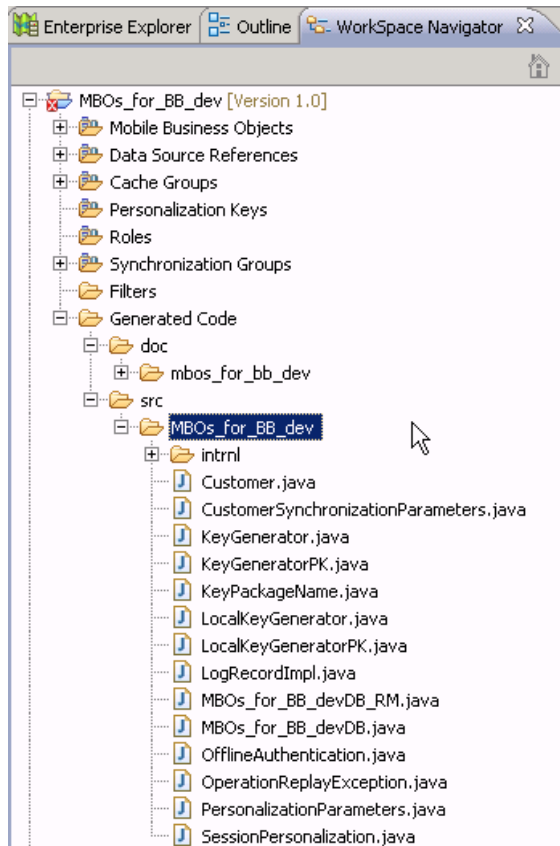
The frequently used Java files in this project, described in code samples include:

Table 2. Source code file descriptions

Java file	Description
MBO class (for example, Customer. java)	Includes all the attributes, operations, object queries, and so on, defined in this MBO.
Synchronization parameter class (for example, CustomerSynchronizationParameters.java)	Includes any synchronization parameters defined in this MBO.

Java file	Description
Key generator classes (for example, KeyGenerator.java)	Includes generation of surrogate keys used to identify and track MBO instances and data.
Local Key generator classes(for example LocalKeyGenerator.java)	Includes generation of surrogate keys used to identify and track MBO instances and data that exist only on the local device.
Personalization parameter classes (for example, PersonalizationParameters.java)	Includes any defined personalization keys.
OfflineAuthentication.java	Saves authentication information locally and includes methods used between the device application and local database for offline authentication (does not communicate with Unwired Server).
<PkgName>DB (where *PkgName* is the name of the project/package, for example, MBOs_for_BB_devDB)	Defines API to handle client database access, synchronization profile, authentication, and synchronization operations.
ObjectManager (for example, MBOs_for_BB_devDB_RM)	Invokes methods and retrieves the metadata of packages, MBOs, attributes, operations, and parameters during runtime using the name (reflection) instead of the object instance.
	Note: ObjectManager classes are generated only when you select the **Generate metadata and object manager classes** option.
Other operation classes (for example, <MBO><OtherOperation>Operation)	Encapsulates each other operation into an object

Generating BlackBerry Device Application Code from the Device Application Designer

After developing the mobile business objects (MBOs), begin device application development using the Device Application Designer, then use the Generate Device Application wizard to generate the device application code required for further development in the BlackBerry JDE.

Use this procedure if you are developing BlackBerry device applications using both the Device Application Designer and the BlackBerry JDE.

1. From Unwired WorkSpace, select **File > New > Device Application Designer**.
2. Follow the Device Application Designer wizard instructions to create a Device Application Designer project based on the MBOs that are appropriate for the type of BlackBerry device application you are developing, and click **Finish**.
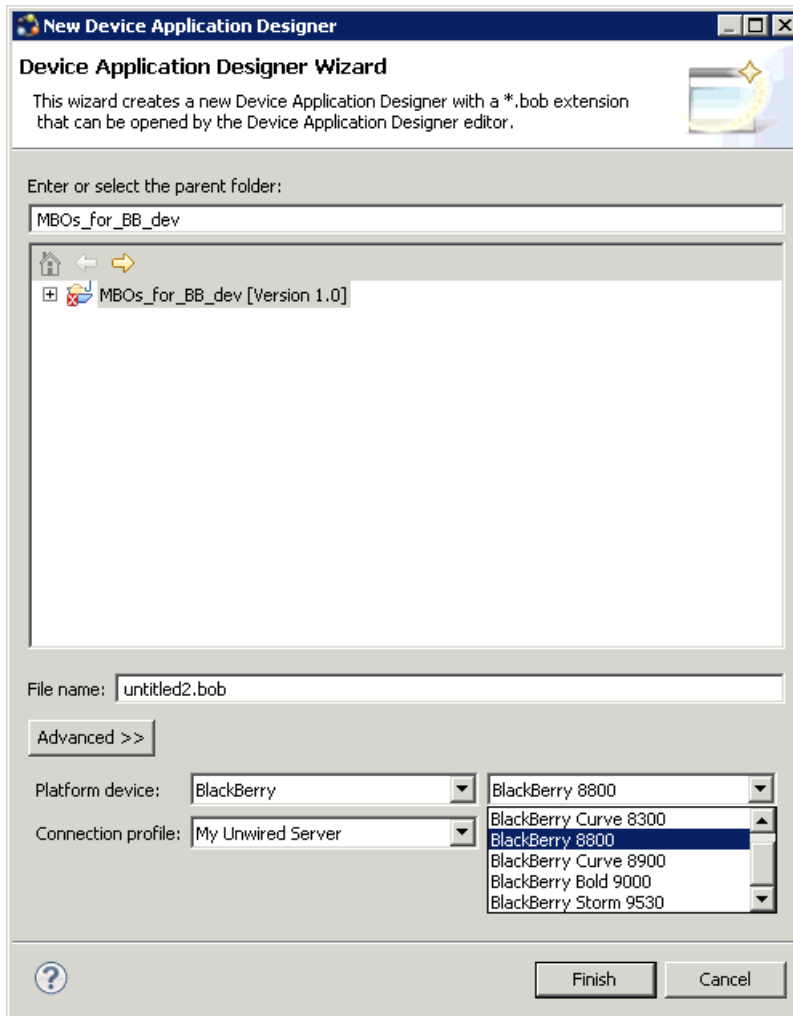
3. Develop as much of the device application as you want using the Device Application Designer.
4. Generate the code for a BlackBerry Device application, then debug and extend the code in the BlackBerry JDE.

Generating Code For a BlackBerry Device Application
Use the Generate Device Application wizard to generate the device application code.

Prerequisites

Verify the device application and fix any errors that are found. Device applications with errors cannot be generated.

Task

1. Click the Verify icon on the toolbar to verify the device application has no errors.
2. Click the code generation icon on the toolbar.
3. In the Generate Device Application wizard, in Device Platform, select **BlackBerry**, and, optionally, select:

Option	Description
Server domain	Select the domain to use for the connection profile. The profiles used in the design appear in the Profile column. The initial value of "default" appears under the domain. Select the Domain column to choose a different domain.
	You can enter any value you want or select one that is available. The list of available domains are returned from the profile. If you have previously connected, it caches the list of last known domains. If you have never connected, "default" is returned.
Locale	Expand this section to see a list of available locales from which you can select.
Advanced	Expand this section for advanced options: • Check mobile business object on Sybase Unwired Platform Server – select to verify that the mobile business objects that are used in the device application exist on the corresponding Unwired Server. • Mobile Business Object Group – the mobile business object group that contains the mobile business objects you want to verify. Click **Generate Code** to launch the Generate Code wizard.

4. Click **Next**.
5. In the Generate Device Application wizard, enter the information, then click **Finish**:

Option	Description
Favorite Configurations	(Optional) Select a saved configuration from the drop-down list.
	Note: The **Remove the custom folder** option state is not saved in a favorite configuration. You must explicitly choose that option when you want to remove the custom folder that contains any custom coding you have added.
Locations	• Generate code only – if this option is selected, the code is not compiled. • Deploy the BlackBerry application – select to deploy the BlackBerry application to the specified location. Selecting this option activates the Deploy Configurations section, where you can set the locations for the BlackBerry rapc compiler, simulator location, and so on.

Option	Description
Deploy Configurations	• BlackBerry rapc compiler – the BlackBerry JDE rapc compiler location. This determines the operating system version of the generated BlackBerry application. • Copy to Simulator location – the filepath to the location of the compatible simulator you want to use for development and testing. BlackBerry JDE 4.2.1 is the default. **Note:** BlackBerry JDE 4.7 supports the touch screen features of BlackBerry Storm. • Start the BlackBerry Simulator after copying – select to start the simulator after you generate the device client application. • Start MDS automatically – select if a connection to the mobile business object is required. This is a one-time action. A check is performed to see whether the MDS is started. If it is, it will not be started twice. Once the MDS is running, it does not have to be restarted. • Disable signing – disables the signing process for testing only. This option is available only when you are copying the generated COD files to a simulator location for testing. • Copy to desktop location – the location where you want to copy the generated .cod files. • Copy to Workspace location – the location for the workspace to which you want to copy the device client application.

Option	Description
Debug the BlackBerry application	Debug configurations: • Client project • Options project • Launch the BlackBerry project • Launch configuration **Note:** This option is enabled only if you have installed the BlackBerry Java Plug-in for Eclipse.
Advanced	• Open the generated folder under Windows Explorer – select this option to open the generated folder under Windows Explorer when code generation is complete. This is useful to locate the generated source so you can move it to a JDE project or for custom coding. • Generated artifacts location – this option indicates the location of the folder where all the code is generated. The default is \Sybase\UnwiredPlatform-1_5\workspace \.metadata\.plugins\com.syb-ase.uep.bob.rim\appname.bob. Click **Browse** to change the location. Click **Restore** to restore to the default location. • Remove the Custom folder – select this option to remove any previously generated custom code placed in the Custom folder. • Use JDK path – use this JDK path for the **rapc.exe** compiler from RIM. This must be in the following format: `c:\program files\java \jdk1.6.0_16\bin`.

Device Application Designer Generated Code Structure

This topic illustrates the structure of code generated by the Device Application Designer, and describes the contents of folders.

The BlackBerry application is built into two parts: the client application and the options library. The custom package contains all client code, while the options package contains

---

options code. The custom package accommodates all screen classes as well as any tab folder packages. The tab panel classes are contained in the tab folder packages.

