Developer Guide: Android Object API Applications

Sybase Unwired Platform 2.1 ESD #2

Contents

Getting Started with Android Development

Use advanced Sybase® Unwired Platform features to create applications for Android devices. The audience is advanced developers who may be new to Sybase Unwired Platform.

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

Companion guides include:

- *Sybase Unwired WorkSpace – Mobile Business Object* Development
- *Troubleshooting for Sybase Unwired Platform*.
- *A complete Client Object API reference is available in the Unwired Platform installation directory* `<UnwiredPlatform_InstallDir>\MobileSDK\ObjectAPI\apidoc\android`*.*
- *Fundamentals* contains high-level mobile computing concepts, and a description of how Sybase Unwired Platform implements the concepts in your enterprise.

Object API Applications

Object API applications are customized, full-featured mobile applications that use mobile data model packages, either using mobile business objects (MBOs) or Data Orchestration Engine, to facilitate connection with a variety of enterprise systems and leverage synchronization to support offline capabilities.

The Object API application model enables developers to write custom code — C#, Java, or Objective-C, depending on the target device platform — to create device applications.

Development of Object API applications provides the most flexibility in terms of leveraging platform specific services, but each application must be provisioned individually after being compiled, even for minor changes or updates.

Development involves both server-side and client-side components. Unwired Server brokers data synchronization and transaction processing between the server and the client components.

- Server-side components address the interaction between the enterprise information system (EIS) data source and the data cache. EIS data subsets and business logic are encapsulated in artifacts, called mobile business object or DOE packages, that are deployed to Unwired Server.

- Client-side components are built into the mobile application and address the interaction between the data cache and the mobile device data store. This can include synchronizing data with the server, offline data access capabilities, and data change notification.

These applications:

- Allow users to connect to data from a variety of EIS systems, including SAP® systems.
- Build in more complex data handling and logic.
- Leverage data synchronization to optimize and balance device response time and need for real-time data.
- Ensure secure and reliable transport of data.

Best Uses for Object API Applications

Synchronization applications provide operation replay between the mobile device, the middleware, and the back-end system. Custom native applications are designed and built to suit specific business scenarios from the ground up, or start with a bespoke application and be adapted with a large degree of customization.

Cache Synchronization

Cache synchronization allows mapping mobile data to SAP Remote Function Calls (RFCs) using Java Connector (JCO) and to other non-SAP data sources such as databases and Web services. When Sybase Unwired Platform is used in a stand-alone manner for data synchronization (without Data Orchestation Engine), it utilizes an efficient bulk transfer and data insertion technology between the middleware cache and the device database.

In an Unwired Platform standalone deployment, the mobile application is designed such that the developer specifies how to load data from the back end into the cache and then filters and downloads cache data using device-supplied parameters. The mobile content model and the mapping to the back end are directly integrated.

This style of coupling between device and back-end queries implies that the back end must be able to respond to requests from the middleware based on user-supplied parameters and serve up mobile data appropriately. Normally, some mobile-specific adaptation is required within SAP Business Application Programming Interfaces (BAPI). Because of the direct nature of application parameter mapping and RBS protocol efficiencies, Sybase Unwired Platform cache synchronization deployment is ideal:

- With large payloads to devices (may be due to mostly disconnected scenarios)
- Where ad hoc data downloads might be expected
- For SAP® or non-SAP back ends

Large payloads, for example, can occur in task worker (service) applications that must access large product catalogs, or where service occurs in remote locations and workers might synchronize once a day. While Sybase Unwired Platform synchronization does benefit from middleware caching, direct coupling requires the back end to support an adaptation where mobile user data can be determined.

Client Runtime Architecture

The goal of synchronization is to keep views (that is, the state) of data consistent among multiple tiers. The assumption is that if data changes on one tier (for example, the enterprise system of record), all other tiers interested in that data (mobile devices, intermediate staging areas/caches and so on) are eventually synchronized to have the same data/state on that system.

The Unwired Server synchronizes data between the device and the back-end by maintaining records of device synchronization activity in its cache database along with any cached data that may have been retrieved from the back-end or pushed from the device. The Unwired Server employs several components in the synchronization chain.

Mobile Channel Interfaces

Mobile channel interfaces provide a conduit for transporting data to and from remote devices. Two main channel interfaces provide messaging and replication.

- The messaging channel serves as the abstraction to all device-side notifications (BlackBerry Enterprise Service, Apple Push Notification Service, and others) so that when changes to back-end data occur, devices can be notified of changes relevant for their application and configuration. This channel also enables data synchronization on iOS.

  The messaging channel sends these types of notifications:

  - Change notifications - when Unwired Server detects changes in the back-end EIS, Unwired Server can send a notification to the device. By default, sending change notifications is disabled, but you can enable sending change notifications per synchronization group.

    To capture change notifications, you can register an onSynchronize callback. The synchronization context in the callback has a status you can retrieve.
  - When synchronizing, operation replay records are sent to the Unwired Server and the messaging channel sends a notification of `replayFinished`. The application must call another synchronize method to retrieve the result.
- The synchronization channel sends data to keep the Unwired Server and client synchronized. The synchronization is bi-directional.

Mobile Middleware Services

Mobile middleware services (MMS) arbitrate and manage communications between device requests from the mobile channel interfaces in the form that is suitable for transformation to a common MBO service request and a canonical form of enterprise data supplied by the data services.

Data Services

Data services is the conduit to enterprise data and operations within the firewall or hosted in the cloud. Data services and mobile middleware services together manage the cache database (CDB) where data is cached as it is synchronized with client devices.

Once a mobile application model is designed, it can be deployed to the Unwired Server where it operates as part of a specialized container-managed package interfacing with the mobile middleware services and data services components. Cache data and messages persist in the databases in the data tier. Changes made on the device are passed to the mobile middleware services component as an operation replay and replayed against the data services interfaces with the back-end. Data that changes on the back- end as a result of device changes, or those originating elsewhere, are replicated to the device database.

Documentation Roadmap for Unwired Platform

Sybase® Unwired Platform documents are available for administrative and mobile development user roles. Some administrative documents are also used in the development and test environment; some documents are used by all users.

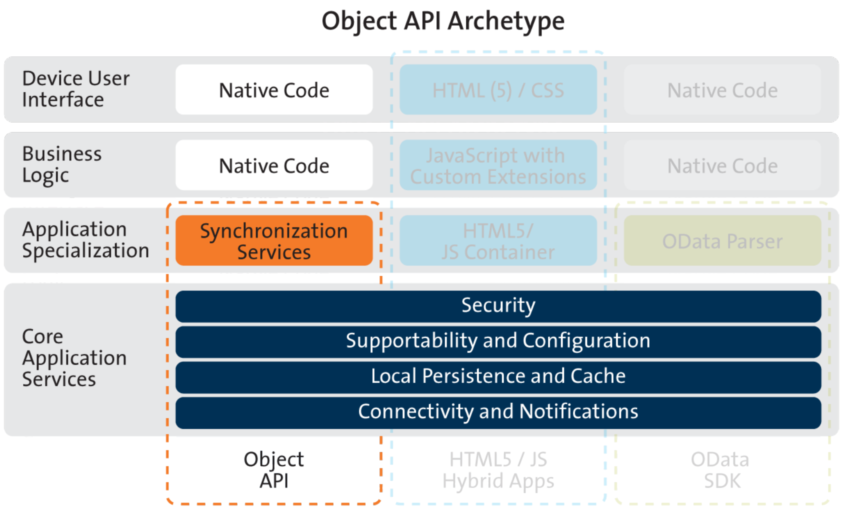
See *Documentation Roadmap* in *Fundamentals* for document descriptions by user role. *Fundamentals* is available on the Sybase Product Documentation Web site.

Check the Sybase Product Documentation Web site regularly for updates: access *http://sybooks.sybase.com/nav/summary.do?prod=1289*, then navigate to the most current version.

Development Task Flow for Object API Applications

Describes the overall development task flow for Object API applications, and provides information and procedures for setting up the development environment, and developing device applications.

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



The Object API provides the core application services described in the diagram.

The Authentication APIs provide security by authenticating the client to the Unwired Server.

The Synchronization APIs allow you to synchronize mobile business objects (MBOs) based on synchronization parameters, for individual MBOs, or as a group, based on the group's synchronization policy.

The Application and Connection APIs allow clients to register with and connect to the Unwired Server. The Callback Handler and Listener APIs, and the Target Change Notification APIs provide notifications to the client on operation success or failure, or changes in data.

With non-DOE-based applicaitons, Connectivity uses the MobiLink channel and Notifications use the Messaging channel.

1. *Installing the Android Development Environment*

Install the Android development environment, and prepare Android devices for authentication.

2. *Creating a Project*

   Build a device application project.
3. *Generating Java Object API Code*

   Generate object API code containing mobile business object (MBO) references, which allows you to use APIs to develop device applications for Android devices. You can generate code either in Sybase Unwired WorkSpace, or by using a command-line utility for generating code.
4. *Customizing the Application Using the Object API*

   Use the Object API to customize the application. An application consists of building blocks which the developer uses to start the application, perform functions needed for the application, and shutdown and uninstall the application.
5. *Testing Applications*

   Test native applications on a device or simulator.
6. *Localizing Applications*

   Localize an Android application by creating default and alternate resources.
7. *Packaging Applications*

   Package applications according to your security or application distribution requirements.

Installing the Android Development Environment

Install the Android development environment, and prepare Android devices for authentication.

1. *Installing the Android SDK*

   Install the Android SDK.
2. *Installing ADT in Unwired WorkSpace*

   You can install the supported version of Android Development Tools (ADT) directly in to the Sybase Unwired WorkSpace Eclipse environment.
3. *Installing X.509 Certificates on Android Devices and Emulators*

   Install the .p12 certificate on the Android device or emulator for authentication. A certificate provides an additional level of secure access to an application, and may be required by an organization's security policy.

See also

- *Creating a Project* on page 8

Installing the Android SDK

Install the Android SDK.

1. Confirm your system meets the requirements at *http://developer.android.com/sdk/requirements.html*.
2. Download and install the SDK starter package from *http://developer.android.com/sdk/index.html*.
3. Launch the **Android SDK Manager** and install the Android SDK tools, platform, and compatibility package for Android.
4. Launch the **Android Virtual Device Manager**, and create an Android virtual device to use as your simulator.

Installing ADT in Unwired WorkSpace

You can install the supported version of Android Development Tools (ADT) directly in to the Sybase Unwired WorkSpace Eclipse environment.

1. Download the ADT Plugin for Eclipse at *http://dl.google.com/android/ADT-16.0.1.zip*.
2. Start Eclipse, then select **Help > Install New Software**.
3. Click **Add**, in the top-right corner.
4. In the Add Repository dialog, click **Archive**.
5. Select the ADT Plugin for Eclipse zip file.
6. Enter a **Name** for the local update site, such as Android Plugin, then click **OK**.
7. In the Available Software dialog, select **Development Tools**, then click **Next**.
8. In the next window, a list of downloadable tools, click **Next**.
9. Accept the license agreements, then click **Finish**.

   Note: If you get a security warning about the authenticity or validity of the software, click **OK**.

10. When the installation completes, restart Unwired WorkSpace.

Installing X.509 Certificates on Android Devices and Emulators

Install the .p12 certificate on the Android device or emulator for authentication. A certificate provides an additional level of secure access to an application, and may be required by an organization's security policy.

Prerequisites

- Java SE Development Kit (JDK) must be installed.
- The Android SDK must be installed.

Task

1. Connect the Android device to your computer with the USB cable.
2. To install using Eclipse with the ADT plugin:

   Note: USB debugging must be enabled.

   a) Open the Windows File Explorer view. From the menu bar, navigate to **Window** > **Show View** > **Other**.
   b) In the Show View dialog, expand the Android folder and select **File Explorer**.
   c) Expand **mnt** > **sdcard** and select the **sdcard** folder.
   d) In the top right of the File Explorer view, click **Push a file onto the device**.
   e) In the Put File on Device dialog, select the certificate and click **Open**.
3. To install using Windows Explorer:

   Note: USB debugging must be disabled.

   a) Open **Windows Explorer**
   b) Under your computer, click the Android device to expand the folder.
   c) Click **Device Storage**, navigate to and select the certificate.
   d) Import the certificate to the Device Storage folder.
4. To install using the Android Debug Bridge (adb):

   Note: USB debugging must be enabled. You can enable USB debug mode from the device menu by selecting Settings > Application > USB Debugging.

   a) Open the command line directory to the `adb.exe` file, for example, `C:\Program Files\android-sdk-windows\tools`, or `C:\Program Files\android-sdk-windows\platform-tools`
   b) Run the command: `adb push %PathToCert%\MyCert.p12 /sdcard/MyCert.p12`

Creating a Project

Build a device application project.

1. *Creating a Project in Unwired WorkSpace*

   Create a project for your Android device application in Sybase Unwired WorkSpace.
2. *Importing Libraries and Code*

   Create a specific directory structure, within your Eclipse project, containing the library resources needed to compile your Android client code.

See also

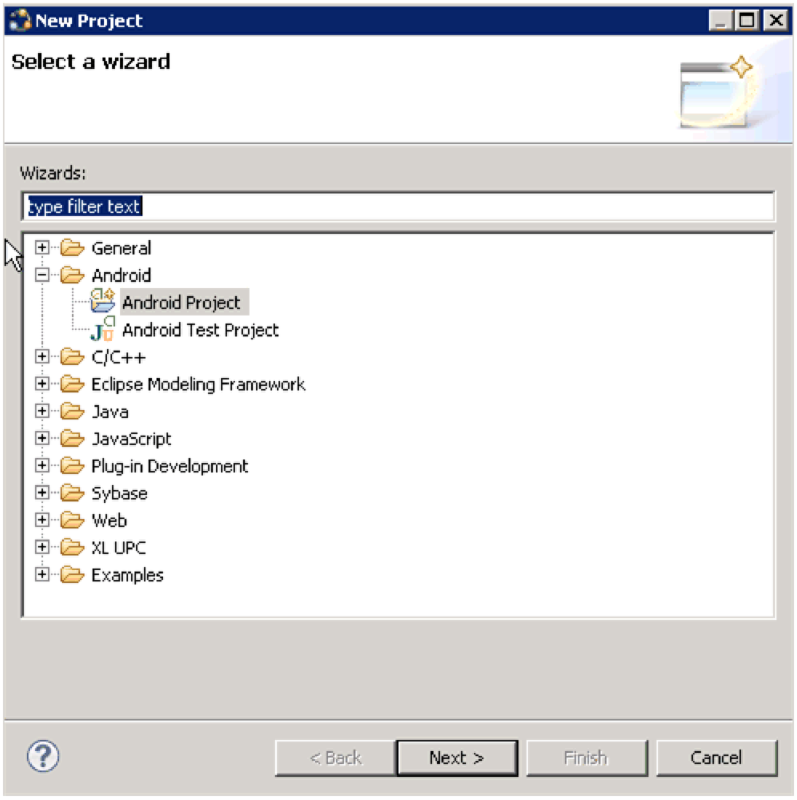
- *Installing the Android Development Environment* on page 6

- *Generating Java Object API Code* on page 11

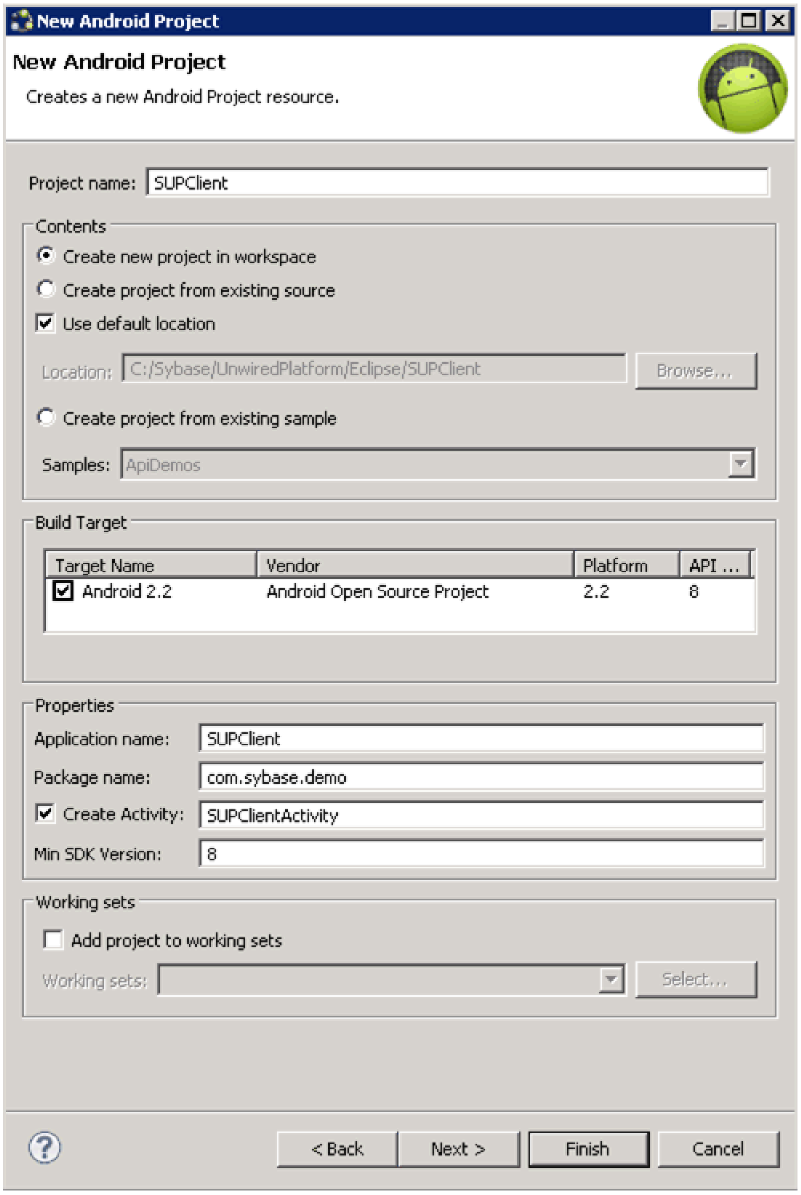
Creating a Project in Unwired WorkSpace

Create a project for your Android device application in Sybase Unwired WorkSpace.

1. In Sybase Unwired WorkSpace, select **File** > **New** > **Project**.
2. Select **Android** > **Android Project**.



3. In the **New Android Project** wizard, enter these values and click **Finish**:
   - **Project name: –** enter the name of the project
   - **Package name: –** enter the name of the package
   - **Min SDK Version: –** 8



Project name: SUPClient

Contents

- [✓ Create new project in workspace]
- Create project from existing source
- [✓] Use default location

Location: C:/Sybase/UnwiredPlatform/Eclipse/SUPClient

- Create project from existing sample

Samples: ApiDemos

Build Target

Target Name	Vendor	Platform	API ...
✓ Android 2.2	Android Open Source Project	2.2	8

Properties

Application name: SUPClient

Package name: com.sybase.demo

[✓] Create Activity: SUPClientActivity

Min SDK Version: 8

Working sets

- Add project to working sets

Working sets:

4. Add the following user permissions in `AndroidManifest.xml`:

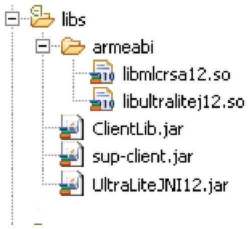
```
<uses-permission android:name="android.permission.INTERNET"></
uses-permission>
<uses-permission
android:name="android.permission.READ_PHONE_STATE"></uses-
permission>
<uses-permission
android:name="android.permission.ACCESS_WIFI_STATE"></uses-
```

```
permission>
<uses-permission
android:name="android.permission.ACCESS_NETWORK_STATE"></uses-
permission>
<uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE"></uses-
permission>
```

Importing Libraries and Code

Create a specific directory structure, within your Eclipse project, containing the library resources needed to compile your Android client code.

1. In your Sybase Unwired WorkSpace project, create a `libs` directory.
2. Copy the following library and JAR files from `<UnwiredPlatform_InstallDir>\UnwiredPlatform\MobileSDK\ObjectAPI\Android` into the `libs` directory within your project, using the exact directory structure shown here:



```
libs
  armeabi
    libmlcrsa12.so
    libultralitej12.so
  ClientLib.jar
  sup-client.jar
  UltraLiteJNI12.jar
```

3. Select **Project** > **Properties** > **Java Build Path**. On the **Libraries** tab, add the libraries to the project.

Generating Java Object API Code

Generate object API code containing mobile business object (MBO) references, which allows you to use APIs to develop device applications for Android devices. You can generate code either in Sybase Unwired WorkSpace, or by using a command-line utility for generating code.

See also

- *Creating a Project* on page 8
- *Customizing the Application Using the Object API* on page 25

Generating Java Object API Code Using Sybase Unwired WorkSpace

Use Sybase Unwired WorkSpace to generate object API code containing mobile business object (MBO) references.