Client code is generated into two categories:

1. Code is generated each time you invoke the Device Application Designer generation wizard. The Device Application Designer Model document is parsed and the screen classes and the BOBCUIDefinition/BOBCOptionsDefinition class are generated.
2. User's custom code is initially generated by the Device Application Designer when the Device Application Designer generation wizard is invoked. The custom code extends the generated screen classes and BOBCUIDefinition/ BOBCOptionsDefinition class.

Modify the custom code only to customize your BlackBerry device application in the BlackBerry JDE (or any other Native IDE).

```
□ 📁 untitled1.bob
   ⊞ 📁 com
   □ 📁 custom
      □ 📁 com
         □ 📁 sybase
            □ 📁 uep
               □ 📁 bobclient
                  □ 📁 custom
                        📁 BOBScreenUntitled6CustomTab
                     📁 options
   □ 📁 generated
      □ 📁 com
         □ 📁 sybase
            □ 📁 uep
               □ 📁 bobclient
                  □ 📁 custom
                        📁 BOBScreenUntitled6Tab
                     📁 options
         📁 images
      📁 images
      📁 lib
   □ 📁 object_api
      □ 📁 genfiles
         □ 📁 java
            □ 📁 src
               □ 📁 test
                     📁 intrnl
      📁 ota
      📁 temp_afx
```
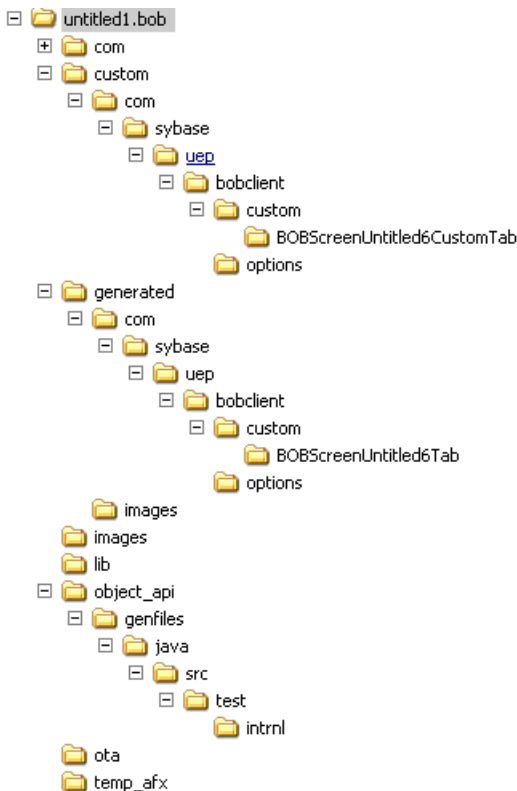
Custom Coding Subclasses

You can enable custom code generation by assigning a value of true to the platform-specific property **Generate a custom coding subclass** for the Device, Screens, Tab Folder, and Tab panels. You can specify this property for all elements from the preference page.

Folder Contents

The application client code and images are in the `generated` folder.

All object API code is generated into a separate folder named `object_api`.

The custom code is generated into the `custom` folder in the same package structure as the generated folder. For each element that enables custom coding, a subclass is generated in this folder. Once generated, the custom subclass is not overwritten in subsequent code generation unless you select **Remove custom folder** in the generation wizard.

You can customize subclasses to insert your own code in your development environment.

Event Delegates

For elements that support widget events, you can specify what events are supported for a particular element. When any of these events are selected, an event delegate is generated in the `custom` folder to delegate all the events for this element. The event delegate class is in same package as its element's containing class (that is, the event delegate class for a button is located in the same package as the subclass for the parent screen).

You can customize the event delegate to apply your own code.

Similar to the custom subclasses, the event delegate is not overwritten during subsequent code generation, unless you select **Remove custom folder**.

Creating Projects and Importing Files into the BlackBerry Development Environment

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

Differences Between Mobile Business Object and Device Application Designer Required Files

The procedures for developing a device application directly from mobile business object (MBO) generated code differ slightly compared to developing from Device Application Designer generated code.

The main differences between the two procedures are:

- Device Application Designer – contains MBO business logic and BlackBerry device application code. You must:
  - Include libraries and JAR files in the BlackBerry project that support both the BlackBerry Client Object API and the BlackBerry Device Framework API.

- Add the Java files from the Device Application Designer Custom folder, generated folder, and the generated MBO classes to the BlackBerry project.
- Mobile business objects – contain only MBO business logic. If you do not plan on using the Device Application Designer, 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, due to a RIM limitation. The libraries must be outside the projects and referred to with an absolute path.
- The debug option in the BlackBerry generation wizard page is enabled if the BlackBerry Java Plug-in for Eclipse is installed. This option can be useful when developing device applications.

Creating a BlackBerry Device Application Project

Create the BlackBerry project and add the generated mobile business object (MBO) Java files, or the Device Application Designer Java and framework files, to the BlackBerry JDE.

Follow these steps whether you are developing the device application directly from code generated from MBOs, or extending an existing Device Application Designer device application, except where noted.

Note: These steps apply only if you are using the BlackBerry JDE to develop the application. If you are using the BlackBerry Eclipse plug-in, configuration is performed automatically by the Device Application Designer when you generate the code.

1. Launch the BlackBerry JDE and create a new workspace.
2. Create a BlackBerry project and name it supOptions.
3. Right-click the project and select **Properties**.
4. In the properties dialog, select the **Application** tab, specify Library for **Project type** and select **Auto-run on startup**.
5. Select the **Build** tab, and click **Add** next to "Imported jar files." Add either:
   - For Device Application Designer – these UltraLiteJ.jar and BOBFramework.jar files to the project:
     - <UnwiredPlatform_InstallDir>\Unwired_WorkSpace \Eclipse\sybase_workspace\mobile\eclipse\plugins \com.sybase.uep.bob.rim<version_time_stamp>\generate \blackberry\UltraLiteJ.jar

- `<UnwiredPlatform_InstallDir>\Unwired_WorkSpace`
  `\Eclipse\sybase_workspace\mobile\eclipse\plugins`
  `\com.sybase.uep.bob.rim<version_time_stamp>\generate`
  `\blackberry\build-4.<version>\BOBFramework.jar`

  > **Note:** (Device Application Designer only) Select the version of
  > `BOBFramework.jar` that corresponds with selected Device Application
  > Designer code generation options (BlackBerry rapc compiler version) from these
  > supported versions: 4.2.1, 4.3.0, 4.5.0, 4.6.1, or 4.7.0.

- For MBO generated code – these `UltraLiteJ.jar` and `sup-client-rim.jar` files to the project:
  - `<UnwiredPlatform_InstallDir>\Servers\UnwiredServer`
    `\ClientAPI\UltraliteJ\J2meRim11\UltraLiteJ.jar`
  - `<UnwiredPlatform_InstallDir>\Servers\UnwiredServer`
    `\ClientAPI\java\RIM42\sup-client-rim.jar`

6. Click **OK**.
7. Right-click the `supOptions` project and select **Add file to project** to add files for the project, which depends on whether you are creating a device application directly from MBO or from Device Application Designer code:
   - MBO generated code – references the Client object API and contains the Java files that implements the business logic of your project. From **Look in**, 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 add all of the `.java` files to your project.
   - Device Application Generated code – references the Device Application Generated code, `BOBCOptionsDefinition.java` (the options definition), and `OptionsMain.java`. Complete these additional steps:
     1. Verify that the generated code is the correct version (you can see the version from the Device Application Designer generation wizard).
     2. From **Look in**, navigate to the object API code at the generated location and add it to the project: `<workspace>\.metadata\.plugins`
        `\com.sybase.uep.bob.rim\<bobfilename>\object_api`
        `\genfiles\java\src`
     3. Locate the options definition file at the following location and add it to the project:
        `<workspace>\.metadata\.plugins\com.sybase.uep.bob.rim`
        `\<bobfilename>\generated\com\sybase\uep\bobclient`
        `\options\BOBCOptionsDefinition.java`
     4. Locate `OptionsMain.java` at the following location and add it to the project:
        `<UnwiredPlatform_InstallDir>\Unwired_WorkSpace`
        `\Eclipse\sybase_workspace\mobile\eclipse\plugins`

```
\com.sybase.uep.bob.rim<version_time_stamp>\generate
\blackberry\src\com\sybase\uep\bobclient\options
\OptionsMain.java
```

Creating a BlackBerry Device Application Client Project

Create a BlackBerry client project that contains Device Application Designer generated code (other than the options definition code), and BOBUIController.java (the application entry point).

1. Create a BlackBerry project in the same workspace that contains the supOptions project and name it supClient.
2. Right-click the project and select **Properties**.
3. In the properties dialog, select the **Application** tab, and specify CLDC Application as the project type.
4. Select the **Build** tab, and in the Imported jar files section, click **Add** to add BOBFramework.jar to the project, which is located in:

   ```
   <UnwiredPlatform_InstallDir>\Unwired_WorkSpace\Eclipse
   \sybase_workspace\mobile\eclipse\plugins
   \com.sybase.uep.bob.rim_<version_timestamp>\generate\
   \blackberry\build-<version>\BOBFramework.jar
   ```

   Note: Select the correct version of BOBFramework.jar for your BlackBerry operating system from these supported versions: 4.2.1, 4.3.0, 4.5.0, 4.6.1, or 4.7.0.

5. Select the supClient project, right-click and select **Add file to project** to insert generated Device Application Designer code into the project:

   a) Verify that the generated code is the correct version (you can see the version from the Device Application Designer generation wizard).

   b) Locate the Device Application Designer generated code at the generated location and add it to the project. For example:

   ```
   <workspace_name>\.metadata\.plugins
   \com.sybase.uep.bob.rim\<bobfilename>\generated\com
   \sybase\uep\bobclient\custom\
   ```

   c) Locate BOBUIController.java in the installation location and add it to the project:

   ```
   <UnwiredPlatform_InstallDir>\Unwired_WorkSpace\Eclipse
   \sybase_workspace\mobile\eclipse\plugins
   \com.sybase.uep.bob.rim_<version_timestamp>\generate
   \blackberry\src\com\sybase\uep\bobclient\controller
   \BOBUIController.java
   ```

   Note: All Device Application Designer generated code and required libraries are imported to the projects, and project types are set. If custom coding or widget events are enabled, you must also add the generated subclasses and widget event delegate to

the client project. This code is located in the custom folder at `<workspace>`
`\.metadata\.plugins\com.sybase.uep.bob.rim\<bobfilename>`
`\custom.`

6. To make the client project dependent on the options project, select the `supClient` project and select **Project Dependencies**. Select the `supOptions` project in the dialog box.

Referencing BlackBerry Device Framework Javadocs

The BlackBerry Device Framework API enables you to customize the BlackBerry device user interface. Javadoc provides a complete reference to the API.

Perform these steps only if developing the application in the BlackBerry Eclipse plug-in (eJDE). The BlackBerry JDE does not support this Javadoc configuration.

1. Click *BOBFrameworkJavadoc.zip* and select **Save**. Save the file to the `<UnwiredPlatform_InstallDir>\UnwiredPlatform` `\Unwired_WorkSpace\Eclipse\sybase_workspace\mobile\eclipse` `\plugins\com.sybase.uep.bob.rim_1.5.2.<version>\generate` `\blackberry` directory.

2. Verify that the directory now contains the `BOBFrameworkJavadoc.zip` file.

3. From the eJDE, right-click your project and choose **Properties**.

4. Select **Java Build Path**, then **Libraries**.

5. Expand `BOBFramework.jar` and select the Javadoc location.

6. Click **Edit**, select **Javadoc in archive**, and set the archive path to `<UnwiredPlatform_InstallDir>\UnwiredPlatform` `\Unwired_WorkSpace\Eclipse\sybase_workspace\mobile\eclipse` `\plugins\com.sybase.uep.bob.rim_1.5.2.<version>\generate` `\blackberry\BOBFrameworkJavadoc.zip`.

Adding Required .jar and .cod Files

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

Add these files only if you are developing the device application in the BlackBerry JDE, which are located in the Unwired WorkSpace installation path as indicated, to your project's build path:

- `sup-client-rim.jar` – from `<UnwiredPlatform_InstallDir>` `\Servers\UnwiredServer\clientAPI\java\RIM42` for the Blackberry client.
- `UltraLiteJ.jar` – from `<UnwiredPlatform_InstallDir>\Servers` `\SQLAnywhere11\UltraLite\UltraliteJ\J2se` for java.

- `UltraLiteJ.jar` from `<UnwiredPlatform_InstallDir>\Servers` `\SQLAnywhere11\UltraLite\UltraliteJ\BlackBerry4.2` for the Device Application Designer client.

Copy required .cod files to the BlackBerry simulator directory: `UltraLiteJ.cod` from `<UnwiredPlatform_InstallDir>\Servers\SQLAnywhere11\UltraLite` `\UltraliteJ\BlackBerry4.2` for the BlackBerry client.

Developing, Debugging, and Customizing BlackBerry Applications

Use the BlackBerry Client Object API, BlackBerry Device Application Framework 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 API or Device Application Designer generated code directly. For Device Application Designer Code, use the customization pattern documented in this guide by either adding event handlers or customization classes. For MBO generated code, create a layer on top of the MBOs using patterns native to the mobile operating system development to extend and add functionality.

Building an Object API based Client Application

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. Log in to Unwired Server using a user name and password:
   ```
   <PkgName>DB.loginToSync("supAdmin", "s3pAdmin");
   ```
2. Synchronize MBOs by group name synchronization:
   ```
   <PkgName>DB.synchronize("default");
   ```
3. Retrieve MBO data:
   ```
   ObjectList customers = Customer.findAll();
   int size = customers.count();
   for (int i = 0; i < size; i++)
   {
       Customer cust = (Customer)customers.elementAt(i);
       //Feed the MBO data to your view…
   }
   ```
4. Update MBO data:
   ```
   Customer cust = Customer.findByPrimaryKey(100);
   cust.setAddress("1 Sybase Dr.");
   ```

```
cust.setPhone("9252360000");
cust.save();//or cust.update();
```

5. Submit pending operations and synchronize again:

```
Customer.submitPendingOperations();
<PkgName>DB.synchronize("default");
```

Adding a Device Application Entry Point

If you are creating a BlackBerry device application from code generated directly from mobile business objects (MBOs), add a main file to the application.

1. From the BlackBerry project that contains your generated MBO code, for example supOptions, 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.

Developing the BlackBerry Device Application

This section provides procedures and compares the differences between creating a BlackBerry Device 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 Device Application Development (Custom Development)*, which creates the business logic and generates the Java files required for the device 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.

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

Follow these procedures to run the SUP101 project in the BlackBerry Eclipse plug-in (eJDE).

1. Modify the build path to point to the correct location for the sup-client-rim.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` 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 as BlackBerry simulator.

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 `uep_ribbon_icon.png` to Icon files.

5. Add `sup-client-rim.jar` and `UltraLiteJ.jar` files to the Build import jar files.

6. Copy `sup-client-rim.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 Reference 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, except for Other operations, which cannot be retrieved with getLogRecords.

  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.
- • View the log records on the **LogInfo** screen, which shows both client and server log
  information for the application. You can change the client log level in **BlackBerry options
  > Logging**.
- • 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.
  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 *http://www.blackberry.com/DevMediaLibrary/view.do? name=debug*.

Customizing Device Application Designer Code

After you generate code from your BlackBerry development environment, choosing the **Custom Class** generation option, go to the folders that contain the custom classes, and modify the code as needed. For example, adding controls to screens, adding widget event code, and so on.

Generated Screen Class Outline:

```
public class BOBScreenSales_order extends
BaseBOBScreen implements IBOBScreen {

    protected void defineScreen() {
        createStyles();
        createControls();
        createMenus();
    }

    protected void createStyles() {
        createStyleById(XXX_STYLE);
    }

    protected void createControls() {
        createControlById(CONTROL1);
        configureControlById(CONTROL1);
    }

    protected void configureControlById(int ID) {
        switch (ID) {
        case LAYOUTMANAGER:
            configureObjectMetaDataById(ID, layoutManager);
            configureObjectHandlersById(ID, layoutManager);
            return;
        }
    }
```

```
    protected void createMenus() {
        menu1 = createMenuById(MENU1);
    }

    public Object getControlById(int ID);
}
```

Manually Adding Controls to a Screen

The `createControls` method allows you to manually add controls to a screen.

The following code adds controls to a layoutManager, which is eventually added to the screen. The layoutManager takes the control, column span, and row span as arguments to add the control and lay it out.

`ConfigureControlById` configures the controls's meta data(calling `configureObjectMetaDataById`) and handlers(calling `configureObjectHandlersById`) by Ids. A custom subclass can override this method to customize controls.

```
layoutManager = (LayoutManager) createControlById(LAYOUTMANAGER);
label1 = (Label) createControlById(LABEL1);
....
configureControlById(LAYOUTMANAGER);
configureControlById(LABEL1);
...
layoutManager.addWidget(label1, 1, 1);
...
this.add(layoutManager);
```

Writing Widget Event Code

To enable widget events, you must first specify events for a control from its coding properties tab.

During code generation, the event delegate gets generated into the custom folder. The event delegate is called by `control.setCustomEventsDelegate(eventDelegate, eventTyle)` in the `configureObjectHandlersById` method.

At runtime, the control's specified events are passed to the event delegate and handled by your code. The event delegate implements `IcustomEventsDelegate` and has the following methods to handle a variety of events:

- Methods to `onLoad` event:
  - `onLoad`: called during loading of the control
- Methods to `onDraw` event:
  - `paint`: called when painting `drawFocus`: called when drawing focus
- Methods to `onClick` event:
  - `onFocus`: called when focused
  - `onUnfocus`: called when unfocused
  - `navigationClick`: called when navigation clicked

- navigationUnclick: called when navigation unclicked
- navigationMovement: called when navigation moved
- touchEvent: called when touched

Note: this event is available only to touch screen devices, make sure you define the correct preprocessor for the client project to make it work properly. Available preprocessors include VER_4_2_1, VER_4_3_0, VER_4_5_0, VER_4_6_1 and VER_4_7_0.

- Methods to onRecordSelectionChange event:
  - onRecordChange: called when scrolling through records and mainly used for table type controls (cell table, grid table, list detail). It can be used for enabling or disabling menu items, activating phone actions, expanding content of a cell, or showing cell content in a fish eye view.
- Methods to onValueChange event:
  - onValueChange: called when the value of an input control changes, and useful for linked parameters in that the values of a control change based on the selected value of another control. This is primarily used in CGI type controls.
- Methods to onOrientationChange event:
  - onOrientationChange: called when orientation changes

Note: this event is available only to touch screen devices, make sure you define the correct preprocessor for the client project to make it work properly. Available preprocessors include VER_4_2_1, VER_4_3_0, VER_4_5_0, VER_4_6_1 and VER_4_7_0.

The following sample code shows how to write widget events:

```
public void onOrientationChange(Object field, int controlID,
                                int width, int height) {
  // custom code
  Switch(controlID)
  {
  case BOBScreenCustomers.BUTTON:
    Dialog.alert("button orientation changed!");
  default:
    return;
  }
};
```

Adding Validators

You can assign TextInput a vector of validators. Validator validates the text input's value based on the validation types and patterns.

The following is a code example for adding validators:

```
case TEXTINPUT1:
  TextInput localtextInput1 = (TextInput) object;
  Vector textField1ValidateVector = new Vector();
  textField1ValidateVector.addElement(new
  Validator("PATTERN_MATCH", "CONTAINS", "my",
          "The input does not contain {0}."));
  textField1ValidateVector.addElement(
```

```
    new Validator("PATTERN_MATCH", "STARTS_WITH", "sup",
                   "The input must start with {0}."));
  localtextInput1.setInvalidValueMessage(
     "Enter a valid value (data type {0} and logical type {1}).");
  localtextInput1.setValidators(textField1ValidateVector);
  localtextInput1.setFontStyle(styleEdit_box_Style);
break;
```

Perform UI Binding to an MBO

You can perform UI binding to an MBO through the `configureObjectMetaDataById`
method.

The following code shows how to perform UI binding to an MBO:

```
protected void configureObjectMetaDataById(int ID, Object object) {
  switch (ID) {
  ...
  case CELLTABLE1:
    CellTable localcellTable1 = (CellTable) object;
    localcellTable1.setColumnPercentage(
        new int[] {33, 34, 33 });
    localcellTable1.setSortingColumn("Sort on column");
    localcellTable1.setMboId(BOBCUIDefinition.MBO_A_B_C_DEPARTMENT);
    localcellTable1.setFocusFontStyle(styleCell_Table_Focus_Style);
    localcellTable1.setNumberOfColumns(3);
    localcellTable1.setColumnConfig(
        new String[] {"dept_id","dept_name","dept_head_id" });
    localcellTable1.setUnfocusFontStyle(
        styleCell_Table_Unfocus_Style);
  break;
  ...
```

The `setMboId` method binds the cell table to the MBO *department*. This also applies to other
MBO data controls such as *grid table*, *select box* and *list detail*.