Prerequisites

Develop the MBOs that will be referenced in the device applications you are developing. A mobile application project must contain at least one non-online MBO. You must have an active connection to the datasources to which the MBOs are bound.

Task

Unwired Platform provides the Code Generation wizard for generating object API code. Code generation creates the business logic, attributes, and operations for your mobile business object.

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

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

2. (Optional; this page of the code generation wizard is seen only if you are using the Advanced developer profile) Enter the information for these options, then click **Next**:

Option	Description
Select code generation configuration	Select one of: • Continue without a configuration – generate device code without using a configuration. • Select an existing configuration – either select an existing configuration from which you generate device client code, or create a new configuration. By default, a configuration named **Most recent configuration** is available.Selecting this option enables: • Select code generation configuration – lists any existing configurations, from which you can select and use for this session. You can also delete any existing saved configurations. • Create new configuration by clicking the **Add** button. In the dialog, enter the **Name** of the new configuration and provide a description, and click **Create** to save the configuration for future sessions. Select an existing configuration as a starting point for this session and click **Clone** to modify the configuration.

3. On the Select Mobile Objects page, select all the MBOs in the mobile application project or select MBOs under a specific synchronization group, for which references, metadata, and dependencies (referenced MBOs) are included in the generated device code. Then click **Next**.

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

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

4. Enter the information for these configuration options:

Option	Description
Language	Select **Java**.
Platform	Select the platform (target device) for which the device client code is intended. • Android
Unwired Server	Specify a default Unwired Server connection profile to which the generated code connects at runtime.
Server domain	Choose the domain to which the generated code connects. By default, if you specified an Unwired Server to which you previously connected successfully, the first domain in the list is chosen. Accept this domain, or enter a different one. **Note:** This field is enabled only when an Unwired Server is selected.

Option	Description
Page size	(Optional) Select the page size for the generated client code. If you do not set the page size, the default is at runtime, which is proposed based on the selected MBO's attributes. The page size should be larger than the sum of all attribute lengths (a binary length greater than 32767 is converted to a binary large object (BLOB), and is not included in the sum; a string greater than 8191 is converted to a character large object (CLOB), and is also not included) for any MBO that is included. If an MBO attribute's length sum is greater than the page size, some attributes automatically convert to BLOB or CLOB, and therefore cannot be put into a `where` clause. **Note:** This field is only enabled when an Unwired Server is selected.
Package	Enter a unique name for the Java package. The default package is the project name.
Destination	Specify the destination of the generated device client files. Enter (or **Browse**) to a **Project path** or **File system path** (Mobile Application project) location, and select `Generated Code` as the destination. JAR files are automatically added to the destination project. Select **Clean up destination before code generation** to clean up the destination folder before generating the device client files.
Third-party jar file	If you select Java as the language, enter, or browse to the `android.jar` file (for Android), which adds it to the project build path, and prevents errors after code generation.

5. The check box for "Generate metadata classes" is automatically selected as read only for Android, and the "Including object manager classes" checkbox is de-selected.
6. (Optional) Select **Generate JavaDoc** to generate API documentation from the source code.

   Note: For Android, this is selected by default.
7. Click **Finish**.
8. Examine the generated code location and contents.

9. Validate the generated code.

Generating Java Object API Code Using the Code Generation Utility

Use the Code Generation Utility to generate object API code containing mobile business object (MBO) references. This method of generating code allows you to automate the process of code generation, for example through the use of scripts.

Prerequisites

- Use Unwired WorkSpace to develop and package your mobile business objects. See *Sybase Unwired WorkSpace - Mobile Business Object Development* > *Develop* > *Developing a Mobile Business Object*.
- Deploy the package to Unwired Server, creating files required for code generation from the command line. See *Sybase Unwired WorkSpace - Mobile Business Object Development* > *Develop* > *Packaging and Deploying Mobile Business Objects* >*Automated Deployment of Unwired WorkSpace Projects*

Task

1. Locate `<domain name>_package.jar` in your mobile project folder. For the SUP101 example, the project is deployed to the default domain, and the deploy jar file is in the following location: `SUP101\Deployment\.pkg.profile \My_Unwired_server\default_package.jar`.
2. Make sure that the JAR file contains this file:
   - `deployment_unit.xml`
3. Use a utility to extract the `deployment_unit.xml` file to another location.
4. From <*UnwiredPlatform_InstallDir*>`\MobileSDK\ObjectAPI\Utils \bin`, run the `codegen.bat` utility, specifying the following parameters:

   ```
   codegen.bat -java -client -android -ulj deployment_unit.xml [-
   output <output_dir>] [-doc]
   ```

   - The `-output` parameter allows you to specify an output directory. If you omit this parameter, the output goes into the <*UnwiredPlatform_InstallDir*> `\MobileSDK\ObjectAPI\Utils\genfiles` directory, assuming codegen.bat is run from the <*UnwiredPlatform_InstallDir*>`\MobileSDK \ObjectAPI\Utils\genfiles` directory.
   - The `-doc` parameter specifies that documentation is generated for the generated code.

   Ignore these warnings:

   ```
   log4j:WARN No appenders could be found for logger ...
   log4j:WARN Please initialize the log4j system properly.
   ```

Generated Code Location and Contents

If you generated code in Unwired WorkSpace, generated object API code is stored by default in the "Destination" location you specified during code generation. If you generated code with the Code Generation Utility, generateed object API code is stored in the `<UnwiredPlatform_InstallDir>\UnwiredPlatform\MobileSDK \ObjectAPI\Utils\genfiles` folder after you you generate code .

The contents of the folder is determined by the options you selected in the Generate Code wizard in Unwired WorkSpace, or specified in the Code Generation Utility. The contents include generated class files that contain:

- MBO – class which handles persistence and operation replay of your MBOs.
- Synchronization parameters – any synchronization parameters for the MBOs.
- Personalization parameters – personalization parameters used by the package.
- Metadata – Metadata class that allow you to query meta data including MBOs, their attributes, and operations, in a persistent table at runtime..

Validating Generated Code

Validation rules are enforced when generating client code. Define prefix names in the Mobile Business Object Preferences page of the Code Generation wizard to correct validation errors.

Sybase Unwired WorkSpace validates and enforces identifier rules and checks for keyword conflicts in generated code, for example, by displaying error messages in the Properties view or in the wizard. Other than the known name conversion rules (converting '.' to '_', removing white space from names, and so on), there is no other language-specific name conversion. For example, cust_id is not changed to custId.

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

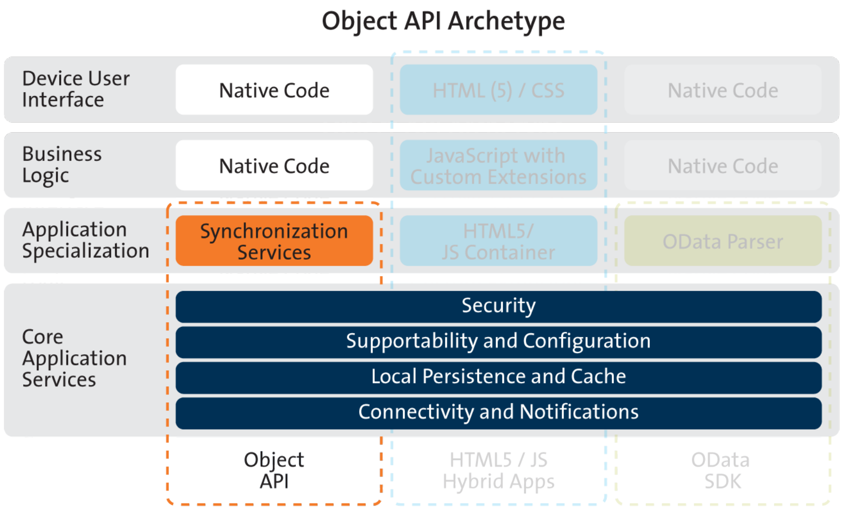
1. Select **Window > Preferences**.
2. Expand **Sybase, Inc > Mobile Development**.
3. Select **Mobile Business Object**.
4. Add or modify the **Naming Prefix** settings as needed.

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

Development Task Flow for DOE-based Object API Applications

Describes the overall development task flow for DOE-based native applications, and provides information and procedures for setting up the development environment, and developing DOE-based device applications.

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



The Object API provides the core application services described in the diagram.

The Authentication APIs provide security by authenticating the client to the Unwired Server.

The Synchronization APIs allow you to synchronize mobile business objects (MBOs) based on synchronization parameters, for individual MBOs, or as a group, based on the group's synchronization policy.

The Application and Connection APIs allow clients to register with and connect to the Unwired Server. The Callback Handler and Listener APIs, and the Target Change Notification APIs provide notifications to the client on operation success or failure, or changes in data.

With non-DOE-based applicaitons, Connectivity uses the MobiLink channel and Notifications use the Messaging channel.

1. *Installing the Android Development Environment*

Install the Android development environment, and prepare Android devices for authentication.

2. *Creating a Project*

   Build a device application project.
3. *Generating Java Object API Code*

   Use the Code Generation Utility to generate object API code, which allows you to use APIs to develop device applications for Android devices.
4. *Customizing the Application Using the Object API*

   Use the Object API to customize the application. An application consists of building blocks which the developer uses to start the application, perform functions needed for the application, and shutdown and uninstall the application.
5. *Testing Applications*

   Test native applications on a device or simulator.
6. *Localizing Applications*

   Localize an Android application by creating default and alternate resources.
7. *Packaging Applications*

   Package applications according to your security or application distribution requirements.

Installing the Android Development Environment

Install the Android development environment, and prepare Android devices for authentication.

1. *Installing the Android SDK*

   Install the Android SDK.
2. *Installing X.509 Certificates on Android Devices and Emulators*

   Install the .p12 certificate on the Android device or emulator for authentication. A certificate provides an additional level of secure access to an application, and may be required by an organization's security policy.

See also

- *Creating a Project* on page 20

Installing the Android SDK

Install the Android SDK.

1. Confirm your system meets the requirements at *http://developer.android.com/sdk/requirements.html*.

2. Download and install the SDK starter package from *http://developer.android.com/sdk/index.html*.
3. Launch the **Android SDK Manager** and install the Android SDK tools, platform, and compatibility package for Android.
4. Launch the **Android Virtual Device Manager**, and create an Android virtual device to use as your simulator.

Installing X.509 Certificates on Android Devices and Emulators

Install the .p12 certificate on the Android device or emulator for authentication. A certificate provides an additional level of secure access to an application, and may be required by an organization's security policy.

Prerequisites

- Java SE Development Kit (JDK) must be installed.
- The Android SDK must be installed.

Task

1. Connect the Android device to your computer with the USB cable.
2. To install using Eclipse with the ADT plugin:

   Note: USB debugging must be enabled.

   a) Open the Windows File Explorer view. From the menu bar, navigate to **Window** > **Show View** > **Other**.
   b) In the Show View dialog, expand the Android folder and select **File Explorer**.
   c) Expand **mnt > sdcard** and select the **sdcard** folder.
   d) In the top right of the File Explorer view, click **Push a file onto the device**.
   e) In the Put File on Device dialog, select the certificate and click **Open**.
3. To install using Windows Explorer:

   Note: USB debugging must be disabled.

   a) Open **Windows Explorer**
   b) Under your computer, click the Android device to expand the folder.
   c) Click **Device Storage**, navigate to and select the certificate.
   d) Import the certificate to the Device Storage folder.
4. To install using the Android Debug Bridge (adb):

   Note: USB debugging must be enabled. You can enable USB debug mode from the device menu by selecting Settings > Application > USB Debugging.

a) Open the command line directory to the `adb.exe` file, for example, `C:\Program Files\android-sdk-windows\tools`, or `C:\Program Files\android-sdk-windows\platform-tools`
b) Run the command: `adb push %PathToCert%\MyCert.p12 /sdcard/MyCert.p12`

Creating a Project

Build a device application project.

1. *Creating a Project in Unwired WorkSpace*

   Create a project for your Android device application in Sybase Unwired WorkSpace.
2. *Importing Libraries and Code*

   Create a specific directory structure, within your Eclipse project, containing the library resources needed to compile your Android client code.

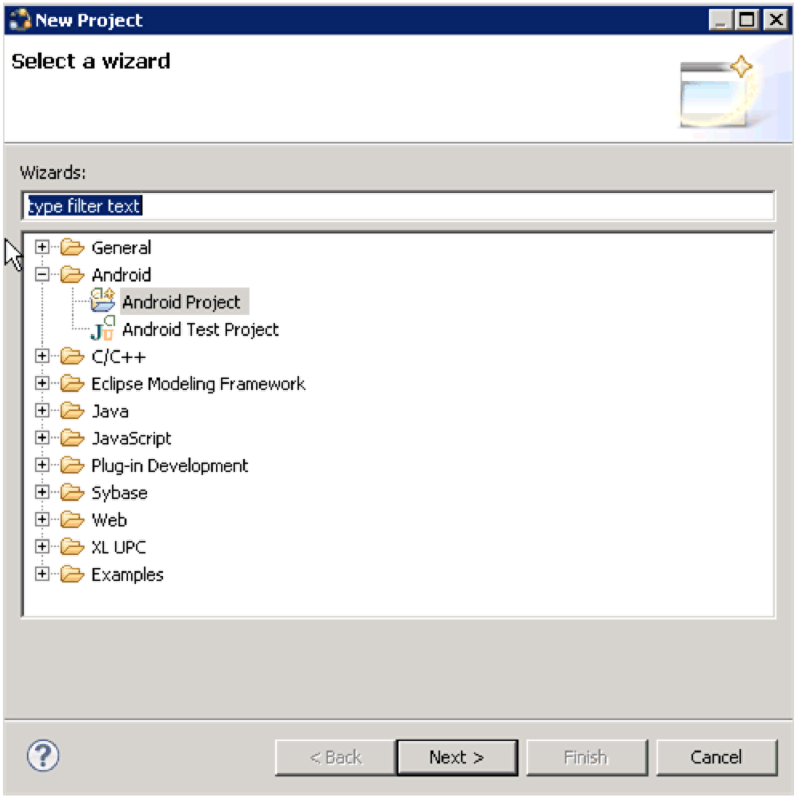
See also

- *Installing the Android Development Environment* on page 18
- *Generating Java Object API Code* on page 23

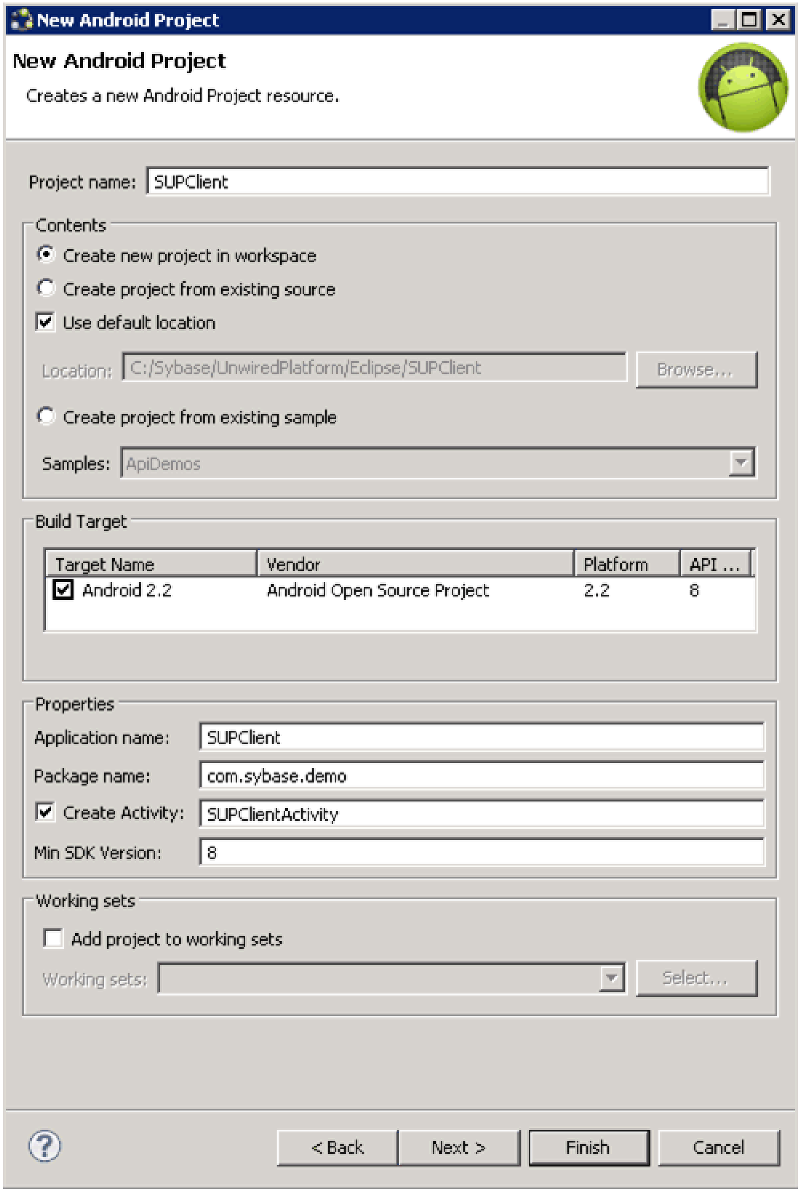
Creating a Project in Unwired WorkSpace

Create a project for your Android device application in Sybase Unwired WorkSpace.

1. In Sybase Unwired WorkSpace, select **File** > **New** > **Project**.
2. Select **Android** > **Android Project**.



3. In the **New Android Project** wizard, enter these values and click **Finish**:
   - **Project name: –** enter the name of the project
   - **Package name: –** enter the name of the package
   - **Min SDK Version: –** 8



New Android Project

Creates a new Android Project resource.

Project name: SUPClient

Contents

(●) Create new project in workspace

( ) Create project from existing source

[✓] Use default location

Location: C:/Sybase/UnwiredPlatform/Eclipse/SUPClient

( ) Create project from existing sample

Samples: ApiDemos

Build Target

Target Name	Vendor	Platform	API ...
[✓] Android 2.2	Android Open Source Project	2.2	8

Properties

Application name:	SUPClient
Package name:	com.sybase.demo
[✓] Create Activity:	SUPClientActivity
Min SDK Version:	8

Working sets

[ ] Add project to working sets

Working sets:

4. Add the following user permissions in `AndroidManifest.xml`:

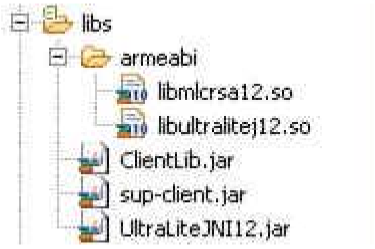
```
<uses-permission android:name="android.permission.INTERNET"></
uses-permission>
<uses-permission
android:name="android.permission.READ_PHONE_STATE"></uses-
permission>
<uses-permission
android:name="android.permission.ACCESS_WIFI_STATE"></uses-
```

```
permission>
<uses-permission
android:name="android.permission.ACCESS_NETWORK_STATE"></uses-
permission>
<uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE"></uses-
permission>
```

Importing Libraries and Code

Create a specific directory structure, within your Eclipse project, containing the library resources needed to compile your Android client code.

1. In your Sybase Unwired WorkSpace project, create a `libs` directory.
2. Copy the following library and JAR files from `<UnwiredPlatform_InstallDir>\UnwiredPlatform\MobileSDK\ObjectAPI\Android` into the `libs` directory within your project, using the exact directory structure shown here:



```
libs
  armeabi
    libmlcrsa12.so
    libultralitej12.so
  ClientLib.jar
  sup-client.jar
  UltraLiteJNI12.jar
```

3. Select **Project** > **Properties** > **Java Build Path**. On the **Libraries** tab, add the libraries to the project.

Generating Java Object API Code

Use the Code Generation Utility to generate object API code, which allows you to use APIs to develop device applications for Android devices.

Prerequisites

- Generate and download the ESDMA bundle for you application.
- Run the ESDMA Converter utility to turn your ESDMA into an Unwired Platform package.
- Deploy the package to Unwired Server.

Task

1. Locate `<domain name>_package.jar` in your mobile project folder. For the SUP101 example, the project is deployed to the default domain, and the deploy jar file is in

the following location: `SUP101\Deployment\.pkg.profile\My_Unwired_server\default_package.jar`.

2. Make sure that the JAR file contains this file:
   - `deployment_unit.xml`
3. From `<UnwiredPlatform_InstallDir>\UnwiredPlatform\MobileSDK\ObjectAPI\Utils\bin`, run the `codegen.bat` utility, specifying the following parameters:

   ```
   codegen -android -client -doe -java -ulj
   [-output <output_dir>] [-doc] <ESDMA_dir>\META-INF\sup-db.xml
   ```

   - The `-output` parameter allows you to specify an output directory. If you omit this parameter, the output goes into the `<UnwiredPlatform_InstallDir>\UnwiredPlatform\MobileSDK\ObjectAPI\Utils\genfiles` directory, assuming codegen.bat is run from the `<UnwiredPlatform_InstallDir>\UnwiredPlatform\MobileSDK\ObjectAPI\Utils\bin` directory.
   - The `-doc` parameter specifies that documentation is generated for the generated code.