Access Pending Operations and Operation Logs

MBOModel is an object for mobile business objects, and has a method called
`getPendingObjects` to return pending operations. A pending object is the combined
pending operations on that object instance.

```
CommonMBOModel mboInstance =
    MBOModelFactory.getInstance().getCommonMBOModel(mboId);
if ( !(mboInstance instanceof MBOModel) ) {
  continue;
}
MBOModel mobileApp = (MBOModel) mboInstance;
ObjectList pendingObjectList = mobileApp.getPendingObjects();
```

Similarly, the static method `getLogs` returns all logs for the specified package name and
query criteria.

```
Query query = new Query();
query.setTestCriteria(new AttributeTest("component", componentName,
```

```
AttributeTest.EQUAL));
LogRecord[] logs = MBOModel.getLogs(packageName, query);
```

Connecting to Unwired Server

`CommonMBOModel` defines `loginToSync` to log in to the Unwired Server.

```
CommonMBOModel.loginToSync(
    profile.getPackageName(),
    profile.getUserName(),
    profile.getPassword());
```

Adding or Modifying Navigation

To add navigation to a screen, you can add ScreenAction to controls or menus that take actions.

```
protected void configureObjectHandlersById(
    int ID, Object object) {
  switch (ID) {
  ...
  case MENU2:
    MenuAction menu2 = (MenuAction) object;
    //Create list of actions
    ActionList actionList15 = new ActionList();
    IBOBAction connectionAction9 = new
    ScreenAction(UIDefinition.getScreen("screen16"), false, null);
    menu2.setAction(actionList15);
    IBOBAction contextAction4
        = new SaveMobileDataContextAction(
            cellTable1);
    actionList15.addAction(contextAction4);
    actionList15.addAction(connectionAction9);
    ...
```

To remove navigation, you can override the menu or control's action to remove the screen action.

```
protected void configureObjectHandlersById(
    int ID, Object object) {
  switch (ID) {
  ...
  case MENU2:
    MenuAction menu2 = (MenuAction) object;
    ...
    menu2.setAction(xxxx);
  default:
    super.configureObjectHandlersById(ID, object);
```

For controls that support an action, you can create a widget event:

```
public boolean navigationClick(
    Object field, int controlID,
    ActionList actions, int status, int time)
{
  // custom code
  // modify actions which include the screen action
  return false;
}
```

Adding or Modifying Actions

You can add or modify actions for controls or menus through the
configureObjectHandlersById method.

Note: By default, the `configureObjectHandlersById` method adds all actions to controls and menus.

The following is a code example for adding actions:

```
protected void configureObjectHandlersById(int ID, Object object)
{
  switch (ID) {
    ...
  case BUTTON8:
    Button localbutton8 = (Button) object;
    //Create list of actions
    ActionList actionList3 = new ActionList();
    //Create set of submit elements
    Vector submit2 = new Vector();
    //Create submit element "dept_id"
    submit2.addElement(new SubmitElement("dept_id", "2",
        VariableProperties.SUBMIT_CONTROL_TYPE, null, true,
        null, -1, "dept_id", MBOAttribute.SCHEMA_TYPE_INT,
        false, null, false));
    //Create submit element "dept_name"
    submit2.addElement(new SubmitElement("dept_name", "4",
        VariableProperties.SUBMIT_CONTROL_TYPE, null,
        false, null, 40, "dept_name",
        MBOAttribute.SCHEMA_TYPE_STRING,
        false, null, false));
    //Create submit element "dept_head_id"
    submit2.addElement(new SubmitElement("dept_head_id", "6",
        VariableProperties.SUBMIT_CONTROL_TYPE, null,
        false, null, -1, "dept_head_id",
        MBOAttribute.SCHEMA_TYPE_INT,
        false, null, false));
    localbutton8.setAction(actionList3);
    IBOBAction submitAction2 = new SubmitAction(
        BOBCUIDefinition.MBO_A_B_C_DEPARTMENT, this,
        OperationTypes.OPERATION_INSERT, submit2, false,
        "Input {0} is required.",
        "Input {0} exceeds the maximum length of {1}.",
        "create");
    actionList3.addAction(submitAction2);
    IBOBAction backAction1 = BackAction.getInstance();
    actionList3.addAction(backAction1);
  break;
```

To modify actions, you can override this method by using custom code, and the widget event method `navigationClick`.

Creating and Assigning Variables

There are four types of variables: user, system, table and personalized.

To create user variables, add the variable in BOBCUIDefinition:

```
addVariable(VARIABLE_HISVAR, "hisVarValue",
  VariableProperties.VARIABLE_TYPE_USER,
  MBOAttribute.SCHEMA_TYPE_STRING);
```

To use a user variable in controls:

```
case LABEL2:
  Label locallabel2 = (Label) object;
  locallabel2.setFontStyle(styleLabel_Style);
  //locallabel2.setFooterField(null);
  locallabel2.setFocusFontStyle(styleDefault_Style);
  locallabel2.setWrapText(false);
  locallabel2.setVariableLabel(new ControlVariable(
      BOBCUIDefinition.VARIABLE_HISVAR,
      VariableProperties.VARIABLE_TYPE_USER, null,
      null));
```

To persist a user variable:

```
String key = "variableName";
        // NOTE: Sybase only supports this type.
        String keyType = VariableProperties.VARIABLE_TYPE_USER;
        String schemaType = MBOAttribute.SCHEMA_TYPE_BOOLEAN;
        Object value = "true";
        Util.addVariable( key, value, keyType, schemaType );
```

System variables are used in a similar manner.

To create table variable and bind to text input:

```
localtextInput4.setVariableInput(
    new ControlVariable("dept_head_id",
        VariableProperties.VARIABLE_TYPE_TABLE,
        BOBCUIDefinition.MBO_A_B_C_DEPARTMENT, null));
```

You must save a Context before table variables can be used by a context action.

```
case MENU6:
  MenuAction menu6 = (MenuAction) object;
  //Create list of actions
  ActionList actionList13 = new ActionList();
  IBOBAction connectionAction9 =
      new ScreenAction(
        UIDefinition.getScreen("screen4"), false, null);
  menu6.setAction(actionList13);
  IBOBAction contextAction4 =
      new SaveMobileDataContextAction(cellTable1);
  actionList13.addAction(contextAction4);
  actionList13.addAction(connectionAction9);
  cellTable1.setDefaultAction(actionList13);
```

Using PIM Actions

You can add a PIM action to a control to integrate the control with a BlackBerry PIM application. The PIM action constructor takes the following arguments.

`int type`:

```
public interface RIMPimConstants
{
    // #################### Available RIM applications
#################### //

    public static int RIM_PIM_CONTACT  = 0;
    public static int RIM_PIM_EMAIL    = 1;
    public static int RIM_PIM_PHONE    = 2;
    public static int RIM_PIM_EVENT    = 3;
    public static int RIM_PIM_TODO     = 4;
    public static int RIM_PIM_MEMO     = 5;
}
```

`boolean isRead`: if true, the constructor reads from the BlackBerry PIM application; if false the constructor writes to the BlackBerry PIM application.

Object control: A
`com.sybase.uep.bobclient.controls.MobileDataControl` widget such as
`com.sybase.uep.bobclient.controls.MobileAppTable`,
`com.sybase.uep.bobclient.controls.TwoColumnLayout` or
`com.sybase.uep.bobclient.screens.IBOBScreen`.

The following code example shows the use of the PIM read action, which requires the control's logical type to be set as one of the PIM application type, and the creation of a PIM action that refers to the control or the control's parent screen.

```
TextInput textInput = new TextInput("", "",
    BasicEditField.DEFAULT_MAXCHARS, Field.FIELD_LEFT);
// set logical type of text input, this is important for PIM usage
textInput.setLogicalType(
    new LogicalType(
        RIMPimConstants.RIM_PIM_CONTACT,
        BlackBerryContact.NAME,
        PIMItem.ATTR_NONE, Contact.NAME_GIVEN));
layoutManager.addWidget(textInput1, 1, 1);
Button button2 = (Button) object;
// Create list of actions
ActionList actionList2 = new ActionList();
button2.setAction(actionList2);
// pass the text input's parent screen to PIM action
Action rimAction2 =
    new RIMPimAppAction(
        RIMPimConstants.RIM_PIM_CONTACT, true, this, false);
actionList2.addAction(rimAction2);
layoutManager.addWidget(button2, 1, 1);
```

The following is a code example for the PIM write action:

```
TextInput textInput3 = (TextInput) object;
textInput3.setInvalidValueMessage(
    "Enter a valid value (data type {0} and logical type {1}).");
textInput3.setLogicalType(
    new LogicalType(
        RIMPimConstants.RIM_PIM_CONTACT,
        BlackBerryContact.NAME,
        PIMItem.ATTR_NONE, Contact.NAME_FAMILY));
layoutManager.addWidget(textInput3, 1, 1);
Button button4 = (Button) object;
//Create list of actions
ActionList actionList2 = new ActionList();
button4.setAction(actionList2);
Action rimAction2 = new RIMPimAppAction(
    RIMPimConstants.RIM_PIM_CONTACT, false, this,
    true);
actionList2.addAction(rimAction2);
layoutManager.addWidget(button4, 1, 1);
```

boolean launchPIMApp: if *true*, the PIM application gets launched after performing a
write operation, otherwise *false*.

```
case MENU13:
  MenuAction menu13 = (MenuAction) object;
  Action rimAction9 =
      new RIMPimAppAction(
          RIMPimConstants.RIM_PIM_CONTACT, true, this,
          false);
  menu13.setAction(rimAction9);
  break;
```

Using LayoutManager
These examples illustrate how to customize margin and button layout.

```
int[] colPercentages = { 35, 65 };
LayoutManager layoutManager = new LayoutManager(2,
    colPercentages, true, styleDisplay_Style);
...
layoutManager.addWidget(textInput52, 1, 1);
layoutManager.addWidget(horizontalRuler53, 2);
layoutManager.addWidget(button54, 1, 1);
layoutManager.addWidget(button55, 1, 1);
this.add(layoutManager);
```

Adding a Table Header
Add a Region which contains the table header columns for the screen that you want to
customize using the Device Application Designer. Then modify the generated gap class:

```
protected void createControls() {
        super.createControls();

        if (regionManager1.getManager() != null) {
            Manager m = regionManager1.getManager();
            m.delete(regionManager1);
            layoutManager.delete(m);
```

```
        }

        setBanner(null);
        VerticalFieldManager vfm = new VerticalFieldManager();
        vfm.add(navigationBarField);

regionManager1.setColumnPercentage(cellTable1.getTableConfig()
            .getColumnPercentage());
        vfm.add(regionManager1);
        layoutManager.setScreenHeader(vfm);
        setBanner(vfm);
    }
```

Filling a Space with a Button
By default, the generated button adjusts its width based on the displayed content and does not fill any extra space.