   Ignore these warnings:

   ```
   log4j:WARN No appenders could be found for logger ...
   log4j:WARN Please initialize the log4j system properly.
   ```

See also

- *Creating a Project* on page 20
- *Customizing the Application Using the Object API* on page 25

Generated Code Location and Contents

The location of the generated Object API code is the location you specified when you generated the code using codegen.bat at the command line.

The contents of the folder is determined by the parameters you pass to `codegen.bat` in the command line, and include generated class files that contain:

- DatabaseClass – package level class that handles subscription, login, synchronization, and other operations for the package.
- MBO – class which handles persistence and operation replay of your MBOs.
- Personalization parameters – personalization parameters used by the package.
- Metadata – Metadata class that allows you to query meta data including MBOs, their attributes, and operations, in a persistent table at runtime..

Customizing the Application Using the Object API

Use the Object API to customize the application. An application consists of building blocks which the developer uses to start the application, perform functions needed for the application, and shutdown and uninstall the application.

See also

- *Generating Java Object API Code* on page 11
- *Testing Applications* on page 49
- *Generating Java Object API Code* on page 23

Initializing an Application

Initialize the application when it starts the first time and subsequently.

- *Initially Starting an Application*

  Starting an application the first time.
- *Subsequently Starting an Application*

  Subsequent start-ups are different from the first start-up.

Initially Starting an Application

Starting an application the first time.

1. *Setting Up Application Properties*

   The Application instance contains the information and authentication credentials needed to register and connect to the Sybase Unwired Platform server.
2. *Registering an Application*

   Each device must register with the server before establishing a connection.
3. *Setting Up the Connection Profile*

   The Connection Profile stores information detailing where and how the local database is stored, including location and page size. The connection profile also contains UltraLiteJ runtime tuning values.
4. *Setting Up Connectivity*

   Store connection information to the Sybase Unwired Server data synchronization channel.
5. *Creating and Deleting a Device's Local Database*

   There are methods in the generated package database class that allow programmers to delete or create a device's local database. A device local database is automatically created when needed by the Object API. The application can also create the database programatically by calling the createDatabase method. The device's local database should be deleted when uninstalling the application.

6. *Logging In*

   Use online authentication with the server, and offline authentication with the device.
7. *Turn Off API Logger*

   In production environments, turn off the API logger to improve performance.
8. *Setting Up Callbacks*

   When your application starts, it can register database and MBO callback listeners, as well as synchronization listeners.
9. *Connecting to the Device Database*

   Establish a connection to the database on the device.
10. *Synchronizing Applications*

    Synchronize package data between the device and the server.
11. *Specifying Personalization Parameters*

    Use personalization parameters to provide default values used with synchronization, connections with back-end systems, MBO attributes, or EIS arguments. The PersonalizationParameters class is within the generated code for your project.
12. *Specifying Synchronization Parameters*

    Use synchronization parameters within the mobile application to download filtered MBO data.

See also

- *Application APIs* on page 57
- *Connection APIs* on page 67

Setting Up Application Properties

The Application instance contains the information and authentication credentials needed to register and connect to the Sybase Unwired Platform server.

The following code illustrates how to set up the minimum required fields:

```
// Initialize Application settings
Application app = Application.getInstance();

// The identifier has to match the application ID deployed to the SUP
server
app.setApplicationIdentifier("SUP101");
// Set the android.content.Context for the application
app.setApplicationContext(context); // context is the
android.content.Context

// ConnectionProperties has the infomation needed to register
// and connect to SUP server
ConnectionProperties connProps = app.getConnectionProperties();
```

```
connProps.setServerName("supserver.mycompany.com");
connProps.setPortNumber(5001);
// Other connection properties need to be set when connecting through
relay server

// provide user credentials
LoginCredentials loginCred = new LoginCredentials("supAdmin",
"supPwd");
connProps.setLoginCredentials(loginCred);

// Initialize generated package database class with this Application
instance
SUP101DB.setApplication(app);
```

Note: `setApplicationIdentifier` and `setApplicationContext` must be called in the user interface thread.

See also

- *Application APIs* on page 57

Registering an Application

Each device must register with the server before establishing a connection.

To register the device with the server during the initial application startup, use the `registerApplication` method in the `com.sybase.mobile.Application` class. You do not need to use the `registerApplication` method for subsequent application startups.To start the connection to complete the registration process, use the `Application.startConnection` method.

Call the generated database's `setApplication` method before starting the connection or registering the device.

The following code shows how to register the application and device.

```
// Initialize Application settings
Application app = Application.getInstance();

// The identifier has to match the
// application ID deployed to the SUP server
app.setApplicationIdentifier("SUP101");
ApplicationCallback appCallback = new MyApplicationCallback(); //
MyApplicationCallback implements ApplicationCallback
app.setApplicationCallback(appCallback); // optional
app.setApplicationContext(myAndroidContext); // required
              // use the android.content.Context for the application

// set connection properties, login credentials,  etc
...

SUP101DB.setApplication(app);

if (app.getRegistrationStatus() != RegistrationStatus.REGISTERED)
{
   // If the application has not been registered to the server,
   // register now
   app.registerApplication(<timeout_value>);
}
else
{
   // start the connection to server
   app.startConnection(<timeout_value>);
}
```

See also

- *Application APIs* on page 57

Setting Up the Connection Profile

The Connection Profile stores information detailing where and how the local database is stored, including location and page size. The connection profile also contains UltraLiteJ runtime tuning values.

Set up the connection profile before the first database access, and check if the database exists by calling the `databaseExists` method in the generated package database class. Any settings you establish after the connection has already been established will not go into effect.

The generated database class automatically contains all the default settings for the connection profile. You may add other settings if necessary. For example, you can set the database to be stored in an SD card or set the encryption key of the database.

Use the `com.sybase.persistence.ConnectionProfile` class to set up the locally generated database:

1. Retrieve the connection profile object using the Sybase Unwired Platform database's `getConnectionProfile` method.
2. Use the connection profile object's `save` method to set the values once when the application first starts. On subsequent usage of the application, the connection profile will contain all the settings from the last `save` call.

```
// Initialize the device database connection profile (if needed)
ConnectionProfile connProfile = SUP101DB.getConnectionProfile();

// Store the database in an SD card
connProfile.setProperty("databaseFile",
android.os.Environment.getExternalStorageDirectory().getPath() + "/
SUP1011_0.ulj");

// encrypt the database
connProfile.setEncryptionKey("your encryption key"); //Encryption
key can be of arbitary length, but generally the longer, the better.

// use 100K for cache size
connProfile.setCacheSize(102400);
```

```
// save it
connProfile.save();
```

You can also automatically generate a encryption key and store it inside a data vault.

See also

- *ConnectionProfile* on page 67

Setting Up Connectivity

Store connection information to the Sybase Unwired Server data synchronization channel.

See also

- *Creating and Deleting a Device's Local Database* on page 30

Setting Up the Synchronization Profile

You can set Unwired Server synchronization channel information by calling the synchronization profile's setter method. By default, this information includes the server host, port, domain name, certificate and public key that are pushed by the message channel during the registration process.

Settings are automatically provisioned from the Unwired Server. The values of the settings are inherited from the application connection template used for the registration of the application connection (automatic or manual). You must make use of the connection and security settings that are automatically used by the Object API.

Typically, the application uses the settings as sent from the Unwired Server to connect to the Unwired Server for synchronization so that the administrator can set those at the application deployment time based on their deployment topology (for example, using relay server, using e2ee security, or a certificate used for the intermediary, such as a Relay Server Web server). See the *Applications* and *Application Connection Templates* topics in *System Administration*.

When the client registers and starts the application, the certificate is downloaded to the client, so that the client can be assigned the trusted certificate.

Set up a secured connection using the `ConnectionProfile` object.

1. Retrieve the synchronization profile object using the Sybase Unwired Platform database's `getSynchronizationionProfile` method.

   ```
   ConnectionProfile cp = SUP101DB.getSynchronizationProfile();
   ```

2. Set the connection fields in the `ConnectionProfile` object.

   ```
   cp.setServerName("SUP_Host");
   cp.setPortNumber(2481);
   cp.getStreamParams().setTrusted_Certificates(appname
   +"_trustedCertificates.crt");
   cp.setNetworkProtocol("https");
   ```

See also

- *Synchronization Profile* on page 70

Creating and Deleting a Device's Local Database

There are methods in the generated package database class that allow programmers to delete or create a device's local database. A device local database is automatically created when needed by the Object API. The application can also create the database programatically by calling the `createDatabase` method. The device's local database should be deleted when uninstalling the application.

Check if the locally generated database exists, create the database, or delete the database:

1. Check if an instance of the generated database exists by calling the generated database instance's `databaseExists` method.
2. If an instance of a the generated database does not exist, call the generated database instance's `createDatabase` method.

   ```
   if (!SUP101DB.databaseExists())
       {
           SUP101DB.createDatabase();
       }
   ```
3. Connect to the generated database by calling the generated database instance's `openConnection` method.

   ```
   SUP101DB.openConnection();
   ```

   If the database does not already exist, the `openConnection` method creates it.
4. When the local database is no longer needed, delete it by calling the generated database instance's `deleteDatabase` method.

   ```
   SUP101DB.deleteDatabase();
   ```

See also

- *Setting Up Connectivity* on page 29

Logging In

Use online authentication with the server, and offline authentication with the device.

1. Normally, the user is authenticated through the `registerApplication` and `startConnection` methods in the `Application` class. Once this is done there is no need to authenticate again. However, the user can authenticate directly with the server at any time during the application's execution by calling the generated database instance's `onlineLogin` method.
2. Authenticate using the last successful credentials on the device by calling the generated database instance's `offlineLogin` method.

Turn Off API Logger

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

```
SUP101DB.getLogger().setLogLevel(LogLevel.OFF);
```

Setting Up Callbacks

When your application starts, it can register database and MBO callback listeners, as well as synchronization listeners.

Callback handler and listener interfaces are provided so your application can monitor changes and notifications from Sybase Unwired Platform:

- The `com.sybase.mobile.ApplicationCallback` class is used for monitoring changes to application settings, messaging connection status, and application registration status.
- The `com.sybase.persistence.CallbackHandler` interface is used to monitor notifications and changes related to the database. Register callback handlers at the package level use the `registerCallbackHandler` method in the generated database class. To register for a particular MBO, use the `registerCallbackHandler` method in the generated MBO class.
- The `com.sybase.persistence.SyncStatusListener` class is used for debugging and performance measures when monitoring stages of a synchronization session, and can be used in the user interface to indicate synchronization progress.

See also

- *Connecting to the Device Database* on page 36
- *Callback and Listener APIs* on page 101

Setting Up Callback Handlers

Use the callback handlers for event notifications.

Use the `com.sybase.persistence.CallbackHandler` API for event notifications including login for synchronization and replay. If you do not register your own implementation of the `com.sybase.persistence.CallbackHandler` interface, the generated code will regsiter a new default callback handler.

1. The generated database class contains a method called `registerCallbackHandler`. Use this method to install your implementation of `CallbackHandler`.
   For example:
   ```
   SUP101DB.registerCallbackHandler(new MyCallbackHandler());
   ```
2. Each generated MBO class also has the same method to register your implementation of the `CallbackHandler` for that particular type. For example, if `Customer` is a generated MBO class, you can use the following code:

```
Customer.registerCallbackHandler(new
MyCustomerMBOCallbackHandler());
```

Create a Custom Callback Handler

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

```
import com.sybase.persistence.DefaultCallbackHandler;
……
public class Test
{
   public static void main(String[] args)
   {
      SUP101DB.registerCallbackHandler(new MyCallbackHandler());
      GenericList<SynchronizationGroup> sgs = new
GenericList<SynchronizationGroup>();
      sgs.add(SUP101DB.getSynchronizationGroup("sg1"));
      sgs.add(SUP101DB.getSynchronizationGroup("sg2"));
      SUP101DB.beginSynchronize(sgs, "my test synchronization
context");
   }
}

class MyCallbackHandler extends DefaultCallbackHandler
{
   public int onSynchronize(GenericList<SynchronizationGroup>
groups, SynchronizationContext context)
   {
      if ( context == null )
      {
         return SynchronizationAction.CANCEL;
      }

      if ("my test synchronization context".equals((String)
(context.getUserContext())))
      {
         return super.onSynchronize(groups, context);
      }

      switch (context.getStatus())
      {
         case SynchronizationStatus.STARTING:
            if (waitForMoreChanges())
            {
               return SynchronizationAction.CANCEL;
            }
            else
            {
               return SynchronizationAction.CONTINUE;
            }
         default:
            return SynchronizationAction.CONTINUE;
      }
   }
}
```

For DOE-based applications, replace `onSynchronize` in the example below with , `onReplaySuccess`, or one of the other callback handlers that is designed specifically to for DOE-based applications. See the *CallbackHandler API* topic in the *Client Object API Usage* section of this document.

Asynchronous Operation Replay

Upload operation replay records asynchronously.

When an application calls `submitPending` on an MBO on which a create, update, or delete operation is performed, an operation replay record is created on the device local database.

When `synchronize` is called, the operation replay records are uploaded to the server. The method returns without waiting for the backend to replay those records. The `synchronize` method downloads all the latest data changes and the results of the previously uploaded operation replay records that the backend has finished replaying. If you choose to disable asynchronous operation replay, each `synchronize` call will wait for the backend to finish replaying all the current uploaded operation replay records.

For DOE-based application, the operation replay record is sent to server via message channel when **submitPending** is called. When operation replay is done on server, one of the callback methods **onReplaySuccess** and **onReplayFailure** will be invoked depends on the result of the replay.

This feature is enabled by default. You can enable or disable the feature by setting the `asyncReplay` property in the synchronization profile. The following code shows how to disable asynchronous replay:

```
SUP101DB.getSynchronizationProfile().setAsyncReplay(false);
```

When asynchronous replay is enabled and the replay is finished, the onSynchronize callback method is invoked with a SynchronizationStatus value of `SynchronizationStatus.ASYNC_REPLAY_COMPLETED`. Use this callback method to invoke a synchronize call to pull in the results, as shown in the following callback handler.

```
public class MyCallbackHandler extends DefaultCallbackHandler
{
  public int onSynchronize(GenericList<SynchronizationGroup> groups,
SynchronizationContext context)
   {
     switch(context.getStatus())
     {
       case SynchronizationStatus.ASYNC_REPLAY_UPLOADED:
         LogMessage("AsyncReplay uploaded");
         break;
       case SynchronizationStatus.ASYNC_REPLAY_COMPLETED:
         // operation replay finished, return
SynchronizationAction.CONTINUE
         // will start a background synchronization to pull in the
results.
         LogMessage("AsyncReplay Done");
```

```
        break;
      default:
        break;
    }

    return SynchronizationAction.CONTINUE;
  }
}
```

Synchronize Status Listener for NON-DOE-based Applications

Retrieve the synchronization status.

Synchronize Status Listener is mainly for debugging and performance measuring purposes to monitor stages of a synchronize session. It could also be used in UI for synchronization progress status. Below is a sample Synchronize Status Listener.

```
import com.sybase.persistence.ObjectSyncStatusData;
import com.sybase.persistence.SyncStatusListener;
import com.sybase.persistence.SyncStatusState;

public class MySyncStatusListener implements SyncStatusListener
{
  long start;

  public MySyncStatusListener()
  {
    start = System.currentTimeMillis();
  }

  public boolean objectSyncStatus(ObjectSyncStatusData statusData)
  {
    long now = System.currentTimeMillis();
    long interval = now - start;
    start = now;
    String infoMessage;

    int syncState = statusData.getSyncStatusState();

    switch (syncState)
    {
      case SyncStatusState.SYNC_STARTING:
        infoMessage = "START [" + interval + "]";
        break;
      case SyncStatusState.APPLICATION_SYNC_SENDING_HEADER:
        infoMessage = "SENDING HEADERS [" + interval + "]";
        break;
      case SyncStatusState.APPLICATION_SYNC_SENDING_SCHEMA:
        infoMessage = "SENDING SCHEMA [" + interval + "]";
        break;
      case SyncStatusState.APPLICATION_DATA_UPLOADING:
        infoMessage = "DATA UPLOADING [" + interval + "] "
          + statusData.getCurrentMBO() + ": (S>"
          + statusData.getSentByteCount() + ":"
          + statusData.getSentRowCount() + " R<"
          + statusData.getReceivedByteCount() + ":"
          + statusData.getReceivedRowCount() + ")";
        break;
      case SyncStatusState.APPLICATION_SYNC_RECEIVING_UPLOAD_ACK:
        infoMessage = "RECEIVING UPLOAD ACK [" + interval + "]";
        break;
      case SyncStatusState.APPLICATION_DATA_UPLOADING_DONE:
        infoMessage = "UPLOAD DONE [" + interval + "] "
          + statusData.getCurrentMBO() + ": (S>"
          + statusData.getSentByteCount() + ":"
          + statusData.getSentRowCount() + " R<"
          + statusData.getReceivedByteCount() + ":"
          + statusData.getReceivedRowCount() + ")";
        break;
      case SyncStatusState.APPLICATION_DATA_DOWNLOADING:
        infoMessage = "DATA DOWNLOADING[" + interval + "] "
          + statusData.getCurrentMBO() + ": (S>"
          + statusData.getSentByteCount() + ":"
          + statusData.getSentRowCount() + " R<"
          + statusData.getReceivedByteCount() + ":"
          + statusData.getReceivedRowCount() + ")";
        break;
      case SyncStatusState.APPLICATION_SYNC_DISCONNECTING:
        infoMessage = "DISCONNECTING [" + interval + "]";
        break;
      case SyncStatusState.APPLICATION_SYNC_COMMITTING_DOWNLOAD:
        infoMessage = "COMMITTING DOWNLOAD [" + interval + "] "
          + statusData.getCurrentMBO() + ": (S>"
          + statusData.getSentByteCount() + ":"
          + statusData.getSentRowCount() + " R<"
          + statusData.getReceivedByteCount() + ":"
          + statusData.getReceivedRowCount() + ")";
        break;
      case SyncStatusState.APPLICATION_SYNC_CANCELLED:
        infoMessage = "SYNC CANCELED [" + interval + "]";
        break;
      case SyncStatusState.APPLICATION_DATA_DOWNLOADING_DONE:
        infoMessage = "DATA DOWNLOADING DONE [" + interval + "]";
        break;
      case SyncStatusState.SYNC_DONE:
        infoMessage = "DONE [" + interval + "]";
        break;
      default:
        infoMessage = "STATE" + syncState + "[" + interval + "]";
        break;
    }
    LogMessage(infoMessage);
    return false;
  }
}
```

The application can pass an instance of an implementation of `SyncStatusListener` to the synchronize API of the generated package database class to monitor the synchronization status.

```
SUP101DB.synchronize(new MySyncStatusListener())
```

Connecting to the Device Database

Establish a connection to the database on the device.

After completing the device registration, call the generated database's `openConnection` method to connect to the UltraLite/UltraLiteJ database on the device. If no device database exists, the `openConnection` method creates one.

See also

- *Setting Up Callbacks* on page 31

Synchronizing Applications

Synchronize package data between the device and the server.

The generated database provides you with synchronization methods that apply to either all synchronization groups in the package or a specified list of groups.

For information on synchronizing DOE-based applications, see *Message-Based Synchronization APIs* in the *Client Object API Usage* section of this document.

See also

- *Specifying Personalization Parameters* on page 38
- *Synchronization APIs* on page 75
- *Specifying Synchronization Parameters* on page 39

Configuring Data Synchronization Using SSL Encryption for Non-DOE-Based Applications

Enable SSL encryption by configuring the synchronization HTTPS port.

1. In the left navigation pane of Sybase Control Center for Unwired Platform, expand the **Servers** node and click the server name.
2. Click **Server Configuration**.
3. In the right administration pane, click the **Replication** tab.
4. Select **Secure synchronization port** 2481 as the protocol used for synchronization, and configure the certificate properties. In the optional properties section, specify the security certificate file, the public security certificate file using the fully qualified path to the file, along with the password you entered during certificate creation.

Nonblocking Synchronization

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

Subscribe to the package using synchronization APIs in the generated database class, specify the groups to be synchronized, and invoke the asynchronous synchronization method (`beginSynchronize`).

1. If you have not yet synchronized with Unwired Server, perform a synchronization.

   ```
   SUP101DB.synchronize("system")
   ```

2. Set the synchronization parameters if there are any.

   ```
   CustomerSynchronizationParameters syncParameter =
   Customer.getSynchronizationParameters();
   syncParameter.setYourParameters(...);
   syncParameter.save();
   ```

3. Make a blocking synchronize call to Unwired Server to pull in all MBO data:

   ```
   SUP101DB.synchronize();
   ```

4. List all customer MBO instances from the local database using an object query, such as `FindAll`, which is a predefined object query.

   ```
   GenericList<Customer> customers = Customer.findAll();
   int n = customers.size();
   for (int i = 0; i < n; ++i )
   {
     Customer customer = customers.get(i);
     //Work on customer information
   }
   ```

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

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

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

   ```
   cust.submitPending();
   ```

7. Use non-blocking synchronize call to upload the pending changes to the Unwired Server. The previous replay results and new changes are downloaded to the client device in the download phase of the synchronization session.

   ```
   GenericList<SynchronizationGroup> sgs = new
   GenericList<SynchronizationGroup>();
   sgs.add(SUP101DB.getSynchronizationGroup("default")); // Customer
   MBO is in "default" sync group
   SUP101DB.beginSynchronize(sgs, "mycontext");
   ```

Enabling Change Notifications

A synchronization group can enable or disable its change notification.

By default, change notifcations are disabled for synchronization groups. To enable change notification, call the `SynchronizationGroup` object's `setEnableSIS` method.

```
com.sybase.persistence.SynchronizationGroup sg =
SUP101DB.getSynchronizationGroup("PushEnabled");

if (!sg.getEnableSIS())
{
  sg.setEnableSIS(true);
  sg.setInterval(2);
  sg.save();
  SUP101DB.synchronize("PushEnabled");
}
```

Specifying Personalization Parameters

Use personalization parameters to provide default values used with synchronization, connections with back-end systems, MBO attributes, or EIS arguments. The `PersonalizationParameters` class is within the generated code for your project.

1. To instantiate a `PersonalizationParameters` object, call the generated database instance's `getPersonalizationParameters` method:

   ```
   PersonalizationParameters pp =
   SUP101DB.getPersonalizationParameters();
   ```
2. Assign values to the `PersonalizationParameters` object:

   ```
   pp.setPKCity( "New York" );
   ```
3. Save the `PersonalizationParameters` value to the local database:

   ```
   pp.save();
   ```

   Note: If you define a default value for a personalization key that value will not take effect, unless you call `pp.save()`.
4. Synchronize the `PersonalizationParameters` value to the Unwired Server:

   ```
   SUP101DB.synchronize();
   ```

See also

- *Synchronizing Applications* on page 36
- *Personalization APIs* on page 74

Specifying Synchronization Parameters

Use synchronization parameters within the mobile application to download filtered MBO data.

Assign the synchronization parameters of an MBO before a synchronization session. The next synchronize sends the updated synchronization parameters to the server. The `SynchronizationParameters` class is within the generated code for your project.

Note: If you do not save the `SynchronizationParameters`, no data is downloaded to the device even if there are default values set for those `SynchronizationParameters`. Call the `save` method for all `SynchronizationParameters` and for all MBOs when the application is first started. Do this after application registration and the first synchronization. This only applies to non-DOE-based applications.

1. Retrieve the synchronization parameters object from the MBO instance. For example, if you have an MBO named `Customer`, the synchronization parameters object is accessed as a public field and returned as a `CustomerSynchronizationParameters` object:

   ```
   CustomerSynchronizationParameters sp =
   Customer.getSynchronizationParameters();
   ```
2. Assign values to the synchronization parameter. For example, if the Customer MBO contains a parameter named `cityname`, assign the value to the `CustomerSynchronizationParameters` object's `Cityname` field:

   ```
   sp.setCityname("Kansas City");
   ```
3. Save your changes by calling the synchronization parameters object's `save` method:

   ```
   sp.save();
   ```

   Note: If you defined a default value or bound a PersonalizationParameters in the `SynchronizationParameters`, then that value will not take effect unless you call `sp.save()`.

   After you save the synchonization parameters, call another **synchronize()** to download the data.

   For DOE-based applications, call `SUP101DB.beginSynchronize()` to download the data.
4. When using synchronization parameters to retrieve data from an MBO during a synchronization session, clear the previous synchronization parameter values:

   ```
   CustomerSynchronizationParameters sp =
   Customer.getSynchronizationParameters();
   sp.delete();
   sp = Customer.getSynchronizationParameters();//Must re-get
   synchronization parameter instance.
   sp.setCityname("New City");
   sp.save();
   ```

See also

- *Synchronizing Applications* on page 36
- *Synchronization APIs* on page 75

Subsequently Starting an Application

Subsequent start-ups are different from the first start-up.

Starting an application on subsequent occasions:

1. Set up the `com.sybase.mobile.Application` instance with the required `com.sybase.mobile.ConnectionProperties`, including user credentials.
2. Set up the connection profile properties if needed for database location and tuning parameters.
3. Set up the synchronization profile properties if needed for SSL or a relay server.
4. Start the application connection to the server.
5. Open the database connection.

You can open the database connection in parallel with starting the application connection to the server.

```
// Calls non-blocking startConnection
// This call will return immediately.
application.startConnection();

// Open the device database connection while establishing
// the messaging channel connection in the background
SUP101DB.openConnection();

// Once the device database connection has been opened, check
// whether the messaging channel is connected using the
// ApplicationCallback interface or the
Application.getConnectionStatus() API
```

See also

- *Application APIs* on page 57

Accessing MBO Data

Use MBO object queries to retrieve lists of MBO instances, or use dynamic queries that return results sets or object lists.

See also

- *Query APIs* on page 112
- *Object Queries* on page 41
- *Dynamic Queries* on page 41
- *MBOs with Complex Types* on page 42

- *Relationships* on page 43

Object Queries

Use the generated static methods in the MBO classes to retrieve MBO instances.

1. To find all instances of an MBO, invoke the static `findAll` method contained in that MBO. For example, an MBO named `Customer` contains a method such as `public static com.sybase.collections.GenericList<SUP101.Customer> findAll()`.
2. To find a particular instance of an MBO using the primary key, invoke `MBO.findByPrimaryKey(...)`. For example, if a Customer has the primary key "id" as int, the Customer MBO would contain the `public static Customer findByPrimaryKey(int id)` method, which performs the equivalent of `Select x.* from Customer x where x.id = :id`.

If the return type is a list, additional methods are generated for you to further process the result, for example, to use paging. For example, consider this method, which returns a list of MBOs containing the specified city name: `com.sybase.collections.GenericList<SUP101.Customer> findByCity(String city, int skip, int take);`. The `skip` parameter specifies the number of rows to skip, and the `take` parameter specifies the maximum number of rows to return.

See also

- *Accessing MBO Data* on page 40
- *Query APIs* on page 112

Dynamic Queries

Build queries based on user input.

Use the `com.sybase.persistence.Query` class to retrieve a list of MBOs.

1. Specify the where condition used in the dynamic query.

```
Query query = new Query();

AttributeTest aTest = new AttributeTest();

aTest.setAttribute("state");
aTest.setTestValue("NY");
aTest.setTestType(AttributeTest.EQUAL);
query.setTestCriteria(aTest);

SortCriteria sort = new SortCriteria();
sort.add("lname", SortOrderType.ASCENDING);
sort.add("fname", SortOrderType.ASCENDING);
query.setSortCriteria(sort);
```

2. Use the `findWithQuery` method in the MBO to dynamically retrieve a list of MBOs acccording to the specified attributes.

```
GenericList<Customer> customers = Customer.findWithQuery(query);
int n = customers.size();
for (int i = 0; i < n; ++i)
{
   Customer c = (Customer)customers.get(i);
   System.out.println("Customer " + i + ": "
       + c.getLname() + ", "  + c.getFname());
}
```

3. Use the generated database's `executeQuery` method to query multiple MBOs through the use of joins.

```
Query query = new Query();

query.select("c.fname,c.lname,s.order_date,s.id");
query.from("Customer", "c");
query.join("Sales_order", "s", "s.cust_id", "c.id");

AttributeTest ts = new AttributeTest();
ts.setAttribute("lname");
ts.setTestValue("Smith");
ts.setOperator(AttributeTest.EQUAL);
query.setTestCriteria(ts);
QueryResultSet qrs = SUP101DB.executeQuery(query);

while(qrs.next())
{
   System.out.println("order: " +
      qrs.getInt(4) +           // 4 is s.id
      qrs.getString(1) +        // 1 is c.fname
      ", " + qrs.getString(2) + // 2 is c.lname
      " "  + qrs.getDate(3));   // 3 is s.order_date
}
```

See also

- *Accessing MBO Data* on page 40
- *Query APIs* on page 112

MBOs with Complex Types

Mobile business objects are mapped to classes containing data and methods that support synchronization and data manipulation. You can develop complex types that support interactions with backend data sources such as SAP® and Web services. When you define an MBO with complex types, Sybase Unwired Platform generates one class for each complex type.

Using a complex type to create an MBO instance.

1. Suppose you have an MBO named `SimpleCaseList` and want to use a complex data type called `AuthenticationInfo` to its `Create` method's parameter. Begin by creating the complex datatype:

   ```
   AuthenticationInfo authen = new AuthenticationInfo();
   authen.setUserName("Demo");
   ```

2. Instantiate the MBO object:

   ```
   SimpleCaseList newCase = new SimpleCaseList();
   newCase.setCase_Type("Incident");
   newCase.setCategory("Networking");
   newCase.setCreate_Time(new
   java.sql.Timestamp(System.currentTimeMillis()));
   ```

3. Call the `create` method of the SimpleCaseList MBO with the complex type parameter as well as other parameters, and call `submitPending()` to submit the `create` operation to the operation replay record. Subsequent synchronizations upload the operation replay record to the Unwired Server and get replayed.

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

See also

- *Accessing MBO Data* on page 40
- *Query APIs* on page 112

Relationships

The Object API supports one-to-one, one-to-many, and many-to-one relationships.

Navigate between MBOs using relationships.

1. Suppose you have one MBO named `Customer` and another MBO named `SalesOrder`. This code illustrates how to navigate from the `Customer` object to its child `SalesOrder` objects:

   ```
   Customer cust = Customer.findById(101);
   GenericList<Sales_order> orders = cust.getSalesOrders();
   ```

2. To filter the returned child MBO's list data, use the `Query` class:

   ```
   Query query = new Query();
   AttributeTest at = AttributeTest.equal("theAttribute",
   "theTestValue");
   query.where(at);
   orders = cust.getSalesOrdersFilterBy(query);
   ```

3. For composite relationship, you can call the parent's `SubmitPending` method to submit the entire object tree of the parent and its children. Submitting the child MBO also submits the parent and the entire object tree. (If you have only one child instance, it would not make any difference. To be efficient and get one transaction for all child operations, it is recommened to submit the parent MBO once, instead of submitting every child).

If the primary key for a parent is assigned by the EIS, you can use a multilevel insert cascade operation to create the parent and child objects in a single operation without synchronizing multiple times. The returned primary key for the parent's `create` operation populates the children prior to their own creation.

The following example illustrates how to submit the parent MBO which also submits the child's operation:

```
Customer cust = Customer.findById(101);
Sales_order order = new Sales_order();
order.setId(1001);
order.setCustomer(cust);
order.setOrder_date(new Date(System.currentTimeMillis()));
order.setFin_code_id("r1");
order.setRegion("Eastern");
order.setSales_rep(101);
order.save(); // or order.create();
cust.save();
cust.submitPending();
```

See also

- *Accessing MBO Data* on page 40
- *Query APIs* on page 112

Manipulating Data

Create, update, and delete instances of generated MBO classes.

You can create a new instance of a generated MBO class, fill in the attributes, and call the `create` method for that MBO instance.

You can modify an object loaded from the database by calling the `update` method for that MBO instance.

You can load an MBO from the database and call the `delete` method for that instance.

See also

- *Persistence APIs* on page 122

Creating, Updating, and Deleting MBOs

Perform create, update, and delete operations on MBO instances.

You can call the `create`, `update`, and `delete` methods for MBO instances.

Note: For MBOs with custom create or update operations with parameters, you should use the custom operations, rather than the default `create` and `update` operations. See *MBOs with Complex Types*.

1. Suppose you have an MBO named `Customer`. To create an instance within the database, invoke its `create` method, which causes the object to enter a pending state. Then call the MBO instance's `submitPending` method. Finally, synchronize with the generated database:

   ```
   Customer newcustomer = new Customer();
   //Set the required fields for the customer
   // …

   newcustomer.create();
   newcustomer.submitPending();
   SUP101DB.synchronize();
   ```

   For DOE-based applications, omit `SUP101DB.synchronize;` above.
2. To update an existing MBO instance, retrieve the object instance through a query, update its attributes, and invoke its `update` method, which causes the object to enter a pending state. Then call the MBO instance's `submitPending` method. Finally, synchronize with the generated database:

   ```
   Customer customer = Customer.findByPrimaryKey(myCustomerId); //
   find by primary key
   customer.setCity("Dublin"); //update any field to a new value
   customer.update();
   customer.submitPending();
   SUP101DB.synchronize();
   ```

   For DOE-based applications, omit `SUP101DB.synchronize;` above.
3. To delete an existing MBO instance, retrieve the object instance through a query and invoke its `delete` method, which causes the object to enter a pending state. Then call the MBO instance's `submitPending` method. Finally, synchronize with the generated database:

   ```
   Customer customer = Customer.findByPrimaryKey(myCustomerId); //
   find by primary key
   customer.delete();
   customer.submitPending();
   SUP101DB.synchronize();
   ```

   For DOE-based applications, omit `SUP101DB.synchronize;` above.

See also
- *Operations APIs* on page 123

Other Operations

Use operations other than create, update, or delete.

In this example, a customized operator is used to perform a sum operation.

1. Suppose you have an MBO named `MyMBO` that has an operator that generates a customized sum. Begin by creating an object instance and assigning values to its attributes, specifying the `"Add"` operation:

```
MyMBOAddOperation op = new MyMBOAddOperation(); //Convension is
<MBO Name>+<Operation Name>+"Operation"

op.setOperand1(12);
op.setOperand2(23);
op.setOperator("Add");
op.save();
```

2. Call the MBO instance's `submitPending` method and synchronize with the generated database:

   ```
   op.submitPending();
   SUP101DB.synchronize();
   ```

   With DOE-based applications, omit `SUP101DB.Synchronize();` above.

See also

- *Operations APIs* on page 123

Using SubmitPending and SubmitPendingOperations

You can submit a single pending MBO, all pending MBOs of a single type, or all pending MBOs in a package. Once those pending changes are submitted to the server, the MBOs enter a replay pending state.

Note that **submitPendingOperations** APIs are expensive. Sybase recommends using the **submitPending** API with the MBO instance whenever possible.

Database Classes

Submit pending operations for all entities in the package or synchronization group, cancel all pending operations that have not been submitted to the server, and check if there are pending oprations for all entities in the package.

1. To submit pending operations for all pending entities in the package, invoke the generated database's `submitPendingOperations` method.

   Note that **submitPendingOperations** APIs are expensive. Sybase recommends using the **submitPending** API with the MBO instance whenever possible.
2. To submit pending operations for all pending entities in the specified synchronization group, invoke the generated database's `submitPendingOperations (string synchronizationGroup)` method.
3. To cancel all pending operations that have not been submitted to the server, invoke the generated database's `cancelPendingOperations` method.

Generated MBOs

Submit pending operations for all entities for a given MBO type or a single instance, and cancel all pending operations that have not been submitted to the server for the MBO type or a single entity.

1. To submit pending operations for all pending entities for a given MBO type, invoke the MBO class' static `submitPendingOperations` method.

   Note that **submitPendingOperations** APIs are expensive. Sybase recommends using the **submitPending** API with the MBO instance whenever possible.
2. To submit pending operations for a single MBO instance, invoke the MBO object's `submitPending` method.
3. To cancel all pending operations that have not been submitted to the server for the MBO type, invoke the MBO class' static `cancelPendingOperations` method.
4. To cancel all pending operations for a single MBO instance, invoke the MBO object's `cancelPending` method.

Shutting Down the Application

Shut down an application and clean up connections.

Closing Connections

Clean up connections from the generated database instance prior to application shutdown.

1. To release an opened application connection, stop the messaging channel by invoking the application instance's `stopConnection` method.

   ```
   // wait the timeout value for the connection to stop
   // if it is not stopped within the timeout value an exception will
   be thrown
   app.stopConnection(<timeout_value>);
   ```
2. Close all connections to device database by calling the `closeConnection` method in the generated package database class. If one application has multiple packages, invoke the `closeConnection` API in all the packages.

Uninstalling the Application

Uninstall the application and clean up all package- and MBO-level data.

Deleting the Database and Unregistering the Application

Delete the package database, and unregister the application.

1. To delete the package database, call the generated database's `deleteDatabase` method.

   ```
   SUP101DB.deleteDatabase();
   ```
2. Unregister the application by invoking the `Application` instance's `unregisterApplication` method.

```
app.unregisterApplication(<time out value>);
```

Testing Applications

Test native applications on a device or simulator.

See also

- *Customizing the Application Using the Object API* on page 25
- *Localizing Applications* on page 53

Testing an Application Using a Emulator

Run and test the application on an emulator and verify that the application automatically registers to Unwired Server using the default application connection template.

Prerequisites

You must have created an Android Virtual Device when you installed the Android SDK in your Android development environment. The Android Virtual Device (AVD) must use the same target as the test package.

Task

1. In the Eclipse Package Explorer, right-click the project and select **Run As** > **Android Application**.
   The ADT plugin for Eclipse installs your application, starts the emulator automatically, and launches the application. The application will automatically register with Unwired Server using the default application connection template.
2. In Sybase Control Center verify that the application connection was created in **Applications** > **Application Connections**.
   When the application has successfully registered, the application connection displays a value of zero in the Pending Items column.
3. Test the functionality of the application. Use debug tools as necessary, setting breakpoints at appropriate places in the application.

Client-Side Debugging

Identify and resolve client-side issues while debugging the application.

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

- Unwired Server connection failed - use your device browser to check the connectivity of your device to the server.
- Data does not appear on the client device - check if your synchronization and personalization parameters are set correctly. If you are using queries, check if your query conditions are correctly constructed and if the device data match your query conditions.
- Physical device problems, such as low memory - implement `ApplicationCallback.onDeviceConditionChanged` to be notified if device storage gets too low, or recovers from an error.

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 the API Reference information about using the `Logger` class to add logs to the client log record and synchronize them to the server (viewable in Sybase Control Center).
- Check the log record on the device. Use the **<PkgName>DB.getLogRecords (com.sybase.persistence.Query)** or **Entity.getLogRecords()** methods.