To change the button so it fills the extra space:

1. Create a customized button class which uses all the available layout space:

```
public int getButtonWidth() {
        //always use layout width to fill the space
        if ( _layoutWidth  > 0 )
        {
            return _layoutWidth;
        }
        return super.getButtonWidth();
    }

    public void setLayoutWidth(int width) {
            super.setLayoutWidth(width);
        _layoutWidth = width;
    }
```

2. Use the new FillButton class in the generated gap class:

```
// customize BOBScreenUpdate_Sales_order
    protected Object createControlById(int ID) {
        switch (ID) {
        case BUTTON16:
            //Create button "Submit"
          Button localbutton16 = new FillButton(Field.FIELD_RIGHT
                    | Field.FIELD_VCENTER);

            return localbutton16;
        case BUTTON17:
            //Create button "Cancel"
          Button localbutton17 = new FillButton(Field.FIELD_LEFT
                    | Field.FIELD_VCENTER);

            return localbutton17;
        default:
            break;
        }
```

```
        return super.createControlById(ID);
    }
```

Removing the CellTable Margin

By default, the CellTable is surrounded with a border. To remove it, customize the generated gap class:

```
protected void configureObjectMetaDataById(int ID, Object object) {
        super.configureObjectMetaDataById(ID, object);
        switch (ID) {
        case LAYOUTMANAGER:
          LayoutManager locallayoutManager = (LayoutManager) object;
            //remove margin for cell table
            locallayoutManager.setMarginWidth(0);
            locallayoutManager.setMarginHeight(0);
            break;
        default:
            break;
        }
    }
```

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 devices (as opposed to simulators).

You can implement code signing from the Device Application Designer or the BlackBerry JDE:

*   Device Application Designer – the BlackBerry generation wizard includes a "Disable signing" option. If this option is unselected the wizard presents a signing dialog after compiling. This dialog allows you to input a sign key to sign the options and the client COD (.cod) file, so they can run on a physical device.

- 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 *http://www.blackberry.com/DevMediaLibrary/view.do?name=deploying*.

Deploying BlackBerry Applications

You can deploy BlackBerry applications to physical devices through BlackBerry Desktop Manager or over-the-air (OTA).

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:

- OTA installation – requires all the JAD and COD files located in the `ota` subdirectory of the BlackBerry build version. For example: `<UnwiredPlatform_InstallDir>`
`\UnwiredPlatform\Unwired_WorkSpace\Eclipse`
`\sybase_workspace\mobile\eclipse\plugins`
`\com.sybase.uep.bob.rim_1.5.2.<timestamp>\generate`
`\blackberry\build-<version>\ota`.
- Desktop Manager installation – requires ALX and COD files located in the `BlackBerry<buildversion>` subdirectory. For example:
`<UnwiredPlatform_InstallDir>\UnwiredPlatform`
`\Unwired_WorkSpace\Eclipse\sybase_workspace\mobile\eclipse`
`\plugins\com.sybase.uep.bob.rim_1.5.2.<timestamp>\generate`
`\blackberry\build-<version>`.

Required files include:

- UltralLiteJ.cod/.alx/.jad files
- BOBFramework.cod/.alx/.jad files
- SUPPushListener.cod/.alx/.jad files

In addition to these generated files:

- client.cod/.alx/.jad files
- options.cod/.alx/.jad files

Deploy Applications through BlackBerry Desktop Manager
BlackBerry Desktop Manager allows you to customize synchronization and configuration settings between a desktop PC and a BlackBerry device, and to deploy third-party applications.

1. Connect your BlackBerry to your PC.
2. Once the device is attached, launch the BlackBerry Desktop Manager.

3. Launch the Application Loader from inside the Desktop Manager. Select the alx files on the PC and deploy them to the device.

4. Click the **Add** button in the Application Loader. Browse and select each of the Application Loader Files (.alx).
   The files appear in the list of items to install.

5. Follow the instructions to complete the installation.

   Note: For more information on using BlackBerry Desktop Manager, see the RIM documentation.

Deploying Applications Over the Air

There is an OTA folder at the generated location, which includes all of the .cod files, split into a number of .cod files of smaller size for the purpose of OTA deployment. You can also find .jad files in that location, which are required for OTA deployment.

1. Place all the files on a Web server that supports an HTTP connection.

2. BlackBerry users access the Web server and download the applications by selecting the corresponding .jad files.

Reference

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

BlackBerry Client Object API

Describes solutions and examples for tasks and uses of the Sybase Unwired Platform BlackBerry Client Object API. The Client Object API enables you to customize mobile business object data flow and handling for the BlackBerry device application.

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>\Servers \UnwiredServer\ClientAPI\apidoc`. There are subdirectories for `\j2me` and `\j2se`.

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");
```

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" );
```

To connect to Unwired Server through Relay Server, set the required parameters with the `setNetworkStreamParams` method.

```
profile.setNetworkStreamParams
("url_suffix=ias_relay_server/client/rs_client.dll/Ryan.SUPFarm");
```

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. The client application should put the call to

submitPending within a try/catch block and should not attempt to send any more messages to the server if loginToSync throws an exception.

```
<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");
```

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);
```

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);
```

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 3. 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.

AttributeTest

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.

An AttributeTest defines a filter condition using an MBO attribute, and supports these 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);
```

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);
```

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));
}
```

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();

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:

```
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.

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.

```
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 Remedy 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 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).

Name	Java Type	Description
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.
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.

Name	Java Type	Description
replayPending	long	Returns a long value. When a pending row is submitted to the server, the value of replayCounter is copied to re-playPending. 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 re-playFailure 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.

Method Name	Java Type	Description
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
```

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" );
profile.setNetworkStreamParams
( "trusted_certificates=rsa_public_cert.crt");
```

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.

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

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());
...
}
```

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 4. 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.
int onSynchronize(com.sybase.collections.Ob- jectList groups,SynchronizationContext context)	This method will be invoked at the specified sta- tus of the synchronization.*groups* is a list of syn- chronization group names. *context* is the syn- chronization context.
void onSynchronizeFailure(com.sybase.collec- tions.ObjectList groups)	Synchronization failure notification. *groups* is a list of synchronization group names.
void onSynchronizeSuccess(com.sybase.collec- tions.ObjectList groups)	Synchronization success notification. *groups* is a list of synchronization group names.

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.

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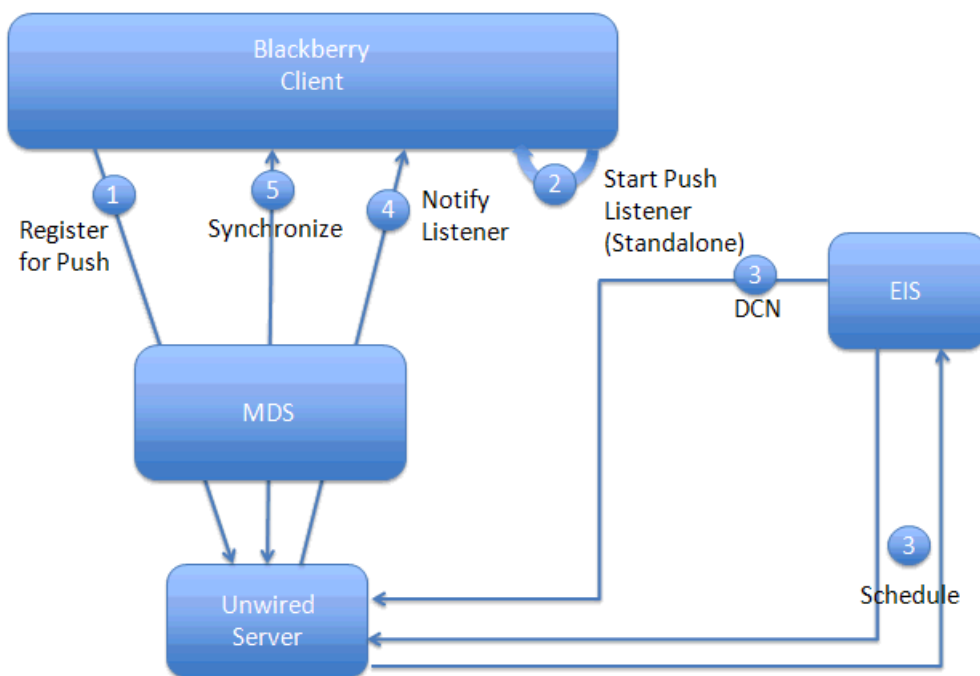
The HTTP push gateway can also be used for network-connected Sybase Unwired Platform applications (for example the Java desktop). The address of the subscription contains an HTTP URL to an HTTP listener which the application creates. The URL contains a query parameter such as:

```
&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.

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.

You can develop the push application directly from generated mobile business object (MBO) code, or from the Device Application Designer.

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.
   You can either develop the push application directly from MBO generated code or by using the Device Application Designer:
   • Develop the application directly from MBO code:
      1. Generate the Object API code.
      2. 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");
        }
    }
}
```