  This is the log format

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

  This log format generates output similar to:

  ```
  level code eisCode message component entityKey operation requestId
  timestamp
   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` – Unwired Server administration codes.
    - Synchronization codes:
      - 200 – success.
      - 500 – failure.
  - `eisCode` – maps to HTTP error codes. If no mapping exists, defaults to error code 500 (an unexpected server failure).
  - `message` – the message content.
  - `component` – 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.
- If you have implemented `ApplicationCallback`.**onConnectionStatusChanged** for synchronization in the `CallbackHandler`, the connection status between Unwired Server and the device is reported on the device. See the `CallbackHandler` API reference information. The device connection status, device connection type, and connection error message are reported on the device:
    - 1 – current device connection status.
    - 2 – current device connection type.
    - 3 – connection error message.
- For other issues, you can turn on SQLTrace trace on the device side to trace Client Object API activity. To enable SQLTrace using the ConnectionProfile's enableTrace API:

```
// To enable SQL trace with values also displayed
SUP101DB.getConnectionProfile().enableTrace(true, true);
```

Server-Side Debugging

Identify and resolve server-side issues while debugging the application.

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

- The domain or package does not exist. If you create a new domain, with a default status of disabled, it is unavailable until enabled.
- Authentication failed for the application user credentials.
- 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.
- An operation failed on the Web Service, REST, or SAP® back end.

To find out more information on the Unwired Server side:

- Check the Unwired Server log files.
- For message-based synchronization mode, you can set the log level to DEBUG to obtain detailed information in the log files:
    1. Set the log level using Sybase Control Center. See *Sybase Control Center for Unwired Platform > Administer > Server Log > Configuring Server Log Setting*.

    Note: Return to INFO mode as soon as possible, since DEBUG mode can affect system performance.
- Obtain DEBUG information for a specific device:
    - In the SCC administration console:

1. Set the DEBUG level to a higher value for a specified device:
   a. In SCC, select **Application Connections**, then select **Properties... > Device Advanced**.
   b. Set the Debug Trace Level value.
2. Set the TRACE file size to be greater than 50KB.
3. View the trace file through SCC.

- Check the `<server_install_folder>\UnwiredPlatform\Servers\MessagingServer\Data\ClientTrace` directory to see the mobile device client log files for information about a specific device.

Note: Return to INFO mode as soon as possible, since DEBUG mode can affect system performance.

Localizing Applications

Localize an Android application by creating default and alternate resources.

For information, best practices, and tutorials on localizing Android applications, see *http://developer.android.com/guide/topics/resources/localization.html*

See also

- *Testing Applications* on page 49

Packaging Applications

Package applications according to your security or application distribution requirements.

You can package all libraries into one package. This packaging method provide more security since packaging the entire application as one unit reduces the risk of tampering of individual libraries.

You may package and install modules separately only if your application distribution strategy requires sharing libraries between Sybase Unwired Platform applications.

Once you build your application, deploy the Android package (APK) file. For more information on publishing your Android application, see *http://developer.android.com/guide/publishing/publishing_overview.html*.

Signing

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

All applications must be signed. The system will not install an application on an emulator or a device if it is not signed.

To test and debug your application, the build tools sign your application with a special debug key that is created by the Android SDK build tools.

Client Object API Usage

The Sybase Unwired Platform Client Object API consists of generated business object classes that represent mobile business objects (MBOs) that are designed and built in the Unwired WorkSpace development environment. Device applications use the Client Object API to retrieve data and invoke mobile business object operations.

Refer to these sections for more information on using the APIs described in *Developer Guide: Android Object API Applications* > *Customizing the Application Using the Object API*.

Client Object API Reference

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

Review the reference details in the Client Object API documentation, located in the Unwired Platform installation directory `<UnwiredPlatform_InstallDir>\MobileSDK \ObjectAPI\apidoc`.

There is a subdirectory for `android`.

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.

Application APIs

The `Application` class, in the `com.sybase.mobile` Java package, manages mobile application registrations, connections and context.

Note: Sybase recommends that you use the Application API operations with no timeout parameter, and register an ApplicationCallback to handle completion of these operations.

See also

- *Initially Starting an Application* on page 25
- *Setting Up Application Properties* on page 26
- *Registering an Application* on page 27
- *Subsequently Starting an Application* on page 40

getInstance

Retrieves the `Application` instance for the current mobile application.

Syntax

```
public static Application getInstance()
```

Returns

`getInstance` returns a singleton `Application` object.

Examples

- **Get the Application Instance –**

```

Application app = Application.getInstance();
```

setApplicationIdentifier

Sets the identifier for the current application.

Set the application identifer before calling `startConnection`, `registerApplication` or `unregisterApplication`.

Syntax

```
public void setApplicationIdentifier(java.lang.String value)
```

Parameters

- **value –** The identifier for the current application.

Examples

- **Set the Application Identifier –** Sets the application identifier to `SUP101`.

  Note: The application identifier is case sensitive.

```
// Initialize Application settings
Application app = Application.getInstance();

// The identifier has to match the
// application ID deployed to the SUP server
app.setApplicationIdentifier("SUP101");
```

Usage

This method must be called in the user interface thread.

getRegistrationStatus

Retrieves the current status of the mobile application registration.

Syntax

```
public int getRegistrationStatus()
```

Returns

`getRegistrationStatus` returns one of the values defined in the `RegistrationStatus` class.

```
public class RegistrationStatus {

public static final int REGISTERED = 203;
public static final int REGISTERING = 202;
public static final int REGISTRATION_ERROR = 201;
public static final int UNREGISTERED = 205;
public static final int UNREGISTERING = 204;
}
```

Examples

- **Get the Registration Status –** Registers the application if it is not already registered.

  ```
  if (app.getRegistrationStatus() != RegistrationStatus.REGISTERED)
  {
     // If the application has not been registered to the server,
     // register now
     app.registerApplication();
  }
  else
  {
     // start the connection to server
     app.startConnection();
  }
  ```

registerApplication

Creates the registration for this application and starts the connection. This method is equivalent to calling `registerApplication(0)`, but is a non-blocking call which returns immediately.

Syntax

```
public void registerApplication()
```

Parameters

None.

Examples

- **Register an Application –** Start registering the application and return at once.

```
   app.registerApplication();
```

Usage

You must set up the `ConnectionProperties` and `ApplicationIdentifier` before you can invoke `registerApplication`.

```
Application app = Application.getInstance();
// set Application ID - need to match as the server side Application
ID
app.setApplicationIdentifier("SUP101");
app.setApplicationCallback(new MyApplicationCallbackHandler());
ConnectionProperties props = app.getConnectionProperties();
props.setServerName("supserver.mycompany.com");
props.setPortNumber(5001);
LoginCredentials loginCred = new LoginCredentials("supAdmin",
"supPwd");
props.setLoginCredentials(loginCred);

SUP101DB.setApplication(app);

if (app.getRegistrationStatus() != RegistrationStatus.REGISTERED)
{
  app.registerApplication();
}
```

registerApplication (int timeout)

Creates the registration for this application and starts the connection. An `ApplicationTimeoutException` is thrown if the method does not succeed within the number of seconds specified by the timeout.

If a callback handler is registered and network connectivity is available, the sequence of callbacks as a result of calling `registerApplication` is:

```
onRegistrationStatusChanged(RegistrationStatus.REGISTERING, 0, "")
onConnectionStatusChanged(ConnectionStatus.CONNECTING, 0, "")
onConnectionStatusChanged(ConnectionStatus.CONNECTED, 0, "")
onRegistrationStatusChanged(RegistrationStatus.REGISTERED, 0, "")
```

When the connectionStatus of CONNECTED has been reached and the application's applicationSettings have been received from the server, the application is now in a suitable state for database subscriptions and/or synchronization. If a callback handler is registered and network connectivity is unavailable, the sequence of callbacks as a result of calling registerApplication is:

```
onRegistrationStatusChanged(RegistrationStatus.REGISTERING, 0, "")
onRegistrationStatusChanged(RegistrationStatus.REGISTRATION_ERROR,
code, message)
```

In such a case, the registration process has permanently failed and will not continue in the background. If a callback handler is registered and network connectivity is available for the start of registration but becomes unavailable before the connection is established, the sequence of callbacks as a result of calling registerApplication is:

```
onRegistrationStatusChanged(RegistrationStatus.REGISTERING, 0, "")
onConnectionStatusChanged(ConnectionStatus.CONNECTING, 0, "")
onConnectionStatusChanged(ConnectionStatus.CONNECTION_ERROR, code,
message)
```

In such a case, the registration process has temporarily failed and will continue in the background when network connectivity is restored.

Syntax

```
public void registerApplication(int timeout)
```

Parameters

- **timeout –** Number of seconds to wait until the registration is created. If the the timeout is greater than zero and the registration is not created within the timeout period, an `ApplicationTimeoutException` is thrown (the operation might still be completing in a background thread). If the timeout value is less than or equal to 0, then this method returns immediately without waiting for the registration to finish (a non-blocking call).

Examples

- **Register an Application –** Registers the application with a one minute waiting period.

  ```
  app.registerApplication(60);
  ```

Usage

You must set up the `ConnectionProperties` and `ApplicationIdentifier` before you can invoke `registerApplication`.

```
Application app = Application.getInstance();
// set Application ID - need to match as the server side Application
ID
app.setApplicationIdentifier("SUP101");
app.setApplicationCallback(new MyApplicationCallbackHandler());
ConnectionProperties props = app.getConnectionProperties();
props.setServerName("supserver.mycompany.com");
props.setPortNumber(5001);
LoginCredentials loginCred = new LoginCredentials("supAdmin",
"supPwd");
props.setLoginCredentials(loginCred);

SUP101DB.setApplication(app);

if (app.getRegistrationStatus() != RegistrationStatus.REGISTERED)
{
app.registerApplication(60);
}
```

setApplicationCallback

Sets the callback for the current application. It is optional, but recommended, to register a callback so the application can respond to changes in connection status, registration status, and application settings.

Syntax

```
public void setApplicationCallback(ApplicationCallback value)
```

Parameters

- **value –** The mobile application callback handler.

Examples

- **Set the Application Callback –**

  ```
  // Initialize Application settings
  Application app = Application.getInstance();

  // The identifier has to match the
  // application ID deployed to the SUP server
  app.setApplicationIdentifier("SUP101");
  ApplicationCallback appCallback = new MyApplicationCallback();
  app.setApplicationCallback(appCallback);
  ```

getApplicationContext

Returns the Android application context which allows access to application-specific resources and classes.

Syntax

```
public android.content.Context getApplicationContext()
```

Returns

`getApplicationContext` returns a single Context object.

Examples

- **Get the Application Context –**

  ```
  getApplicationContext()
  ```

setApplicationContext

Sets the Android application context, which is required before calling the`startConnection`, `registerApplication` or `unregisterApplication` methods. This method must be called in an user interface thread, not a background thread.

Syntax

```
public void setApplicationContext(android.content.Context context)
```

Returns

None.

Examples

- **Set the Application Context –**
  ```
  setApplicationContext(android.content.Context context)
  ```

startConnection

Starts the connection for this application. This method is equivalent to calling `startConnection(0)`, but is a non-blocking call which returns immediately. Use `getConnectionStatus` or the `ApplicationCallback` to retrieve the connection status.

Syntax

```
public void startConnection()
```

Returns

None.

Examples

- **Start the Application –**
  ```
  startConnection()
  ```

startConnection (int timeout)

Starts the connection for this application. If the connection was previously started, then this operation has no effect. You must set the appropriate `connectionProperties` before calling this operation. An `ApplicationTimeoutException` is thrown if the method does not succeed within the number of seconds specified by the timeout.

If connection properties are improperly set, a `ConnectionPropertyException` is thrown. You can set the `applicationCallback` before calling this operation to receive asynchronous notification of connection status changes. If a callback handler is registered and network connectivity is available, the sequence of callbacks as a result of calling `startConnection` is:

```
onConnectionStatusChanged(ConnectionStatus.CONNECTING, 0, "")
 onConnectionStatusChanged(ConnectionStatus.CONNECTED, 0, "")
```

If a callback handler is registered and network connectivity is unavailable, the sequence of callbacks as a result of calling `startConnection` is:

```
onConnectionStatusChanged(ConnectionStatus.CONNECTING, 0, null)
 onConnectionStatusChanged(ConnectionStatus.CONNECTION_ERROR, code,
message)
```

After a connection is successfully established, it can transition at any later time to CONNECTION_ERROR status or NOTIFICATION_WAIT status and subsequently back to CONNECTING and CONNECTED when connectivity resumes.

Note: The application must have already been registered for the connection to be established. See *registerApplication* for details.

Syntax

```
public void startConnection(int timeout)
```

Parameters

- **timeout –** The number of seconds to wait until the connection is started. If the timeout is greater than zero and the connection is not started within the timeout period, an `ApplicationTimeoutException` is thrown (the operation may still be completing in a background thread).

Returns

None.

Examples

- **Start the Application –**
  ```
  startConnection(int timeout)
  ```

getConnectionStatus

Return current status of the mobile application connection.

Syntax

```
public int getConnectionStatus()
```

Returns

`connectionStatus` returns one of the `ConnectionStatus` class values.

Examples

- **Get the Application Connection Status –**

  ```
  getConnectionStatus()
  ```

stopConnection

Stops the connection for this application. This method is equivalent to calling stopConnection(0).

Syntax

```
public void stopConnection()
```

Returns

None.

Examples

- **Stop the Connection for the Application –**

  ```
  stopConnection();
  ```

stopConnection (int timeout)

Stop the connection for this application. An `ApplicationTimeoutException` is thrown if the method does not succeed within the number of seconds specified by the timeout.

If no connection was previously stopped, then this operation has no effect. You can set the applicationCallback before calling this operation to receive asynchronous notification of connection status changes.

If a callback handler is registered, the sequence of callbacks as a result of calling stopConnection is:

- onConnectionStatusChanged(ConnectionStatus.DISCONNECTING, 0, "")
- onConnectionStatusChanged(ConnectionStatus.DISCONNECTED, 0, "")

Syntax

```
public void stopConnection(int timeout)
```

Parameters

- **timeout –** The number of seconds to wait until the connection is stopped.

Returns

None.

Examples

- **Stop the Application –**
  ```
  stopConnection(60)
  ```

unregisterApplication

Delete the registration for this application, and stop the connection. If no registration was previously created, or a previous registration was already deleted, then this operation has no effect. This method is equivalent to calling `unregisterApplication(0)`, but is a non-blocking call which returns immediately. You can set the applicationCallback before calling this operation to receive asynchronous notification of registration status changes.

Syntax

```
unregisterApplication()
```

Parameters

None.

Examples

- **Unregister an Application –** Unregisters the application.
  ```
  app.unregisterApplication();
  ```

unregisterApplication(int timeout)

Delete the registration for this application, and stop the connection. If no registration was previously created, or a previous registration was already deleted, then this operation has no effect. You can set the applicationCallback before calling this operation to receive asynchronous notification of registration status changes.

If a callback handler is registered and network connectivity is available, the sequence of callbacks as a result of calling unregisterApplication should be:

- onConnectionStatusChanged(ConnectionStatus.DISCONNECTING, 0, "")
- onConnectionStatusChanged(ConnectionStatus.DISCONNECTED, 0, "")
- onRegistrationStatusChanged(RegistrationStatus.UNREGISTERING, 0, "")
- onRegistrationStatusChanged(RegistrationStatus.UNREGISTERED, 0, "")

If a callback handler is registered and network connectivity is unavailable, the sequence of callbacks as a result of calling unregisterApplication should be:

- onConnectionStatusChanged(ConnectionStatus.DISCONNECTING, 0, "")
- onConnectionStatusChanged(ConnectionStatus.DISCONNECTED, 0, "")
- onRegistrationStatusChanged(RegistrationStatus.UNREGISTERING, 0, "")
- onRegistrationStatusChanged(RegistrationStatus.REGISTRATION_ERROR, code, message)

Syntax

```
unregisterApplication(int timeout)
```

Parameters

- **timeout –** Number of seconds to wait until the application is unregistered.

Examples

- **Unregister an Application –** Unregisters the application with a one minute waiting period.

  ```
  app.unregisterApplication(60);
  ```

Connection APIs

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

See also

- *Initially Starting an Application* on page 25

ConnectionProfile

The `ConnectionProfile` class manages local database information. Set its properties, including the encryption key, during application initialization, and before creating or accessing the local client database.

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

```
ConnectionProfile profile = SUP101DB.getConnectionProfile();
profile.setPageSize( 4*1024 );
profile.setEncryptionKey("Your key of more than 16 characters");
```

You can also generate an encryption key by calling the generated database's `generateEncryptionKey` method, and then store the key inside a `DataVault` object. The `generateEncryptionKey` method automatically sets the encryption key in the connection profile.

See also

- *Setting Up the Connection Profile* on page 28

Managing Device Database Connections

Use the `openConnection()` and `closeConnection()` methods generated in the package database class to manage device database connections.

Note: Any database operation triggers the establishment of the database connection. You do not need to explicitly call the `openConnection` API.

The `openConnection()` method checks that the package database exists, creates it if it does not, and establishes a connection to the database. This method is useful when first starting the application: since it takes a few seconds to open the database when creating the first connection, if the application starts up with a login screen and a background thread that performs the `openConnection()` method, after logging in, the connection is most likely already established and is immediately available to the user.

All `ConnectionProfile` properties should be set before the first access to database, otherwise they will not take effect.

The `closeConnection()` method closes all database connections for this package and releases all resources allocated for those connections. This is recommended to be part of the application shutdown process.

Improving Device Application Performance with One Writer Thread and Multiple Database Access Threads

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

Connection management allows you to have at most one writer thread concurrent with multiple reader threads. There can be other reader threads at the same time that the writer thread is writing to the database. The total number of threads are controlled by the `maxDbConnections` property.

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

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

You can specify the number of total threads using `maxDbConnections`.

The `ConnectionProfile` class in the persistence package includes the `maxDbConnections` property, which you set before performing any operation in the application. The default value (maximum number of concurrent read threads) is 2.

```
ConnectionProfile connectionProfile =
SUP101DB.getConnectionProfile();
```

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

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

Set Database File Property

You can use `setProperty` to specify the database file name on the device, such as the directory of the running program, a specific directory path, or a secure digital (SD) card.

```
ConnectionProfile cp = SUP101DB.getConnectionProfile();
cp.setProperty("databaseFile", "SUP101.ulj");
cp.save();
```

Examples

If you specify the *databaseFile* name only, with no path, the *databaseFile* is created in the path where the program is running:

```
mydb.ulj
```

To store the database on the SD card:

```
Environment.getExternalStorageDirectory().getAbsolutePath() + "/
mydb.ulj"
```

Note: For the database file path and name, the forward slash (/) is required as the path delimiter, for example /mnt/sdcard/sup101.ulj.

Usage

- Be sure to call this API before the database is created..
- The database is UltraLiteJ™; use an absolute path to the database file name like `/sdcard/mydb.ulj`.
- If the device client user changes the file name, he or she must make sure the input file name is a valid name and path on the client side.

Note: Sybase recommends using industrial grade SD cards using Single Level Cell (SLC) technology. SD cards that use SLC technology are generally more reliable and faster than MLC cards, although they may be more limited in size and more expensive per unit of storage. Not all SD cards perform equally, and it is advised that customers evaluate the benchmarks available from different suppliers.

Synchronization Profile

The Synchronization Profile contains information for establishing a connection with the Unwired Server's data synchronization channel where the server package has been deployed. The `com.sybase.persistence.ConnectionProfile` class manages that information. By default, this information includes the server host, port, domain name, certificate and public key that are pushed by the message channel during the registration process.

Settings are automatically provisioned from the Unwired Server. The values of the settings are inherited from the application connection template used for the registration of the application connection (automatic or manual). You must make use of the connection and security settings that are automatically used by the Object API.

Typically, the application uses the settings as sent from the Unwired Server to connect to the Unwired Server for synchronization so that the administrator can set those at the application deployment time based on their deployment topology (for example, using relay server, using e2ee security, or a certificate used for the intermediary, such as a Relay Server Web server). See the *Applications* and *Application Connection Templates* topics in *System Administration*.

```
ConnectionProfile profile = SUP101DB.getSynchronizationProfile();
profile.setDomainName( "default" );
profile.setServerName( "sup.sybase.com" );
profile.setPortNumber( 2480 );
profile.setNetworkProtocol( "http" );
profile.getStreamParams().setTrusted_Certificates( "rsa_public_cert
.crt" );
```

You can allow clients to compress traffic as they communicate with the Unwired Server by including "compression=zlib" into the sync parameters:

```
SUP101DB.getSynchronizationProfile().getStreamParams().setZlibCompr
ession(true);
```

See also

- *Setting Up the Synchronization Profile* on page 29

Connect the Data Synchronization Channel Through a Relay Server

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

Edit SUP101DB by modifying the values of the relay server properties for your Relay Server environment.

```
getSynchronizationProfile().setServerName("examplexp-vm1");
getSynchronizationProfile().setPortNumber(80);
getSynchronizationProfile().setNetworkProtocol("http");
NetworkStreamParams streamParams =
getSynchronizationProfile().getStreamParams();
streamParams.setUrl_Suffix("/cli/iarelayserver/<FarmName>");
getSynchronizationProfile().setDomainName("default");
```

To update properties for the relay server installed on Internet Information Services (IIS) on Microsoft Windows:

```
getSynchronizationProfile().setServerName("examplexp-vm1");
getSynchronizationProfile().setPortNumber(80);
getSynchronizationProfile().setNetworkProtocol("http");
NetworkStreamParams streamParams =
getSynchronizationProfile().getStreamParams();
streamParams.setUrl_Suffix("/ias_relay_server/client/rs_client.dll/
<FarmName>");
getSynchronizationProfile().setDomainName("default");
```

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

Asynchronous Operation Replay

When an application calls `submitPending` on an MBO on which a create, update, or delete operation is performed, an operation replay record is created on the device local database

When `synchronize` is called, the operation replay records are uploaded to the server. The method returns without waiting for the backend to replay those records. The `synchronize` method downloads all the latest data changes and the results of the previously uploaded operation replay records that the backend has finished replaying. If you choose to disable asynchronous operation replay, each `synchronize` call will wait for the backend to finish replaying all the current uploaded operation replay records.

By default, synchronization will not wait for the operations to be replayed on the backend. When the replay is finished, the onSynchronize callback method will be called with this status code in the SynchronizeContext:

```
SynchronizationStatus.ASYNC_REPLAY_COMPLETED
```

The application can set the following property in the synchronization profile to use the previous Synchronous Operation Replay behavior.

```
SUP101DB.getSynchronizationProfile().setAsyncReplay(false);
```

Authentication APIs

You can log in to the Unwired Server with your user name and credentials and use the X.509 certificate you installed in the task flow for single sign-on.

Logging In

The generated package database class provides a default synchronization connection profile according to the Unwired Server connection profile and server domain selected during code generation. You can log in to the Unwired Server with your user name and credentials.

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

- **`onlineLogin(String username, String password)`** – authenticates credentials against the Unwired Server.
- **`offlineLogin(String username, String password)`** – authenticates against the most recent successfully authenticated credentials. Once the client connects for the first time, the server validated user name and password are stored locally. `offlineLogin` verifies with the client database if those credentials are valid. The method returns YES if the user name and password are correct, otherwise the method returns NO.

  There is no communication with Unwired Server in this method. This method is useful if there is no connection the Unwired Server and you want to access the client application locally.

Sample Code

Illustrates importing the certificate and setting up login credentials, as well as other APIs related to certificate handling:

```
/// SUP101DB is a generated database class
///First install certificates on your simulator, for example
"Sybase101.p12"


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

// Save the login certificate to your synchronization profile
SUP101DB.getSynchronizationProfile().setCertificate(lc);

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

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

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

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

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

Refer to the API Reference for implementation details.

Importing a Certificate into the Data Vault

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

Importing a certificate from a system store is not supported on Android. You can only import a certificate binary large object (BLOB), which is a digitally signed copy of the public X.509 certificate, from a file directory.

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


// Import a certificate from a file on SDCard
String certFile = "/mnt/sdcard/mycert.p12";
String password = "my p12 password";

LoginCertificate cert =
certStore.getSignedCertificateFromFile(certFile, password);

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

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

Selecting a Certificate for Unwired Server Connections

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

```
LoginCertificate cert = LoginCertificate.load("myCert", vault);
ConnectionProfile syncProfile =
SUP101DB.getSynchronizationProfile();
syncProfile.setCertificate(cert);
```

Connecting to Unwired Server with a Certificate

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

```
SUP101DB.onlineLogin();
```

Personalization APIs

Personalization keys allow the application to define certain input parameter values that are personalized for each mobile user. Personalization parameters provide default values for synchronization parameters when the synchronization key of the object is mapped to the personalization key while developing a mobile business object. The Personalization APIs allow you to manage personalization keys, and get and set personalization key values.

See also

- *Specifying Personalization Parameters* on page 38

Type of Personalization Keys

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

A personalization parameter can be a primitive or complex type.

A personalization key is metadata that enables users to store their search preferences on the client, the server, or by session. The preferences narrow the focus of data retrieved by the mobile device (also known as the filtering of data between client and Unwired Server). Often personalization keys are used to hold backend system credentials, so that they can be propagated to the EIS. To use a personalization key for filtering, it must be mapped to a synchronization parameter. The developer can also define personalization keys for the application, and can use built-in personalization keys available in Unwired Server. Two built-in (session) personalization keys — username and password — can be used to perform single sign-on from the device application to the Unwired Server, authentication and authorization on Unwired Server, as well as connecting to the back-end EIS using the same set of credentials. The password is never saved on the server.

Getting and Setting Personalization Key Values

The `PersonalizationParameters` class is generated automatically for managing personalization keys. When a personalization parameter value is changed, the call to `save` automatically propagates the change to the server.

An operation can have a parameter that is one of the Sybase Unwired Platform list types (such as `IntList`, `StringList`, or `ObjectList`). This code shows how to set a personalization key, and pass an array of values and an array of objects:

```
PersonalizationParameters pp =
SUP101DB.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.

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.

Note: The `loginToSync` API is now deprecated. Call `synchronize` or `beginSynchronize` before saving synchronization parameters. After saving the synchronization parameters, call `synchronize` or `beginSynchronize` again to retrieve the new values filtered by those parameters.

See also

- *Synchronizing Applications* on page 36
- *Specifying Synchronization Parameters* on page 39

Changing Synchronization Parameters

Synchronization parameters let an application change the parameters that retrieve data from an MBO during a synchronization session.

Note: This topic is not applicable for DOE-based applications.

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.

When a synchronization parameter value is changed, the call to `save` automatically propagates the change to the Unwired Server.

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

Note: The Sybase Unwired Platform server will not send MBO data to a device if an MBO has synchronization parameters defined, unless the application client code calls the `save` method. The next synchronize call will retrieve data from the server. This is true even if default values are defined for its synchronization parameters.

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.

This code synchronizes an MBO package using a specified connection:

```
SUP101DB.synchronize (string synchronizationGroup)
```

The package database class includes two synchronization methods. You can synchronize a specified group of MBOs using the synchronization group name:

```
SUP101DB.synchronize("my-sync-group");
```

Or, you can synchronize all synchronization groups:

```
SUP101DB.synchronize();
```

There is a default synchronization group within every package. The default synchronization group includes all MBOs except those already included by other synchronization groups. To synchronize a default synchronization group call `SUP101DB.beginSynchronize("default");` or `SUP101DB.synchronize("default");`

If there is no other synchronization group, call `SUP101DB.beginSynchronize();` or `SUP101DB.synchronize();`

To synchronize a synchronization group asynchronously:

```
GenericList<SynchronizationGroup> syncGroups = new
GenericList<SynchronizationGroup>();
syncGroups.add(SUP101DB.getSynchronizationGroup("my-sync-group"));
SUP101DB.beginSynchronize(syncGroups, "");
```

When an application uses a create, update, or delete operation in an MBO and calls the `submitPending` metod, an `OperationReplay` object is created for that change. The application must invoke either the `synchronize` or `beginSynchronize` method to upload the `OperationReplay` object to the server to replay the change on the backend data source. The `isReplayQueueEmpty` API is used to check if there are unsent operation replay objects and decide whether a synchronize call is needed.

```
if (!SUP101DB.isReplayQueueEmpty())
{
   // There are OperationReplay not uploaded to server
   GenericList<SynchronizationGroup> sgs = new
GenericList<SynchronizationGroup>();
   sgs.add(SUP101DB.getSynchronizationGroup("system"));
   SUP101DB.beginSynchronize(sgs, "upload OperationReplay objects");
}
```

Message-Based Synchronization APIs

The message-based synchronization APIs enable a user application to subscribe to a server package, to remove an existing subscription from the Unwired Server, to suspend or resume requests to the Unwired Server, and to recover data related to the package from the server.

Note: The message-based synchronization APIs are for use with DOE-based applications only.

beginOnlineLogin

Sends a login message to the Unwired Server with the username and password.

Typically, the generated package database class already has a valid synchronization connection profile and you can log in to the Unwired Server with your username and credentials.

beginOnlineLogin sends a message to the Unwired Server with the username and password. The Unwired Server responds with a message to the client with the login success or failure.

Make sure the connection is active before calling `beginOnlineLogin`, otherwise an exception may be thrown.

When the login succeeds, the `onLoginSuccess` method of the CallbackHandler is invoked. When the login fails, the `onLoginFailure` method of the CallbackHandler is invoked.

Syntax

```
public static void beginOnlineLogin(String userName, String
password)
```

Parameters

- **userName –** the user name.
- **password –** the password.

Returns

None.

Examples

- **Begin an Online Login –** Start logging in with "supAdminID" for the user name and "supPass" for the password.

  ```
  SUP101DB.beginOnlineLogin("supAdminID", "supPwd");
  ```

subscribe

Subscribes to a server package. A subscription message is sent to the Unwired Server and the application receives a subscription request result notification from the Unwired Server. If the subscription succeeds, the `onSubscribeSuccess` method of the ICallbackHandler is invoked. If the subscription fails, the `onSubscribeFailure` method of the ICallbackHandler is invoked.

Prerequisites for using **subscribe**:

- The mobile application is compiled with the client framework and deployed to a mobile device, together with the Sybase Unwired Platform client process.
- The device application has already configured Unwired Server connection information.
- Authentication credentials must also be set, using either the **beginOnlineLogin** or **offlineLogin** APIs.

Syntax

```
public static void subscribe()
```

Parameters

- **None – subscribe** has no parameters.

Returns

None.

Examples

- **Subscribe to a Sample Application –** Subscribe to SUP101DB.

  ```
  SUP101DB.subscribe();
  ```

unsubscribe

Removes an existing subscription to a server package. An unsubscription message is sent to the Unwired Server and the application receives a subscription request result notification from the Unwired Server as a notification. The data on the local database is cleaned. If the unsubscribe succeeds, the `onSubscribeSuccess` method of the CallbackHandler is invoked. If it fails, the `onSubscribeFailure` method of the CallbackHandler is invoked.

The device application must already have a subscription with the server.

Syntax

```
public static void unsubscribe()
```

Parameters

- **None – unsubscribe** has no parameters.

Returns

None.

Examples

- **Unsubscribe from a Sample Application –** Unsubscribe from SUP101DB.

  ```
  SUP101DB.unsubscribe();
  ```

suspendSubscription

Sends a suspend request to the Unwired Server to notify the server to stop delivering data changes. A suspend subscription message is sent to the Unwired Server and the application receives a suspend subscription request result notification from the Unwired Server as a notification. If the suspend succeeds, the `onSuspendSubscriptionSuccess` method of the CallbackHandler is invoked. If the suspend fails, the `onSuspendSubscriptionFailure` method of the CallbackHandler is invoked.

Syntax

```
public static void suspendScription()
```

Parameters

- **None – suspendSubscription** has no parameters.

Returns

None.

Examples

- **Suspend a Subscription –** Suspend the subscription to SUP101DB.

  ```
  SUP101DB.suspendScription();
  ```

beginSynchronize

Sends a message to the Unwired Server to synchronize data between the client and the server. There are two different beginSynchronize APIs, one with no parameters that synchronizes all the groups, and one that takes a list of groups.

The synchronization completes in the background through an asynchronous message exchange with the server. If application code needs to know when the synchronization is complete, a callback handler that implements the `onSynchronize` method must be registered with the database class.

Syntax

```
public static void
beginSynchronize(GenericList<SynchronizationGroup> sgs,
      Object context)
```

Parameters

- **synchronizationGroups –** specifies a list of a list of `SynchronizationGroup` objects representing the groups to be synchronized. If omitted, begin synchronizing data for all groups.

  Note: This parameter is not relevant for DOE packages; pass a null value to this parameter.
- **context –** a reference string used when the server responds to the synchronization request. For more information on the onSynchronize callback handler method, see *Callback Handlers* in *Developer Guide for Android*.
- **uploadOnly –** If true, the synchronization only uploads data; otherwise, the synchronization both uploads and downloads data.

Returns

None.

Examples

- **Synchronize Data between Client and Server –** Synchronize data for SUP101DB for all synchronization groups.

  ```
  SUP101DB.beginSynchronize(null, "my context");
  ```

resumeSubscription

Sends a resume request to the Unwired Server.

The resume request notifies the Unwired Server to resume sending data changes for the subscription that had been suspended. On success, **onResumeSubscriptionSuccess** callback handler method is called. On failure, **onResumeSubscriptionFailure** callback handler is called.

Syntax

```
public static void resumeSubscription()
```

Parameters

- **None – resumeSubscription** has no parameters.

Returns

None.

Examples

- **Resume a Subscription –** Resume the subscription to SUP101DB.

  ```
  SUP101DB.resumeScription();
  ```

Push Synchronization Applications

Clients receive device notifications when a data change is detected for any of the MBOs in the synchronization group to which they are subscribed.

Sybase Unwired Platform uses a messaging channel to send change notifications from the server to the client device. By default, change notification is disabled. You can enable the change notification of a synchronization group:

```
SynchronizationGroup sg = 
SUP101DB.getSynchronizationGroup("TCNEnabled");

if (!sg.getEnableSIS())
{
  sg.setEnableSIS(true);
  sg.setInterval(2); // 2 minutes
  sg.save();
  SUP101DB.synchronize("TCNEnabled");
}
```

When the server detects changes in an MBO affecting a client device, and the synchronization group of the MBO has change detection enabled, the server will send a notification to client device through messaging channel. By default, a background synchronization downloads the changes for that synchronization group. The application can implement the onSynchronize callback method to monitor this condition, and either allow or disallow background synchronization.

```
public int onSynchronize(GenericList<SynchronizationGroup> groups,
SynchronizationContext context)
{
  int status = context.getStatus();
  if (status == SynchronizationStatus.STARTING_ON_NOTIFICATION)
  {
    // There is changes on the synchronization group
    if (busy)
    {
      return SynchronizationAction.CANCEL;
    }
    else
    {
      return SynchronizationAction.CONTINUE;
    }
  }

  // return CONTINUE for all other status
  return SynchronizationAction.CONTINUE;
}
```

Retrieving Information about Synchronization Groups

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

```
/// Determines if the synchronization group was synchronized
public static boolean isSynchronized(java.lang.String
synchronizationGroup)

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

Log Record APIs

The Log Record APIs allow you to customize aspects of logging.

- Writing and retrieving log records (successful operations are not logged).
- Configuring log levels for messages reported to the console.
- Enabling the printing of server message headers and message contents, database exceptions, and `LogRecord` objects written for each import.

- Viewing detailed trace information on database calls.
- The change log can be enabled or disabled with the `enableChangeLog` and `disableChangeLog` methods. You can retrieve the change log by calling the `getChangeLogs` method.

LogRecord API

`LogRecord` stores 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.

This 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();
SUP101DB.synchronize();
result = Customer.findById(id);
for(com.sybase.persistence.LogRecord logRecord :
result.getLogRecords())
{
//Working with logRecord
}
```

The code in the log record is an HTTP status code. See *Developer Guide: Android Object API Applications > Client Object API Usage > Exceptions > Handling Exceptions > HTTP Error Codes*.

There is no logRecord generated for a successful operation replay. The Unwired Server only creates a logRecord when an operation fails.

This sample code shows how to find the corresponding MBO with the LogRecord and to delete the log record when a record is processed.

```
private void processLogs()
{
  Query query = new Query();
  GenericList<LogRecord> logRecords = SUP101DB.getLogRecords(query);
  boolean callSync = false;
  for(LogRecord log : logRecords)
  {
    // log warning message
    Log.warning("log " + log.getComponent() + ":" +
log.getEntityKey()
      + " code:" + log.getCode()
      + " msg:" + log.getMessage());

    if (log.getComponent().equals("Customer"))
    {
      long surrogateKey = Long.parseLong(log.getEntityKey());
      Customer c = Customer.find(surrogateKey);
      if (c.isPending())
      {
    c.cancelPending();
      }

    // delete the LogRecord after it is processed
    log.delete();
    log.submitPending();
    callSync = true;
  }
}
```

Logger APIs

Use the `Logger` API to set the log level and create log records on the client.

Each package has a `Logger`. To obtain the package logger, use the `getLogger` method in the generated database class.

```
Logger logger = SUP101DB.getLogger();

// set log level to debug
logger.setLogLevel(LogLevel.DEBUG);

// create a log record with ERROR level and the error message.
logger.error("Some error message");
```

Change Log API

The change log allows a client to retrieve entity changes from the back end. If a client application already has a list view constructed, it simply needs to add, modify, or delete entries in the list according to the change logs.

A single `ChangeLog` is generated for each changed entity. If the changed entity is a child of a composite relationship, there is also a `ChangeLog` for its parent root entity.

getEntityType

Returns the entity type.

Syntax

```
public int getEntityType()
```

Parameters

None.

Returns

Returns the entity type. The entity type values are defined in the generated java class EntityType.java for the package.

Examples

- **Get the Entity Type –**
  ```
  getEntityType()
  ```

getOperationType

Returns the operation type of the MBO.

Syntax

```
public char getOperationType()
```

Parameters

None.

Returns

The operation type of the MBO. Possible values are 'U' for update and insert, and 'D' for delete.

Examples

- **Get the Entity Type –**
  ```
  getOperationType()
  ```

getRootEntityType

Returns the name of the root parent entity type

Syntax

```
public int getRootEntityType()
```

Parameters

None.

Returns

Returns the root entity type which is the root of the object graph. The entity type values are defined in the generated java class `EntityType.java` for the package.

Examples

- **Get the Entity Type –**
  ```
  getRootEntityType()
  ```

getRootSurrogateKey

Returns the surrogate key of the root parent entity.

Syntax

```
public long getRootSurrogateKey()
```

Parameters

None.

Returns

The surrogateKey of the root entity.

Examples

- **Get the Root Surrogate Key –**
  ```
  getRootSurrogateKey()
  ```

getSurrogateKey

Returns the surrogate key of the entity.

Syntax

```
public long getSurrogateKey()
```

Parameters

None.

Returns

The surrogate key of the affected entity. Note that the change log contains all affected entities, including children of the object graph.

Examples

- **Get the Surrogate Key –**
  ```
  getSurrogateKey()
  ```

Methods in the Generated Database Class

You can use generated methods in the package database class to manage change logs.

enableChangeLog

By default, Change Log is disabled. To enable the change log, invoke the `enableChangeLog` API in the generated database class. The next synchronization will have change logs sent to the client.

Syntax

```
enableChangeLog();
```

Returns

None.

Examples

- **Enable Change Log –**

  ```
  SUP101DB.enableChangeLog();
  ```

getChangeLogs

Retrieve a list of change logs.

Syntax

```
GenericList<com.sybase.persistence.ChangeLog>
getChangeLogs(com.sybase.persistence.Query query);
```

Returns

Returns a GenericList of type <Change Log>.

Examples

- **Get Change Logs –**

  ```
  GenericList<com.sybase.persistence.ChangeLog> clog =
  SUP101DB.getChangeLogs(query);
  ```

deleteChangeLogs

You are recommended to delete all change logs after the application has completed processing them. Use the `deleteChangeLogs` API in the generated database class to delete all change logs on the device.

Syntax

```
deleteChangeLogs();
```

Returns

None.

Examples

- **Delete Change Logs –**
  ```
  SUP101DB.deleteChangeLogs();
  ```

Usage

Ensure that when calling `deleteChangeLogs()`, there are no change logs created from a background synchronization that are not part of the original change log list returned by a specific query:

```
GenericList<ChangeLog> changes = getChangeLogs(myQuery);
```

You should only call `deleteChangeLogs()` in the `onSynchronize()` callback where there are no multiple synchronizations occurring simulatenously.

disableChangeLog

Creating change logs consumes some processing time, which can impact application performance. The application may can disable the change log using the `disableChangeLog` API.

Syntax

```
disableChangeLog();
```

Returns

None.

Examples

- **Disable Change Log –**
  ```
  SUP101DB.disableChangeLog();
  ```

Code Samples

Enable the change log and list all changes, or only the change logs for a particular entity, Customer.

```
SUP101DB.enableChangeLog();
SUP101DB.synchronize();

// Retrieve all change logs
GenericList<ChangeLog> logs = SUP101DB.getChangeLogs(new Query());
System.out.println("There are " + logs.size() + " change logs");
for (ChangeLog log : logs)
```

```
{
  System.out.println(log.getEntityType()
    + "(" + log.getSurrogateKey()
    + "): " + log.getOperationType());
}

// Retrieve only the change logs for Customer:
Query query = new Query();
AttributeTest at = new AttributeTest("entityType",
               new java.lang.Integer(SUP101.EntityType.Customer),
               AttributeTest.EQUAL);
query.setTestCriteria(at);
logs = SUP101DB.getChangeLogs(query);
System.out.println("There are " + logs.size() + " change logs for
Customer");
for (ChangeLog log : logs)
{
  System.out.println(log.getEntityType()
    + "(" + log.getSurrogateKey()
    + "): " + log.getOperationType());
}
```

Security APIs

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

Encrypt the Database

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

The length of the encyption key cannot be fewer than 16 characters.

```
ConnectionProfile profile = SUP101DB.getConnectionProfile();
profile.setEncryptionKey("Your key of length 16 or more
characters");
```

You can use the `generateEncryptionKey()` method to encrypt the local database with a random encryption key.