3. 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;
}
```
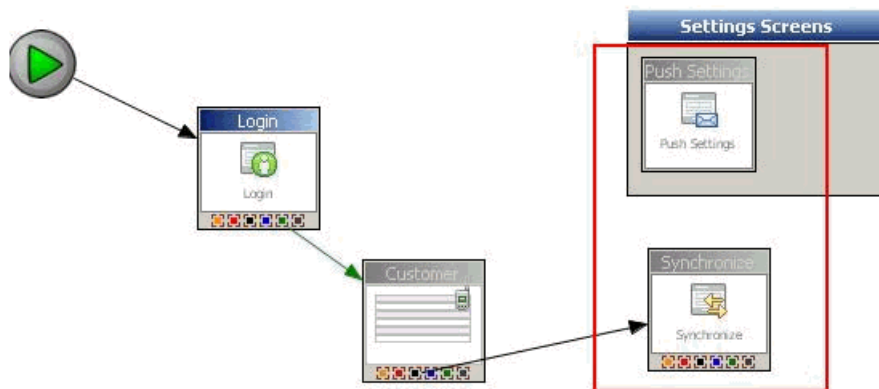
- Develop the application in the Device Application Designer:
  1. Add the Push Settings and Synchronization stock screen:

2. Generate the device application.
3. Run the application in the simulator or on a device and set the appropriate settings.
4. In the Push Settings screen define the MDS server and port. For example:
   - MDS Server: localhost
   - MDS Port: 8080
5. In the Synchronization screen, select a synchronization group and click **Menu > Synchronization Group Info Screen** to navigate to the synchronization group info screen. Select **Enable push** and select a **Push notification mode** (Sync Immediately, Prompt and sync if there is no response, Prompt but don't sync if there is a response, or Don't prompt and don't sync).
6. Select **Menu > Save** to upload the push registration to the server.

BlackBerry Device Framework API

Describes solutions and examples for tasks and uses of the Sybase Unwired Platform BlackBerry Device Framework API, which lets you customize the BlackBerry device user interface.

The Device Framework works as a library that supports the running of device client applications. The main class for a device client is BOBUIController. This class is the entry point when a client is first launched, and keeps track of screens and push actions. The BOBUIController class extends UiApplication, providing methods to add Device Application Designer screens, and push those screens to the stack. It also provides methods to remove screens, invoke objects that run code, and respond to events.

The Device Framework also provides a number of synchronization classes for handling mobile business object (MBO) synchronization, including push sync, as well as synchronization group sync. A couple of utility classes are there for various purposes. See the Java doc for details. Both BOB options and client module depend on the Device Framework.

The BOB options module includes generated object API code and screen registration code. BOB client is the main UI application and it also depends on the options module.

Note: For information on BlackBerry development using the RIM APIs, see the RIM documentation.

BlackBerry Device Framework API Javadocs

Use the Sybase BlackBerry Device Framework API Javadocs as a Device Framework API reference.

Review the reference details in the BlackBerry Device Framework API Javadocs. To access the Framework API Javadocs click *BOBFrameworkJavadoc.zip* and select **Save**. Save the file to a directory on your local machine and unzip it.

Click the index.html file, the Javadoc open in a browser. The top left navigation pane lists all packages installed with Unwired Platform. The applicable documentation is available with each package. Navigate through the Javadoc as required.

Screen Objects

The main structure for a BlackBerry device user interface is the screen object.

All screens that you create are generated into Device Application Designer screens, which extend the BaseBOBScreen class. BaseBOBScreen is the base implementation of IBOBScreen. All Device Application Designer screens implement this interface.

Users cannot change the layout of stock screens; however, they can modify certain exposed stock properties.

Table 5. Device Application Designer Screen Types

Screen Type	Class	Description
Base Device Application Designer Screen	BaseBOBScreen	A base implementation of IBOBScreen that is used by most screens in the framework. It provides methods for adding menus, menu separators, and source screen references or spacers to the current screen, as well as deleting menu actions. It also creates menu items based on current focused controls.
Base Device Application Designer Stock Screen	BaseBOBStockScreen	The base Device Application Designer stock screen is used by most stock screens in the framework.

Screen Type	Class	Description
Base Device Application Designer Settings Screen	BaseSettingsScreen	The base Device Application Designer settings screen is used by most settings screens in the framework.
About Screen	AboutScreen	Information about the Sybase Unwired Platform BlackBerry client application.
Date Picker Screen	DatePickerPopup	Both shows, and allows the user to specify, date and time information.
Exception Screen	ExceptionScreen	Internal server errors.
File Explorer Screen	FileExplorerScreen	Allows the user to select a file from a device.
Image Screen	ImageScreen	Shows the entire image on the screen.
LoginScreen	LoginScreen	Login information for the Sybase Unwired Platform BlackBerry client application.
Logs Screen	LogsScreen	Logs for the Sybase Unwired Platform BlackBerry client application.
Log Information Screen	LogInfoScreen	Log information for the Sybase Unwired Platform BlackBerry client application.
Pending Operation Screen	PendingOperationScreen	Pending operations for the Sybase Unwired Platform BlackBerry client application.
Pending Operation Information Screen	PendingOperationInfoScreen	Pending operation information for the Sybase Unwired Platform BlackBerry client application.
Personalization Screen	PersonalizationScreen	Personalization information for the Sybase Unwired Platform BlackBerry client application.
Profile Screen	ProfileScreen	Profiles for the Sybase Unwired Platform BlackBerry client application.
Profile Update Screen	ProfileUpdateScreen	Profiles information for the Sybase Unwired Platform BlackBerry client application.
Push LogsScreen	PushLogsScreen	Push logs for the Sybase Unwired Platform BlackBerry client application.

Screen Type	Class	Description
Push Request Screen	PushRequestScreen	Push request for the Sybase Unwired Platform Black-Berry client application.
Push Settings Screen	PushSettingsScreen	Push settings for the Sybase Unwired Platform Black-Berry client application.
Screen Saver Screen	ScreenSaverScreen	Screen saver for the Sybase Unwired Platform BlackBer-ry client application.
Search Screen	SearchScreen	Search information for the Sybase Unwired Platform BlackBerry client application.
Synchroniza-tion Group In-formation Screen	SyncronizationGrou-pInfoScreen	Synchronization Group information for the Sybase Un-wired Platform BlackBerry client application.
Synchroniza-tion Screen	SynchronizationScreen	Synchronization information for the Sybase Unwired Platform BlackBerry client application.

This example illustrates how to customize stock/settings screen properties:

```
//Define and register the screen
screen5 = new AboutScreen("About", "untitled1");

//Section to define the stock screen properties
screen5.setProperty(ABOUT_SCREEN_LOGO_IMAGE, "blackberry_16.gif");
screen5.setProperty(ABOUT_SCREEN_VERSION_LABEL, "MyVersion");
```

Control Objects

Control objects represent all of the user interface (UI) components on the Device Application Designer screens, which are rectangular regions that a Manager controls. A control's layout requirements determine the control size. Managers provide scrolling for the fields that they contain.

Table 6. Device Application Designer Control Types

Control Type	Class	Description
Button	Button	Extends Field and displays the assigned text or image. You can set a focus on a button control; thus, you can assign different styles to it for focused and unfocused. You can also bind a button to a list of actions, which run when the trackball is clicked or touched on touchable devices.
Cell Image	CellImage	Extends ImageControl. You can bind CellImage to a MBO attribute, value mapping, state indicator, variable, or image.
Cell Label	CellLabel	Extends Label. You can assign styles to it for focus and unfocus status. You can bind CellLabel to MBO attributes, literals, or variables.
Cell Table	CellTable	Extends ListField and displays the data of the assigned MBO. Cell table accommodates cell labels and cell images, which can be bound to MBO attributes. In addition, cell image can be configured to show a different image based on the specified attribute's value. You can assign styles to cell table for focus and unfocus status. You can bind CellTable to a MBO, whose values are filled into the table.
Check Box	CheckBox	Extends CheckboxField.
Grid Table	MobileAppTable	Extends Field and displays the data of the assigned MBO. The table header displays the MBO attributes, while the cells display the corresponding value. Styles can be assigned to Mobile App table for header, odd row, even row, pending row, border, focused cell. You must bind MobileAppTable to a MBO, for which the values are filled into the table.
Horizontal Ruler	HorizontalRuler	Draws a line across the screen.

Control Type	Class	Description
Hyperlink	HyperLink	Extends LabelField and displays the assigned text with underscore. You can set a focus on a hyperlink control; thus, you can assign different styles to it for focused and unfocused. You can also bind Hyperlink to a list of actions, which run when the trackball is clicked or touched on touchable devices.
Hyperlink Rich Field	HyperLinkRichField	The difference between HyperLinkRichField and Hyper-Link is that HyperLinkRichField uses Blackberry rich context matching to handle phone and email options.
Image Control	ImageControl	Extends BitmapField and displays the assigned image. You can bind ImageControl to a list of actions, which run when the trackball is clicked or touched on touchable devices.
Label	Label	Extends LabelField and displays the assigned text.
List Detail	TwoColumnLayout	Extends LayoutManager, must be bound to a MBO, and can display the details for the assigned MBO in a two column layout. As with MobileAppTable, odd row style, even row style, pending style, border style, and focus style are available. You can assign hot keys as well as menu labels to previous/next menus of this control.
List Item	ListFieldControl	Extends ListField and displays the assigned text or image. You can set a focus on a list item control; thus, you can assign different styles to it for focused and unfocused. You can also bind a list item to a list of actions, which run when the trackball is clicked or touched on touchable devices.
Navigation Bar	NavigationBar	Extends BitmapField and allows users to navigate client screens and keep track of where they are.
Radio Box	RadioBox	Extends RadioButtonField and displays the assigned text. RadioBox is contained in the RadioBoxGroup.
Select Box	SelectBox	Extends ObjectChoiceField and displays the assigned texts. You can bind SelectBox to a MBO, whose values are filled into the select box.
Spacer	Spacer	Extends Field and shows nothing.

Control Type	Class	Description
Text Input	TextInput	Extends EditField and displays the assigned texts. You must assign a data type to TextInput, the default type is STRING. You can also assign it a logical type for personal information management (PIM). Validations are also available.
Toolbar Item	Toolbar Item	Extends Button control. You must add it to Toolbar. Toolbar Item is only available on touch screen devices. Toolbar item can be assigned image and actions.

This example illustrates how to create a control:

```
CellImage cellImage = new CellImage(Field.FIELD_LEFT
| Field.FIELD_VCENTER);

    CellLabel cellLabel = new CellLabel(Field.FIELD_LEFT
| Field.FIELD_VCENTER);

CellTable cellTable = new CellTable(Field.FIELD_LEFT);
```

This example illustrates how to configure a control:

```
CellTable localcellTable1 = (CellTable) object;
//Create set of submit elements
Vector submit1 = new Vector();
submit1.addElement(new SubmitElement("parameter1",
        "2010-06-07", VariableProperties.SUBMIT_USER_TYPE,
        null, false, null, -1, "startDate",
        MBOAttribute.SCHEMA_TYPE_DATE, false, null, false));
localcellTable1.setColumnPercentage(new int[] { 10, 60, 30 });
localcellTable1.setSortingColumn("Sort on column");
localcellTable1.setMboId(BOBCUIDefinition.MBO_POC_ACTIVITY);
localcellTable1.setNamedQuerySubmitElements(submit1);
localcellTable1.setNamedQuery("findByDate");
localcellTable1.setFocusFontStyle(styleCell_Table_Focus_Style);
localcellTable1.setNumberOfColumns(3);
localcellTable1.setColumnConfig(new String[] { "description",
        "status", "actType" });
localcellTable1
        .setUnfocusFontStyle(styleCell_Table_Unfocus_Style);

CellImage localcellImage0 = (CellImage) object;
localcellImage0
        .setImageType(ICellAttributeTypeConstants.IMAGE_VALUEMAPPING
_TYPE);
localcellImage0.setPreserveAspectRatio(true);
localcellImage0.setOrder(0);
localcellImage0.setMboAttrId("typeCode");
```

Layout Manager Objects

Use the layout manager objects to add and position controls on the screen. Device Application Designer provides a number of layout managers for laying out controls.

Table 7. Device Application Designer Layout Manager Types

Layout Manager Type	Class	Description
Row Layout	RowLayout	Extends Manager, and is the base BOB layout manager. All other layout managers leverage this layout manager to lay out controls. RowLayout lays out the controls based on assigned horizontal/vertical spans and the width/height of the controls.
Layout Manager	LayoutManager	Extends Manager, and lay outs the controls that use RowLayout. LayoutManager first creates a RowLayout based on assigned horizontal/vertical spans. When the first RowLayout is full, LayoutManager creates another RowLayout for the rest of the controls.
Region Manager	RegionManager	Extends LayoutManager.
Tab Content Panel	TabContentPanel	The TabContentPanel extends Manager and is used in TabLayoutManager.
Tab Control Layout	TabControlLayout	Extends HorizontalFieldManager and provides methods to add tabs and switch tabs. It is used in TabLayoutManager.
Tab Layout Manager	TabLayoutManager	Extends Manager and lays out controls for tab panels.
Toolbar	ToolbarManager	Extends VerticalFieldManager to accommodate toolbar items. You can assign styles to Toolbar for style and border style. Toolbar is only available to touch screen devices.

This example illustrates how the createControls method is added to the screen:

```
layoutManager = (LayoutManager) createControlById(LAYOUTMANAGER);
label1 = (Label) createControlById(LABEL1);
....
configureControlById(LAYOUTMANAGER);
configureControlById(LABEL1);
```

```
...
layoutManager.addWidget(label1, 1, 1);
...
this.add(layoutManager);
```

Action Objects

Device Application Designer actions can be executed as a specific program or instruction, and are bound to controls such as Button, Hyperlink, Image, or List Item.

You can assign single or multiple actions to each of these controls. All Device Application Designer actions are implementations of the IBOBAction interface.

Table 8. Device Application Designer Action Object Types

Action Type	Class	Description
Action	Action	Action is the base implementation of IBOBAction; its methods include hasFailed, isProcessing for monitoring the action status, and the run method to execute the action.
Action List	ActionList	Extends Action and holds a vector of actions. Its run method executes all the actions.
Alert Action	AlertAction	Shows an alert informational message dialog.
Alert Error Action	AlertErrorAction	Shows an alert error message dialog.
Alert Question Action	AlertQuestionAction	Shows an alert question message dialog.
Navigate Back Action	BackAction	BackAction pops current screen to display the parent screen.
Close Screen Action	CloseScreenAction	Closes the current screen.
Exit Action	ExitAction	Exits the current application.
Google Map Action	GoogleMapAction	Invokes Google Map and locates the address.
Lock Client Action	LockClientAction	Locks the current application.
Login Action	LoginAction	Logs in the user.
Logout Action	LogoutAction	Logs out the current user.

Action Type	Class	Description
Object Query Action	NamedQueryAction	Executes the assigned object query.
Persist Action	PersistAction	Saves all the variables in a form.
Refresh Action	RefreshAction	Refreshes the current focused screen or supplied screen.
RIM PIM Application Action	RIMPimAppAction	Allows the read and write of data in personal information management (PIM) databases from a RIM client application, including e-mail, contacts, phone, or to-do lists.
Save Mobile Data Context Action	SaveMobileDataContextAction	Saves current mobile data control context to memory.
Screen Action	ScreenAction	Goes to another screen.
Submit Action	SubmitAction	Creates a update/insert/delete/others operation.
Synchronization Action	SyncAction	Performs synchronization actions on all MBOs within the synchronization group.
Synchronization Publication Action	SyncPublicationAction	Performs sync actions on specific publications (Publication > Synchronization group).
Tab Action	TabAction	Controls the tab layout manager to switch to different tabs.
Close Screen Action	CloseScreenAction	CloseScreenAction closes the current screen.

This example illustrates adding and modifying an action:

```
protected void configureObjectHandlersById(int ID, Object object) {
switch (ID) {
......
case BUTTON8:
    Button localbutton8 = (Button) object;
    //Create list of actions
    ActionList actionList3 = new ActionList();
    //Create set of submit elements
    Vector submit2 = new Vector();
    //Create submit element "dept_id"
    submit2.addElement(new SubmitElement("dept_id", "2",
            VariableProperties.SUBMIT_CONTROL_TYPE, null, true, null,
            -1, "dept_id", MBOAttribute.SCHEMA_TYPE_INT, false, null,
            false));
    //Create submit element "dept_name"
    submit2.addElement(new SubmitElement("dept_name", "4",
```

```
        VariableProperties.SUBMIT_CONTROL_TYPE, null, false, null,
          40, "dept_name", MBOAttribute.SCHEMA_TYPE_STRING, false,
          null, false));
    //Create submit element "dept_head_id"
    submit2.addElement(new SubmitElement("dept_head_id", "6",
VariableProperties.SUBMIT_CONTROL_TYPE, null, false, null,
          -1, "dept_head_id", MBOAttribute.SCHEMA_TYPE_INT, false,
          null, false));
    localbutton8.setAction(actionList3);
    IBOBAction submitAction2 = new SubmitAction(
          BOBCUIDefinition.MBO_A_B_C_DEPARTMENT, this,
          OperationTypes.OPERATION_INSERT, submit2, false,
          "Input {0} is required.",
         "Input {0} exceeds the maximum length of {1}.", "create");
    actionList3.addAction(submitAction2);
    IBOBAction backAction1 = BackAction.getInstance();
    actionList3.addAction(backAction1);

          break;
```

This example illustrates a PIM action. The PIM action constructor takes an int type argument:

```
public interface RIMPimConstants
{
    // #################### Available RIM applications
#################### //

public static int RIM_PIM_CONTACT  = 0;
public static int RIM_PIM_EMAIL    = 1;
public static int RIM_PIM_PHONE    = 2;
public static int RIM_PIM_EVENT    = 3;
public static int RIM_PIM_TODO     = 4;
public static int RIM_PIM_MEMO     = 5;
}
```

- boolean isRead:
  - True indicates reading from the BlackBerry PIM application
  - False indicates writing to the BlackBerry PIM application
- Object control – Can be the
  com.sybase.uep.bobclient.controls.MobileDataControl widgets
  such as the com.sybase.uep.bobclient.controls.MobileAppTable,
  com.sybase.uep.bobclient.controls.TwoColumnLayout or
  com.sybase.uep.bobclient.screens.IBOBScreen.
- boolean launchPIMApp – true launches the PIM application after performing a
  write operation, otherwise false.

```
case MENU13:
          MenuAction menu13 = (MenuAction) object;
          Action rimAction9 = new RIMPimAppAction(
                  RIMPimConstants.RIM_PIM_CONTACT, true, this,
false);
          menu13.setAction(rimAction9);
          break;
```

Data Objects

Device Application Designer provides layers for wrapping data, such as styles, variables, and mobile business objects (MBOs).

Table 9. Device Application Designer Data Object Types

Data Type	Class	Description
Variable	ControlVariable	Holds the variable attributes for controls. It includes type (USER for user-defined variables, SYSTEM for system-defined variables, TABLE for table context variables), key (variable key), and MBO id (for table context variables).
Variable Management	RIMVariables	Manages variables, including store/access variables.
Style	FontStyle	Holds the font information, including font face, font size, font size unit, font style, background color, foreground color, and gradient color.
Logical Type	LogicalType	Contains all the logical type information. Also contains personal information management (PIM) information if applicable.
MBO	CommonMBOModel MBOModel PKMBOModel	CommonMBOModel contains information about MBOs and their subclasses, including normal MBO, personalization MBO, and local business object. CommonMBOModel provides methods for the MBO, including `submitPendingOperations` and `syncPublication`. Other important classes in this package include: • ModelChangeEvent • MobileApplicationDataHandler • MobileApplicationDataPagingHandler • MBOModelSyncParameters
MBO Attribute	MBOAttribute	Contains information about MBO attribute ID, display name and datatype.
Client Profile	RIMClientProfile	Contains client profile information, including profile name, server name, server port, user name, password, package name, stream parameters, and so on.

Data Type	Class	Description
Link Parameter	RIMLinkParamNode	Contains link parameter information.
MBO Application	RIMMBOMobileApplication	Represents the MBO in Device Application Designer styles.
Repository	RIMRepository	The central place for store or accessing the client profile, client settings, variables, MBO applications, and other settings.
Settings	RIMSettings	Contains various settings information, including the push settings, screen saver settings, log level settings.
Validation Object	RIMValidationObject	Manages validation information, including regular expressions and messages.

These examples illustrate how to assign and read variables.

Adds a variable in BOBCUIdefinition:

```
addVariable(VARIABLE_HISVAR, "hisVarValue",
    VariableProperties.VARIABLE_TYPE_USER,
    MBOAttribute.SCHEMA_TYPE_STRING);
```

Use variables in controls:

```
case LABEL2:
Label locallabel2 = (Label) object;
locallabel2.setFontStyle(styleLabel_Style);
//locallabel2.setFooterField(null);
locallabel2.setFocusFontStyle(styleDefault_Style);
locallabel2.setWrapText(false);
locallabel2.setVariableLabel(new ControlVariable(
        BOBCUIDefinition.VARIABLE_HISVAR,
        VariableProperties.VARIABLE_TYPE_USER, null, null));
```

Use table variables:

```
localtextInput4.setVariableInput(new ControlVariable(
        "dept_head_id", VariableProperties.VARIABLE_TYPE_TABLE,
        BOBCUIDefinition.MBO_A_B_C_DEPARTMENT, null));
```

Context variables must be saved before table variables are used by context actions:

```
case MENU6:
MenuAction menu6 = (MenuAction) object;
//Create list of actions
ActionList actionList13 = new ActionList();
IBOBAction connectionAction9 = new ScreenAction(UIDefinition
        .getScreen("screen4"), false, null);
menu6.setAction(actionList13);
IBOBAction contextAction4 = new SaveMobileDataContextAction(
        cellTable1);
actionList13.addAction(contextAction4);
```

```
actionList13.addAction(connectionAction9);
cellTable1.setDefaultAction(actionList13);
```

Constant Classes

A number of constant classes are defined.

Table 10. Device Application Designer Constant Types

Constant Type	Class	Description
Styles	FontStylesProperties	Defines a number of default styles are defined, including DEFAULT_SCREEN_FONT_STYLE, and FONT_STYLE_HYPERLINK_UNFOCUS.
Literals	Literals	Defines all literals for the Device Application Designer framework.
RIM PIM Constants	RIMPimConstants	PIM-related constants.
Stock/Settings Screen Constants	ScreenProperties	Stock/Settings screen properties.
Validator Constants	ValidatorConstants	Constants for validators.
Variable Properties	VariableProperties	Constants for variables.

Generated Client Code

After you design a Device Application Designer application, and generate the device application from that document, each screen is generated into a class that extends BaseBOBScreen, and each tab panel is generated into a class that extends LayoutManager, where various controls are defined, as well as actions and menus.

An additional BOBCUIDefinition is generated to keep track of user defined variables, styles, MBOs, screens, and so on. The BOBCUIDefinition as well as the screen classes are compiled against the Device Framework and the BOBUIController to produce the final client application. BOBCOptionsDefinition is generated to keep track of settings screens, MBO packages, profiles, and so on. The class and OptionsMain and generated object API code are compiled into options module. The generated client code serves as good sample code to illustrate the usage of the Device Framework.

If you select **Generate custom coding subclass** for device/options/screen/tab panel classes during device application code generation, a subclass is generated with methods skeleton for you to add custom code, as these examples illustrate.

Super class:

```
protected Object createControlById(int ID) {
    switch (ID) {
            ......
    case CELLTABLE1:
    CellTable localcellTable1 = new CellTable(Field.FIELD_LEFT);

return localcellTable1;
......
```

Custom subclass:

```
protected Object createControlById(int ID) {
switch (ID) {
case CELLTABLE1:
    CellTable localcellTable1 = new CellTable(Field.FIELD_LEFT) {
        public boolean keyChar(char key, int status, int time) {
            // TODO sym key seems ignored
            boolean retval = false;
            switch (key) {
            case Characters.ENTER:
            case Characters.NULL:
            case Characters.CONTROL_SYMBOL:
            case Characters.CONTROL_UP:
            case Characters.CONTROL_VOLUME_DOWN:
            case Characters.CONTROL_VOLUME_UP:
            case Characters.TAB:
                retval = super.keyChar(key, status, time);
                break;
            case Characters.DELETE:
            case Characters.BACKSPACE:
                String text = customNaviBar.getFindText();
                if (text.length() > 0) {
                customNaviBar.setFindText(text.substring(0, text
                .length() - 1));
                } else {
                customNaviBar.setFindText("");
                }
                break;
            default:
                customNaviBar.setFindText(customNaviBar.getFindText()
                   + key);
                break;
            }
            return retval;
        }
    };
    return localcellTable1;
default:
    return super.createControlById(ID);
```

Overriding metadata - super class:

```
protected void configureObjectMetaDataById(int ID, Object object) {
        switch (ID) {
      ......
```

```
case CELLTABLE1:
    if (object instanceof CellTable) {
    CellTable localcellTable1 = (CellTable) object;
    //Create set of submit elements
    Vector submit1 = new Vector();
    submit1.addElement(new SubmitElement("parameter1",
            "2010-06-07", VariableProperties.SUBMIT_USER_TYPE,
            null, false, null, -1, "startDate",
            MBOAttribute.SCHEMA_TYPE_DATE, false, null, false));
    localcellTable1.setColumnPercentage(new int[] { 10, 60, 30 });
    localcellTable1.setSortingColumn("Sort on column");
    localcellTable1.setMboId(BOBCUIDefinition.MBO_POC_ACTIVITY);
    localcellTable1.setNamedQuerySubmitElements(submit1);
    localcellTable1.setNamedQuery("findByDate");
    localcellTable1.setFocusFontStyle(styleCell_Table_Focus_Style);
    localcellTable1.setNumberOfColumns(3);
    localcellTable1.setColumnConfig(new String[] { "description",
            "status", "actType" });
localcellTable1
            .setUnfocusFontStyle(styleCell_Table_Unfocus_Style);

}
        break;
      ......
```

Subclass in which the cell table's named query is set to null:

```
protected void configureObjectMetaDataById(int ID, Object object) {
    switch (ID) {
case CELLTABLE1:
super.configureObjectMetaDataById(ID, object);
if (object instanceof CellTable) {
CellTable localcellTable1 = (CellTable) object;
localcellTable1.setNamedQuery(null);
}
break;
    default:
        super.configureObjectMetaDataById(ID, object);
    }
 }
```

Override handler superclass:

```
protected void configureObjectHandlersById(int ID, Object object) {
switch (ID) {
  ......
case MENU8:
if (object instanceof MenuAction) {
    MenuAction menu8 = (MenuAction) object;
    //Create list of actions
    ActionList actionList17 = new ActionList();
    IBOBAction connectionAction7 = new ScreenAction(UIDefinition
            .getScreen("screen34"), false, null);
    menu8.setAction(actionList17);
    IBOBAction contextAction7 = new SaveMobileDataContextAction(
            cellTable1);
    actionList17.addAction(contextAction7);
```

```
    actionList17.addAction(connectionAction7);

    }
break;
.....
}
```

Subclass in which the menu's actions are overridden:

```
protected void configureObjectHandlersById(int ID, Object object) {
switch (ID) {
case MENU8:
 if (object instanceof MenuAction) {
  MenuAction menu8 = (MenuAction) object;
  //Create list of actions
     ActionList actionList17 = new ActionList();
  menu8.setAction(actionList17);
  IBOBAction contextAction7 = new SaveMobileDataContextAction(
    cellTable1);
  actionList17.addAction(contextAction7);
  actionList17.addAction(new Action()
  {
     public void run()
  {
     if (UIDefinition.getScreen("screen34") instanceof
BOBScreenUpdate_Activities_Custom){
     BOBScreenUpdate_Activities_Custom screen =
(BOBScreenUpdate_Activities_Custom)UIDefinition.getScreen("screen34
");
     UiApplication.getUiApplication().pushScreen(screen);
           }
             }
        });
    }
  break;
    default:
          super.configureObjectHandlersById(ID, object);
    }
 }
```

A widget event in which an onDraw event is selected for a control, and the corresponding event delegate is generated:

```
protected void configureObjectHandlersById(int ID, Object object) {
switch (ID) {
......
case SELECTBOX31:
    if (object instanceof SelectBox) {
        SelectBox localselectBox31 = (SelectBox) object;
        localselectBox31.setControlID(SELECTBOX31);
        localselectBox31.setCustomEventsDelegate(
                new BOBScreenUpdate_Activities_SelectBoxDelegate(),
                Literals.CUSTOM_EVENT_ON_DRAW);
    }

break;
     ......
```

Add custom code in the widget event delegate for an onDraw event. In this case we redraw the selectBox by adding custom code to the paint and drawFocus methods:

```
/**
 * (non-Javadoc)
 *
 * @see
com.sybase.uep.bobclient.controls.ICustomEventsDelegate#paint(Objec
t field, int controlID, Graphics g)
 */
 public void paint(Object field, int controlID, Graphics g) {
   // custom code
switch (controlID) {
case BOBScreenUpdate_Activities_.SELECTBOX31:
 g.clear();
 SelectBox selectBox = (SelectBox)field;
   int currentSelect = selectBox.getSelectedIndex();
   if(currentSelect>-1)
   {
        SelectBoxChoice choice =
(SelectBoxChoice)selectBox.getChoice(currentSelect);
        String choiceLabel = choice.getLabel();
        g.setColor(Color.BLACK);
        g.drawText( choiceLabel, H_PADDING,
(selectBox.getPreferredHeight()-
selectBox.getFontStyle().getFont().getHeight())/2,
DrawStyle.ELLIPSIS, selectBox.getPreferredWidth() -
DROPDOWN_HINT_AREA_WIDTH-H_PADDING);
   }
break;
        default:
            break;
        }
      }

/**
 * (non-Javadoc)
 *
 * @see
com.sybase.uep.bobclient.controls.ICustomEventsDelegate#drawFocus
 * (Object field, int controlID, Graphics g, boolean on)
 */
public void drawFocus(Object field, int controlID, Graphics g,
boolean on) {
// custom code
    switch (controlID) {
    case BOBScreenUpdate_Activities_.SELECTBOX31:
        g.clear();
        SelectBox selectBox = (SelectBox)field;
     int oldColor = g.getColor();
     int oldBgColor = g.getBackgroundColor();
     int currentSelect = selectBox.getSelectedIndex();
     if(currentSelect>-1)
     {
       SelectBoxChoice choice =
(SelectBoxChoice)selectBox.getChoice(currentSelect);
```

```
        String choiceLabel = choice.getLabel();
        int y = (selectBox.getPreferredHeight()-
selectBox.getFontStyle().getFont().
        getHeight())/2;
        int height = selectBox.getPreferredHeight();
        int width = selectBox.getPreferredWidth() -
DROPDOWN_HINT_AREA_WIDTH-H_PADDING;
        g.setColor(0x00FFFFFF);
        int actualWidth = g.drawText( choiceLabel, H_PADDING, y,
DrawStyle.ELLIPSIS, width);
        g.setColor(0x00185AB5);
        g.fillRect(H_PADDING, y, actualWidth, height);
        g.setColor(0x00FFFFFF);
        g.drawText( choiceLabel, H_PADDING, y, DrawStyle.ELLIPSIS,
width);
    }
    g.setColor(oldColor);
    g.setBackgroundColor(oldBgColor);
    break;
default:
    break;
    }
}
......
```

Reference

Index