```
SUP101DB.generateEncryptionKey();
// store the encryption key at somewhere for reuse later
ConnectionProfile profile = SUP101DB.getConnectionProfile();
String key = profile.getEncryptionKey();
...
SUP101DB.createDatabase();
```

End to End Encryption and Compression Support APIs

Use encryption communication parameters to ensure end to end encryption and eliminate any WAP gap security problems.

You can use the Client Object API to set up end to end encryption, supported by Ultralite, and HTTPS items in the synchronization profile.

Refer to the following APIs when setting up end to end encryption and compression support:

- `com.sybase.persistence.ConnectionProfile.getStreamParams`
- `com.sybase.persistence.NetworkStreamParams.getTrusted_Certificates`
- `com.sybase.persistence.NetworkStreamParams.setTrusted_Certificates`
- `com.sybase.persistence.NetworkStreamParams.getE2ee_Type`
- `com.sybase.persistence.NetworkStreamParams.setE2ee_Type`
- `com.sybase.persistence.NetworkStreamParams.getE2ee_Public_Key`
- `com.sybase.persistence.NetworkStreamParams.setE2ee_Public_Key`
- `com.sybase.persistence.NetworkStreamParams.setZlibCompression`
- `com.sybase.persistence.NetworkStreamParams.setZlib_Upload_Window_Size`
- `com.sybase.persistence.NetworkStreamParams.setZlib_Download_Window_Size`
- `com.sybase.persistence.NetworkStreamParams.getZlibCompression`
- `com.sybase.persistence.NetworkStreamParams.getZlib_Upload_Window_Size`
- `com.sybase.persistence.NetworkStreamParams.getZlib_Download_Window_Size`

The following code example shows how to set SUP101:

```
ConnectionProfile cp=SUP101DB.getSynchronizationProfile();
cp.setNetworkProtocol("HTTP");
cp.setPortNumber(2480);
cp.getStreamParams().setE2ee_Type("RSA");
cp.getStreamParams().setE2ee_Public_Key(sdcard directory
+ApplicationName+"_e2eeKey.key");
cp.getStreamParams().setZlibCompression(true);
cp.getStreamParams().setZlib_Upload_Window_Size(12);
cp.getStreamParams().setZlib_Download_Window_Size(12);
cp.setUserName(userName);
cp.setPassword(password);
cp.save();
SUP101DB.synchronize();
```

DataVault

The `DataVault` class provides encrypted storage of occasionally used, small pieces of data. All exceptions thrown by `DataVault` methods are of type `DataVaultException`.

If you have installed the `SybaseDataProvider.apk` package, you can use the `DataVault` class for on-device persistent storage of certificates, database encryption keys, passwords, and other sensitive items. Use this class to:

- Create a vault
- Set a vault's properties
- Store objects in a vault
- Retrieve objects from a vault
- Change the password used to access a vault

The contents of the data vault are strongly encrypted using AES-256. The `DataVault` class allows you create a named vault, and specify a password and salt used to unlock it. The password can be of arbitrarily length and can include any characters. The password and salt together are used to generate the AES key. If the user enters the same password when unlocking, the contents are decrypted. If the user enters an incorrect password, exceptions will occur. If the user enters the incorrect password a configurable number of times, the vault is deleted and any data stored within it becomes unrecoverable. The vault can also re-lock itself after a configurable amount of time.

Typical usage of the `DataVault` would be to implement an application login screen. Upon application start, the user is prompted for a password, which is then used to unlock the vault. If the unlock attempt is successful, the user is allowed into the rest of the application. User credentials needed for synchronization can also be extracted from the vault so the user is not repeatedly prompted to re-enter passwords.

To install `SybaseDataProvider.apk` on an Android device:

1. Connect the Android device to your computer.
2. Open the command line directory to the `adb.exe` file, for example, `C:\Program Files\android-sdk-windows\tools`, or `C:\Program Files\android-sdk-windows\platform-tools`.
3. Run the command **adb.exe install** `<UnwiredPlatform_InstallDir>\UnwiredPlatform\MobileSDK\ObjectAPI\Android\SybaseDataProvider.apk`

createVault

Creates a new secure store.

Creates a vault. A unique name is assigned, and after creation, the vault is referenced and accessed by that name. This method also assigns a password and salt value to the vault. If a vault already exists with the same name, this method throws an exception. When created, the vault is in the unlocked state.

Syntax

```
public static DataVault createVault(
   String name,
   String password,
   String salt
)
```

Parameters

- **name –** The vault name.
- **password –** The password.
- **salt –** The encryption salt value.

Returns

createVault creates a `DataVault` instance.

If a vault already exists with the same name, a `DataVaultException` is thrown this with the reason `ALREADY_EXISTS`.

Examples

- **Create a Data Vault –** Creates a new data vault called `myVault`.

  ```
  DataVault vault = null;
  if (!DataVault.vaultExists("myVault"))
  {
     vault = DataVault.createVault("myVault", "password", "salt");
  }
  else
  {
     vault = DataVault.getVault("myVault");
  }
  ```

vaultExists

Tests whether the specified vault exists.

Syntax

```
public static boolean vaultExists(String name)
```

Parameters

- **name –** The vault name.

Returns

vaultExists can return the following values:

Returns	Indicates
true	The vault exists.
false	The vault does not exist.

Examples

- **Check if a Data Vault Exists –** Checks if a data vault called `myVault` exists, and if so, deletes it.

```
if (DataVault.vaultExists("myVault"))
{
   DataVault.deleteVault("myVault");
}
```

getVault

Retrieves a vault.

Syntax

```
public static DataVault getVault(String name)
```

Parameters

- **–** The vault name.

Returns

getVault returns a `DataVault` instance.

If the vault does not exist, a `DataVaultException` is thrown.

deleteVault

Deletes the specified vault from on-device storage.

Deletes a vault having the specified name. If the vault does not exist, this method throws an exception. The vault need not be in the unlocked state, and can be deleted even if the password is unknown.

Syntax

```
public static void deleteVault(String name)
```

Parameters

- **name –** The vault name.

Examples

- **Delete a Data Vault –** Deletes a data vault called `myVault`.

```
if (DataVault.vaultExists("myVault"))
{
   DataVault.deleteVault("myVault");
}
```

lock

Locks the vault.

Once a vault is locked, you must unlock it before changing the vault's properties or storing anything in it. If the vault is already locked, this method has no effect.

Syntax

```
public void lock()
```

Examples

- **Locks the data vault. –** Prevents changing the vaults properties or stored content.

```
vault.lock();
```

isLocked

Tests whether the vault is locked.

Syntax

```
public boolean isLocked()
```

Returns

isLocked can return the following values:

Returns	Indicates
true	The vault is locked.
false	The vault is unlocked.

unlock

Unlocks the vault.

Unlock the vault before changing the its properties or storing anything in it. If the incorrect password or salt is used, this method throws an exception. If the number of unsuccessful unlock attempts exceeds the retry limit, the vault is deleted.

Syntax

```
public void unlock(String password, String salt)
```

Parameters

- **password –** The password.
- **salt –** The encryption salt value.

Returns

If the incorrect password or salt is used, a `DataVaultException` is thrown this with the reason `INVALID_PASSWORD`.

Examples

- **Unlocks the data vault. –** Once the vault is unlocked you can change the its properties and stored content.

  ```
  if (vault.isLocked())
  {
     vault.unlock("password", "salt");
  }
  ```

setLockTimeout

Determines how long a vault remains unlocked.

Determines how many seconds a vault remains unlocked before it automatically locks. The default value, 0, indicates that the lock never times out.

Syntax

```
public void setLockTimeout(int timeout)
```

Parameters

- **–** The number of seconds before the lock times out.

Examples

- **Set the Lock Timeout –** Sets the lock timeout to 1 hour.

  ```
  vault.setLockTimeout( 3600 );
  ```

getLockTimeout

Retrieves the configured lock timeout period.

Retrieves the number of seconds a vault remains unlocked before it automatically locks. The default value, 0, indicates that the lock never times out.

Syntax

```
public int getLockTimeout()
```

Returns

getLockTimeout returns an integer value indicating the number of seconds a vault remains unlocked before it automatically locks. The default value, 0, indicates that the lock never times out.

Examples

- **Set the Lock Timeout –** Retrieves the lock timeout in seconds.

  ```
  int timeout = vault.getLockTimeout();
  ```

setRetryLimit

Sets the retry limit value for the vault.

Determines how many consecutive unlock attempts (with wrong password) are allowed. If the retry limit is exceeded, the vault is automatically deleted. The default value, 0, means that an unlimited number of attempts are permitted. An exception is thrown if the vault is locked when this method is called.

Syntax

```
public void setRetryLimit(int limit)
```

Parameters

- **limit –** The number of consecutive unlock attempts (with wrong password) are allowed.

Examples

- **Set the Retry Limit –** Sets the retry limit to 5 attempts.

  ```
  vault.setRetryLimit( 5 );
  ```

getRetryLimit

Retrieves the retry limit value for the vault.

Retrieves the number of consecutive unlock attempts (with wrong password) are allowed. If the retry limit is exceeded, the vault is automatically deleted. The default value, 0, means that an unlimited number of attempts are permitted.

Syntax

```
public int getRetryLimit()
```

Returns

getRetryLimit returns an integer value indicating the number of consecutive unlock attempts (with wrong password) are allowed. If the retry limit is exceeded, the vault is automatically deleted. The default value, 0, means that an unlimited number of attempts are permitted.

Examples

- **Set the Retry Limit –** Retrieves the number of consecutive unlock attempts (with wrong password) that are allowed.

  ```
  int retrylimit = vault.getRetryLimit();
  ```

setString

Stores a string object in the vault.

Stores a string under the specified name. An exception is thrown if the vault is locked when this method is called.

Syntax

Parameters

- **name –** The name associated with the string object to be stored.
- **value –** The string object to store in the vault.

Examples

- **Set a String Value –** Creates a test string, unlocks the vault, and sets a string value associated with the name `"testString"` in the vault. The `finally` clause in the `try/catch` block ensure that the vault ends in a secure state even if an exception occurs.

  ```
  string teststring = "ABCDEFabcdef";
  try
  {
     vault.unlock("password", "salt");
     vault.setString("testString", teststring);
  }
  catch (DataVaultException e)
  {
     System.out.println("Exception: " + e.toString());
  }
  finally
  {
     vault.lock();
  }
  ```

getString

Retrieves a string value from the vault.

Retrieves a string stored under the specified name in the vault. An exception is thrown if the vault is locked when this method is called.

Syntax

```
public String getString(String name)
```

Parameters

- **name –** The name associated with the string object to be retrieved.

Returns

getString returns a string data value, associated with the specified name, from the vault. An exception is thrown if the vault is locked when this method is called.

Examples

- **Get a String Value –** Unlocks the vault and retrieves a string value associated with the name `"testString"` in the vault. The `finally` clause in the `try/catch` block ensure that the vault ends in a secure state even if an exception occurs.

```
try
{
   vault.unlock("password", "salt");
   string retrievedstring = vault.getString("testString");
}
catch (DataVaultException e)
{
   System.out.println("Exception: " + e.toString());
}
finally
{
   vault.lock();
}
```

setValue

Stores a binary object in the vault.

Stores a binary object under the specified name. An exception is thrown if the vault is locked when this method is called.

Syntax

```
public void setValue(
   string name,
   byte[] value
)
```

Parameters

- **name –** The name associated with the binary object to be stored.
- **value –** The binary object to store in the vault.

Examples

- **Set a Binary Value –** Unlocks the vault and stores a binary value associated with the name `"testValue"` in the vault. The `finally` clause in the `try/catch` block ensure that the vault ends in a secure state even if an exception occurs.

  ```
  try
  {
     vault.unlock("password", "salt");
     vault.setValue("testValue", new byte[] { 1, 2, 3, 4, 5});
  }
  catch (DataVaultException e)
  {
     System.out.println("Exception: " + e.toString());
  }
  finally
  {
     vault.lock();
  }
  ```

getValue

Retrieves a binary object from the vault.

Retrieves a binary object under the specified name. An exception is thrown if the vault is locked when this method is called.

Syntax

```
public byte[] getValue(string name)
```

Parameters

- **name –** The name associated with the binary object to be retrieved.

Returns

getValue returns a binary data value, associated with the specified name, from the vault. An exception is thrown if the vault is locked when this method is called.

Examples

- **Get a Binary Value –** Unlocks the vault and retrieves a binary value associated with the name `"testValue"` in the vault. The `finally` clause in the `try/catch` block ensure that the vault ends in a secure state even if an exception occurs.

```
try
{
   vault.unlock("password", "salt");
   byte[] retrievedvalue = vault.getValue("testValue");
}
catch (DataVaultException e)
{
   System.out.println("Exception: " + e.toString());
}
finally
{
   vault.lock();
}
```

changePassword

Changes the password for the vault.

Modifies all name/value pairs in the vault to be encrypted with a new password/salt. If the vault is locked or the new password is empty, an exception is thrown.

Syntax

Parameters

- **newPassword –** The new password.
- **newSalt –** The new encryption salt value.

Examples

- **Change the Password for a Data Vault –** Changes the password to `"newPassword"`. The `finally` clause in the `try/catch` block ensure that the vault ends in a secure state even if an exception occurs.

```
try
{
   vault.unlock("password", "salt");
   vault.changePassword("newPassword", "newSalt");
}
catch (DataVaultException e)
{
   System.out.println("Exception: " + e.toString());
}
finally
{
   vault.lock();
}
```

Callback and Listener APIs

The callback and listener APIs allow you to optionally register a callback handler and listen for device events, application connection events, and package synchronize and replay events.

See also

- *Setting Up Callbacks* on page 31

CallbackHandler API

The CallbackHandler interface is invoked when any database event occurs. A default callback handler is provided, which basically does nothing. You should implement a custom `CallbackHandler` to register important events. The callback is invoked on the thread that is processing the event. To receive callbacks for database changes, 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 or by implementing the `com.sybase.persistence.CallbackHandler` interface.

In your handler, override the particular callback that you are interested in (for example, `void onReplayFailure(java.lang.Object entity)`). 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.

Table 1. Callbacks in the `CallbackHandler` Interface

Callback	Description
`void onImport(java.lang.Object entity)`	This method is invoked when an import message is successfully applied to the local database. However, it is not committed. One message from server may have multiple import entities and they would be committed in one transaction for the whole message. **Note:** **1.** Stale data may be read from the database at this time before commit of the whole message. Developers are encouraged to wait until the next `onTransactionCommit()` is invoked, then to read from the database to obtain the updated data. **2.** This method is for DOE-based applications only. **3.** Both CallbackHandlers registered for the MBO class of the entity and Package DB will be invoked. **Parameters**: • **entity** – the Mobile Business Object that was just imported.
`void onLoginFailure()`	This method will be invoked when login failed for a `beginOnlineLogin` call. **Note:** **1.** This method is for DOE-based application only. **2.** Only the `CallbackHandler` registered for package DB will be invoked.

Callback	Description
`void onLoginSuccess()`	This method is invoked when login succeeds for a `beginOnlineLogin` call. **Note:** **1.** Only the CallbackHandler registered for package DB is invoked. **2.** This method is for DOE-based application only.
`void onReplayFai-lure(java.lang.Object entity)`	This method is invoked when a replay request fails. **Note:** CallbackHandlers registered for both the MBO class of the entity and the Package DB are invoked. **Parameters**: • **entity** – the Mobile Business Object to replay.
`void onReplaySuc-cess(java.lang.Object entity)`	This method is invoked when a replay request succeeds. **Note:** CallbackHandlers registered for both the MBO class of the entity and the Package DB are invoked. **Parameters**: • **entity** – the Mobile Business Object to replay.
`void onSearchFai-lure(java.lang.Object entity)`	This method is invoked when a back-end search fails. **Note:** **1.** CallbackHandlers registered for both the MBO class of the entity and the Package DB are invoked. **2.** This method is for DOE-based application only. **Parameters**: • **entity** – the back-end search object.

Callback	Description
`void onSearchSuccess(java.lang.Object entity)`	This method is invoked when a back end search succeeds. **Note:** **1.** CallbackHandlers registered for both the MBO class of the entity and the Package DB are invoked. **2.** This method is for DOE-based application only. **Parameters**: • **entity** – the back-end search object.
`void onSubscribeFailure()`	This method is invoked when subscribe fails. **Note:** **1.** Only the CallbackHandler registered for the Package DB is invoked. **2.** This method is for DOE-based application only.
`void onSubscribeSuccess()`	This method is invoked when subscribe succeeds. **Note:** **1.** Only the CallbackHandler registered for the Package DB is invoked. **2.** This method is for DOE-based application only.

Callback	Description
`int onSynchronize(GenericList<SynchronizationGroup> groups, SynchronizationContext context)`	This method is invoked at different stages of the synchronization. The status of the synchronization context specifies the stage of the synchronization. **Parameters**: • **groups** – a list of synchronization groups. • **context** – the synchronization context. **Returns**: Either `SynchronizationAction.CONTINUE` or `Synchronization.CANCEL`. If `SynchronizationAction.CANCEL` is returned, the synchronize is cancelled if the status of the synchronization context is one of the following. • `SynchronizationStatus.STARTING` • `SynchronizationStatus.ASYNC_REPLAY_COMPLETED` • `SynchronizationStatus.STARTING_ON_NOTIFICATION` The return value has no effect if the status is not in the above list.
`void onSuspendSubscriptionFailure()`	This method is invoked when suspend subscription fails. **Note:** **1.** Only the CallbackHandler registered for the Package DB is invoked. **2.** This method is for DOE-based application only.

Callback	Description
`void onSuspendSubscription-Success()`	This method is invoked when suspend subscrip-tion succeeds. **Note:** **1.** Only the CallbackHandler registered for the Package DB is invoked. **2.** This method is for DOE-based application only.
`void onResumeSubscriptionFai-lure()`	This method is invoked when resume subscrip-tion fails. **Note:** **1.** Only the CallbackHandler registered for the Package DB is invoked. **2.** This method is for DOE-based application only.
`void onResumeSubscriptionSuc-cess()`	This method is invoked when resume subscrip-tion succeeds. **Note:** **1.** Only the CallbackHandler registered for the Package DB is invoked. **2.** This method is for DOE-based application only.
`void onUnsubscribeFailure()`	This method is invoked when unsubscribe fails. **Note:** **1.** Only the CallbackHandler registered for the Package DB is invoked. **2.** This method is for DOE-based application only.

Callback	Description
`void onUnsubscribeSuccess()`	This method is invoked when unsubscribe succeeds. **Note:** **1.** Only the CallbackHandler registered for the Package DB is invoked. **2.** This method is for DOE-based application only.
`void onMessageException(java.lang.Exception ex)`	This method is invoked when an exception occurs in the processing of a message. **Note:** In DefaultCallbackHandlers, `onMessageException` re-throws the Exception so that the messaging layer can retry the message. The application developer has the option to implement a custom CallbackHandler that does not re-throw the exception, based on exception types or other conditions, so that the message is not retried. **Parameters**: • **ex** – the exception thrown when processing a message.
`void onTransactionCommit()`	This method is invoked after a message is processed and committed. **Note:** **1.** Only the CallbackHandler registered for the Package DB is invoked. **2.** This method is for DOE-based application only.

Callback	Description
`void onTransactionRollback()`	This method is invoked after a message is rolled back. It only happens when an Exception was thrown when processing the message, or from a custom Callback method. **Note:** **1.** Only the CallbackHandler registered for the Package DB is invoked. **2.** This method is for DOE-based application only.
`void onResetSuccess()`	This method is invoked when all data is cleared by the reset. **Note:** **1.** Only the CallbackHandler registered for the Package DB is invoked. **2.** This method is for DOE-based application only.
`void onRecoverSuccess()`	This method is invoked when recover succeeds. **Note:** **1.** Only the CallbackHandler registered for the Package DB is invoked. **2.** This method is for DOE-based application only.
`void onRecoverFailure()`	This method is invoked when recover fails. **Note:** **1.** Only the CallbackHandler registered for the Package DB is invoked. **2.** This method is for DOE-based application only.

Callback	Description
`void onSubscriptionEnd()`	This method is invoked when a subscription is re-registered or unsubscribed. This method deletes all MBO data on the device. **Note:** **1.** Only the CallbackHandler registered for the Package DB is invoked. **2.** This method is for DOE-based application only.
`void onImportSuccess()`	This method is invoked when all data has been successfully imported. **Note:** **1.** Only the CallbackHandler registered for the Package DB is invoked. **2.** This method is for DOE-based application only.
`void beforeImport(java.lang.Object entity)`	This method is invoked before importing the specified entity. **Note:** **1.** Only the CallbackHandler registered for the Package DB is invoked. **2.** This method is for DOE-based application only. **Parameters**: • **entity** – the Mobile Business Object to be imported.

This code 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);
```

ApplicationCallback API

This callback interface is invoked by events of interest to a mobile application.

You must register an `ApplicationCallback` implementation to your com.sybase.mobile.Application instance to receive these callbacks.

Table 2. Callbacks in the ApplicationCallback Interface

Callback	Description
`void onApplicationSettingsChanged(StringList nameList)`	Invoked when one or more application settings have been changed by the server administration.
`void onConnectionStatusChanged(int connectionStatus, int errorCode, String errorMessage)`	Invoked when the connection status changes. The possible connection status values are defined in the `ConnectionStatus` class.
`void onDeviceConditionChanged(int condition)`	Invoked when a condition is detected on the mobile device that may be of interest to the application or the application user. The possible device condition values are defined in the `DeviceCondition` class.
`void onRegistrationStatusChanged(int registrationStatus, int errorCode, String errorMessage)`	Invoked when the registration status changes. The possible registration status values are defined in the `RegistrationStatus` class.

SyncStatusListener API

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

Note: This topic is not applicable for DOE-based applications.

Create a listener that implements the `SyncStatusListener` interface.

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

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

Pass an instance of the listener to the synchronize methods.

```
SyncStatusListener listener = new MySyncListener();
SUP101DB.synchronize("sync_group", listener);
```

```
// or SUP101DB.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 other than the primary user interface thread, the client cannot update its screen as the status changes. The client must instruct the primary user interface thread to update the screen regarding the current synchronization status.

This is an example of `SyncStatusListener` implementation:

```
public class SyncListener implements 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;
  }
}
```

Query APIs

The Query API allows you to retrieve data from mobile business objects, to page data, and to retrieve a query result by filtering. You can also use the Query API to filter children MBOs of a parent MBO in a one to many relationship.

See also

- *Accessing MBO Data* on page 40
- *Object Queries* on page 41
- *Dynamic Queries* on page 41
- *MBOs with Complex Types* on page 42
- *Relationships* on page 43

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 Queries

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

Object Query methods are generated based on the object queries defined by the modeler in Unwired WorkSpace. Object Query methods carry query names, parameters, and return types defined in Unwired WorkSpace. Object Query methods return either an object, or a collection of objects that match the specified search criteria.

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

This method retrieves all customers:

```
public static com.sybase.collections.GenericList<Customer> findAll()

com.sybase.collections.GenericList<Customer> customers =
Customer.findAll();
```

This method retrieves all customers in a certain page:

```
public static com.sybase.collections.GenericList<Customer>
findAll(int skip, int take)

com.sybase.collections.GenericList<Customer> customers =
Customer.findAll(10, 5);
```

Suppose the modeler defined the following Object Query for the Customer MBO in Sybase Unwired Workspace:

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

The preceding Object Query results in this generated method:

```
public static com.sybase.collections.GenericList<Customer>
findByFirstName(String firstName)

com.sybase.collections.GenericList<Customer> customers =
Customer.findByFirstName("fname");
```

Query and Related Classes

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

Table 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.
SelectItem	Defines the entry of a select query. For example, "select x.attr1 from MBO x", where "X.attr1" represents one SelectItem.
Column	Used in a subquery to reference the outer query's attribute.

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

Arbitrary Find

The arbitrary find method lets custom device applications dynamically build queries based on user input. The `Query.DISTINCT` property lets you exclude duplicate entries from the result set.

The arbitrary find method also lets the user specify a desired ordering of the results and object state criteria. A `Query` class is included in the client object API. 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.

Set the `Query.Distinct` property to `true` to exclude duplicate entries from the result set. The default value is `false` for entity types, and its usage is optional for all other types.

```
Query query1 = new Query();
query1.setDistinct(true);
```

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

TestCriteria

You can construct a query SQL statement to query data from a local database. You can create a `TestCriteria` object (in this example, `AttributeTest`) to filter results. You can also query across multiple tables (MBOs) when using the `executeQuery` API.

```
Query query2 = new Query();
query2.select("c.fname,c.lname,s.order_date,s.region");
query2.from("Customer", "c");
//
// Convenience method for adding a join to the query
// Detailed construction of the join criteria
query2.join("Sales_order", "s", "c.id", "s.cust_id");
AttributeTest ts = new AttributeTest();
ts.setAttribute("fname");
ts.setTestValue("Beth");
query2.where(ts);
QueryResultSet qrs = SUP101DB.executeQuery(query2);
```

Note: You must use explicit column names in **select** clauses; you cannot use wildcards.

AttributeTest

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

- IS_NULL
- NOT_NULL
- EQUAL
- NOT_EQUAL
- LIKE
- NOT_LIKE
- LESS_THAN

- LESS_EQUAL
- GREATER_THAN
- GREATER_EQUAL
- CONTAINS
- STARTS_WITH
- ENDS_WITH
- DOES_NOT_START_WITH
- DOES_NOT_END_WITH
- DOES_NOT_CONTAIN
- IN
- NOT_IN
- EXISTS
- NOT_EXISTS

For example, the Java code shown below is equivalent to this SQL query:

```
SELECT * from A where id in [1,2,3]
```

```
Query query = new Query();
AttributeTest test = new AttributeTest();
test.setAttribute("id");
com.sybase.collections.ObjectList v = new
com.sybase.collections.ObjectList();
v.add("1");
v.add("2");
v.add("3");
test.setValue(v);
test.setOperator(AttributeTest.IN);
query.where(test);
```

When using EXISTS and NOT_EXISTS, the attribute name is not required in the `AttributeTest`. The query can reference an attribute value via its alias in the outer scope. The Java code shown below is equivalent to this SQL query:

```
SELECT a.id from AllType a where exists (select b.id from AllType b
where b.id = a.id)
```

```
Query query = new Query();
query.select("a.id");
query.from("AllType", "a");
AttributeTest test = new AttributeTest();

Query existQuery = new Query();
existQuery.select("b.id");
existQuery.from("AllType", "b");
Column cl = new Column();
cl.setAlias("a");
cl.setAttribute("id");
AttributeTest test1 = new AttributeTest();
test1.setAttribute ("b.id");
test1.setValue(cl);
test1.setOperator(AttributeTest.EQUAL);
```

```
existQuery.where(test1);
test.setValue(existQuery);
test.setOperator(AttributeTest.EXISTS);
query.where(test);
QueryResultSet qs = SUP101DB.executeQuery(query);
```

SortCriteria

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

For example,

```
Query query = new Query();

query.select("c.lname, c.fname");
query.from("Customer", "c");

AttributeTest aTest = new AttributeTest();
aTest.setAttribute("state");
aTest.setTestValue("CA");
aTest.setTestType(AttributeTest.EQUAL);
query.setTestCriteria(aTest);

SortCriteria sort = new SortCriteria();
sort.add("lname", SortOrderType.ASCENDING);
sort.add("fname", SortOrderType.ASCENDING);
query.setSortCriteria(sort);
```

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);

GenericList<Customer> customers = Customer.findWithQuery(props);
```

Aggregate Functions

You can use aggregate functions in dynamic queries.

When using the `Query.select(String)` method, you can use any of these aggregate functions:

Aggregate Function	Supported Datatypes
`COUNT`	integer
`MAX`	string, binary, char, byte, short, int, long, integer, decimal, float, double, date, time, dateTime

Aggregate Function	Supported Datatypes
`MIN`	string, binary, char, byte, short, int, long, integer, decimal, float, double, date, time, dateTime
`SUM`	byte, short, int, long, integer, decimal, float, double
`AVG`	byte, short, int, long, integer, decimal, float, double

If you use an unsupported type, a `PersistenceException` is thrown.

```
Query query1 = new Query();
query1.select("MAX(c.id), MIN(c.name) as minName");
```

For iOS, we need a code sample equivalent to this WM sample:

Grouping Results

Apply grouping criteria to your results.

To group your results according to specific attributes, use the `Query.groupBy(String groupByItem)` method. For example, to group your results by ID and name, use:

```
String groupByItem = ("c.id, c.name");
Query query1 = new Query();

//other code for query1

query1.groupBy(groupByItem);
```

Filtering Results

Specify test criteria for group queries.

You can specify how your results are filtered by using the `Query.having(com.sybase.persistence.TestCriteria)` method for queries using `GroupBy`. For example, limit your AllType MBO's results to `c.id` attribute values that are greater than or equal to 0 using:

```
Query query2 = new Query();
query2.select("c.id, SUM(c.id)");
query2.from("AllType", "c");
AttributeTest ts = new AttributeTest();
ts.setAttribute("c.id");
ts.setValue("0");
ts.setOperator(AttributeTest.GREATER_EQUAL);
query2.where(ts);
query2.groupBy("c.id");

AttributeTest ts2 = new AttributeTest();
ts2.setAttribute("c.id");
ts2.setValue("0");
ts2.setOperator(AttributeTest.GREATER_EQUAL);
query2.having(ts2);
```

Concatenating Queries

Concatenate two queries having the same selected items.

The `Query` class methods for concatenating queries are:

- `union(Query)`
- `unionAll(Query)`
- `except(Query)`
- `intersect(Query)`

This example obtains the results from one query except for those results appearing in a second query:

```
Query query1 = new Query();
... ... //other code for query1

Query query2 = new Query();
... ... //other code for query 2

Query query3 = query1.except(query2);
SUP101DB.executeQuery(query3);
```

Subqueries

Execute subqueries using clauses, selected items, and attribute test values.

You can execute subqueries using the `Query.from(Query query, String alias)` method. For example, the Java code shown below is equivalent to this SQL query:

```
SELECT a.id FROM (SELECT b.id FROM AllType b) AS a WHERE a.id = 1
```

Use this Java code:

```
Query query1 = new Query();
query1.select("b.id");
query1.from("AllType", "b");
Query query2 = new Query();
query2.select("a.id");
query2.from(query1, "a");
AttributeTest ts = new AttributeTest();
ts.setAttribute("a.id");
ts.setValue(1);
query2.where(ts);
com.sybase.persistence.QueryResultSet qs =
SUP101DB.executeQuery(query2);
```

You can use a subquery as the selected item of a query. Use the `SelectItem` to set selected items directly. For example, the Java code shown below is equivalent to this SQL query:

```
SELECT (SELECT count(1) FROM AllType c WHERE c.id >= d.id) AS cn, id
FROM AllType d
```

Use this Java code:

```
Query selQuery = new Query();
selQuery.select("count(1)");
selQuery.from("AllType", "c");
AttributeTest ttt = new AttributeTest();
ttt.setAttribute("c.id");
ttt.setOperator(AttributeTest.GREATER_EQUAL);
Column cl = new Column();
cl.setAlias("d");
cl.setAttribute("id");
ttt.setValue(cl);
selQuery.where(ttt);

com.sybase.collections.GenericList<com.sybase.persistence.SelectIte
m> selectItems = new
com.sybase.collections.GenericList<com.sybase.persistence.SelectIte
m>();
SelectItem item = new SelectItem();
item.setQuery(selQuery);
item.setAlias("cn");
selectItems.add(item);
item = new SelectItem();
item.setAttribute("id");
item.setAlias("d");
selectItems.add(item);
Query subQuery2 = new Query();
subQuery2.setSelectItems(selectItems);
subQuery2.from("AllType", "d");
com.sybase.persistence.QueryResultSet qs =
SUP101DB.executeQuery(subQuery2);
```

CompositeTest

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

Complex Example

This example shows the usage of `CompositeTest`, `SortCriteria`, and `Query` to locate all customer objects based on particular 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

```
Query props = new Query();
//define the attribute based conditions
//Users can pass in a string if they know the attribute name. R1
column name = attribute name.
CompositeTest innerCompTest = new CompositeTest();
innerCompTest.setOperator(CompositeTest.OR);
innerCompTest.add(new AttributeTest("state", "CA",
AttributeTest.EQUAL));
innerCompTest.add(new AttributeTest("state", "NY",
AttributeTest.EQUAL));
CompositeTest outerCompTest = new CompositeTest();
outerCompTest.setOperator(CompositeTest.OR);
outerCompTest.add(new AttributeTest("fname", "Jane",
AttributeTest.EQUAL));
outerCompTest.add(new AttributeTest("lname", "Doe",
AttributeTest.EQUAL));
outerCompTest.add(innerCompTest);
//define the ordering
SortCriteria sort = new SortCriteria();

sort.add("fname", SortOrder.ASCENDING);
sort.add("lname", SortOrder.ASCENDING);
//set the Query object
props.setTestCriteria(outerCompTest);
props.setSortCriteria(sort);
props.setSkip(10);
props.setTake(5);
com.sybase.collections.GenericList<Customer> customers2 =
Customer.findWithQuery(props);
```

QueryResultSet

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

The following example shows how to filter a result set and get values by taking data from two mobile business objects, creating a `Query`, filling in the criteria for the query, and filtering the query results:

```
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 at = new AttributeTest();
at.setAttribute("lname");
at.setTestValue("Devlin");
query.setTestCriteria(at);
QueryResultSet qrs = SUP101DB.executeQuery(query);
while(qrs.next())
{
  System.out.print(qrs.getString(1));
  System.out.print(",");
  System.out.println(qrs.getStringByName("c.fname"));

  System.out.print(qrs.getString(2));
  System.out.print(",");
  System.out.println(qrs.getStringByName("c.lname"));

  System.out.print(qrs.getString(3));
  System.out.print(",");
  System.out.println(qrs.getStringByName("s.order_date"));

  System.out.print(qrs.getString(4));
  System.out.print(",");
  System.out.println(qrs.getStringByName("s.region"));
}
```

Retrieving Relationship Data

A relationship between two MBOs allows the parent MBO to access the associated MBO. A bidirectional relationship also allows the child MBO to access the associated parent 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);
GenericList<Sales_order> orders = customer.getSalesOrders();
```

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

```
Query props = new Query();
// set query parameters
......
GenericList<Sales_order> orders =
customer.getSalesOrdersFilterBy(props);
```

Back-end Search

Backend search allows the client to operate on a subset of data, obtained as a result of executing a specific named query on the server.

Search MBO Create

Consider a named query on the server, `BE_SEARCH_GETLIST`. To initiate a back-end search, the first step is to create a search MBO.

```
BackendSearch search = new BackendSearch();
```

Fill up the required fields for the MBO as follows:

```
//any name as desired by the user.
     search.setSearchName("MySearch");

 //entity type for the result set (corresponds to the return type of
the named query )
     search.setEntityType("ENTITY_TYPE_DETAILS");

//the name of the query to be executed  on the server
[search setNamedQuery: [BackendSearch BE_SEARCH_GETLIST]];
     search.setNamedQuery ("BE_SEARCH_GETLIST");
```

For every named query on the server, the generated client code contains a corresponding class file, with the same name as the query. For example, `BE_SEARCH_GETLIST.java`. The attributes of the class represent the parameters for the query and can be set as follows.

```
//Set up the search parameters , which will be used as the search
criteria
     BE_SEARCH_GETLIST searchParameters = new BE_SEARCH_GETLIST();
     searchParameters.setNAME_FIRST("John");
     //additional parameters if required.

//Now set the above as searchparameters in the MBO
     search.setParameters(searchParameters);
```

The search MBO has other optional fields:

```
     search.setSearchTime(com.sybase.afx.util.DateTimeUtil.now());
     search.setTake(100);
     search.setSearchId(1);
     //…
     // other optional fields of search.

//submit search request to the server
     search.submitPending();
```

After some time the server sends a search failure or success message, and the results of the query. You can get the search result notification from `CallbackHandler`. The results are saved into the back-end search results table on the device database. The result data can be retrieved as follows:

```
GernericList<ENTITY_TYPE_DETAILS> searchResults =
ENTITY_TYPE_DETAILS.searchResults(search);
```

Search MBO Update

```
     searchParameters = new BE_SEARCH_GETLIST();
     searchParameters.setNAME_FIRST(“Ron”);
     search.setParameters(searchParameters);
     search.update();
     search.refresh();
     search.submitPending();
```

Search MBO Delete

Delete on a search MBO will delete the search entity and result locally in the client database. Data on the server cannot be deleted using the search MBO and a call to **submitPending** will not propagate the delete message to the server.

```
     search.delete();
```

Persistence APIs

The persistence APIs include operations and object state APIs.

See also

- *Manipulating Data* on page 44

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 non-static instances 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 affect the 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, methods such as `Save` cannot be automatically generated.

See also

- *Creating, Updating, and Deleting MBOs* on page 44
- *Other Operations* on page 45

Create Operation

The `create` operation allows the client to create a new record in the local database. To execute a create operation on an MBO, create a new MBO instance, and set the MBO attributes, then call the `save()` or `create()` operation. To propagate the changes to the server, call `submitPending`.

```
Customer cust = new Customer();
cust.setFname ( "supAdmin" );
cust.setCompany_name( "Sybase" );
cust.setPhone( "777-8888" );
cust.create();// or cust.save();
cust.submitPending();
SUP101DB.synchronize();
// or SUP101DB.synchronize (String synchronizationGroup)
```

Update Operation

The `update` operation updates a record in the local database on the device. To execute update operations on an MBO, get an instance of the MBO, set the MBO attributes, then call either the `save()` or `update()` operation. To propagate the changes to the server, call `submitPending`.

```
Customer cust = Customer.findById(101);
cust.setFname("supAdmin");
cust.setCompany_name("Sybase");
cust.setPhone("777-8888");
cust.save(); // or cust.update();
cust.submitPending();
SUP101DB.synchronize();
// or SUP101DB.synchronize (String synchronizationGroup)

```

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

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

Delete Operation

The `delete` operation allows the client to delete a new record in the local database. To execute delete operations on an MBO, get an instance of the MBO, set the MBO attributes, then call the `delete` operation. To propagate the changes to the server, call `submitPending`.

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

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

```
Customer cust = Customer.findById(101);
GenericList<Sales_order> orders = cust.getSalesOrders();
Sales_order order = orders.item(0);
order.delete();
cust.submitPending();
SUP101DB.synchronize();
// or SUP101DB.synchronize (String synchronizationGroup)
```

Save Operation

The `save` operation saves a record to the local database. In the case of an existing record, a `save` operation calls the `update` operation. If a record does not exist, the `save` operation creates a new record.

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

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

Other Operation

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

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

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

```
CustomerOtherOperation other = new CustomerOtherOperation();
other.setP1("somevalue");
other.setP2(2);
other.setP3(new Date(System.currentTimeMillis()));
other.save();
other.submitPending();
SUP101DB.synchronize(); // or SUP101DB.synchronize (String
synchronizationGroup)
```

Pending Operation

You can manage the pending state.

- **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();
}
```

You can group multiple operations into a single transaction for improved performance:

```
// load the customer MBO with customer ID 100
Customer customer = Customer.findByPrimaryKey(100);

// Change phone number of that customer
customer.setPhone("8005551212");

// use one transaction to do save and submitPending
```

```
com.sybase.persistence.LocalTransaction tx =
SUP101DB.beginTransaction();
try
{
  customer.save();
  customer.submitPending();
  tx.commit();
}
catch (Exception e)
{
  tx.rollback();
}
```

Complex Attribute Types

Some back-end datasources require complex types to be passed in as input parameters. The input parameters can be any of the allowed attribute types, including primitive lists, objects, and object lists. The MBO examples have attributes that are primitive types (such as `int`, `long`, or `string`), and make use of the basic database operations (`create`, `update`, and `delete`).

Passing Structures to Operations

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

The SimpleCaseList MBO contains a `create` operation that has a number of parameters, including one 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 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 it, 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();
```

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	Type	Description
`isNew`	`boolean`	Returns true if this entity is new, but has not yet been created in the client database.

Name	Type	Description
`isCreated`	`boolean`	Returns true if this entity has been newly created in the client 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 (`replay-Failure` 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 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 (`replay-Failure` 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, this attribute's value is 'C' (create), 'U' (update), 'D' (delete), or '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, this attribute's value is 'N'.
`replayCounter`	long	Returns a `long` value that is updated each time a row is created or modified by the client. This value is a unique value obtained from `KeyGenera-tor.generateID` method. Note that the value increases every time it is retrieved.

Name	Type	Description
`replayPending`	long	Returns a `long` value. When a pending row is submitted to the server, the value of `replayCounter` is copied to `replayPending`. This allows the client code to detect if a row has been changed since it was submitted to the server (that is, if the value of `replayCounter` is greater than `replayPending`).
`replayFailure`	long	Returns a `long` value. When the server responds with a `replayFailure` message for a row that was submitted to the server, the value of `replayCounter` is copied to `replayFailure`, and `replayPending` is set to 0.

Entity State Example

Shows how the values of the entities that support pending state change at different stages during the MBO update process. The values that change between different states appear in bold.

Note these entity behaviors:

- The `isDirty` flag is set if the entity changes in memory but is not yet written to the database. Once you save the MBO, this flag clears.
- The `replayCounter` value that gets sent to the Unwired Server is the value in the database before you call `submitPending`. After a successful replay, that value is imported from the Unwired Server.
- The last two entries in the table are two possible results from the operation; only one of these results can occur for a replay request.

Description	Flags/Values

After reading from the database, before any changes are made.	isNew=false
	isCreated=false
	isDirty=false
	isDeleted=false
	isUpdated=false
	pending=false
	pendingChange='N'
	replayCounter=33422977
	replayPending=0
	replayFailure=0

One or more attributes are changed, but changes not saved.	isNew=false
	isCreated=false
	isDirty=**true**
	isDeleted=false
	isUpdated=false
	pending=false
	pendingChange='N'
	replayCounter=33422977
	replayPending=0
	replayFailure=0

Description	Flags/Values

After `entity.save()[entity save]` or `entity.update()[entity update]` is called.	isNew=false
	isCreated=false
	isDirty=**false**
	isDeleted=false
	isUpdated=**true**
	pending=**true**
	pendingChange=**'U'**
	replayCounter=**33424979**
	replayPending=0
	replayFailure=0

After `entity.submitPending()[entity submitPending]` is called to submit the MBO to the server.	isNew=false
	isCreated=false
	isDirty=false
	isDeleted=false
	isUpdated=true
	pending=true
	pendingChange='U'
	replayCounter=33424981
	replayPending=**33424981**
	replayFailure=0

Description	Flags/Values

Possible result: the Unwired Server accepts the update, sends an import and a `replayResult` for the entity, and then refreshes the entity from the database.	isNew=false
	isCreated=false
	isDirty=false
	isDeleted=false
	isUpdated=**false**
	pending=**false**
	pendingChange=**'N'**
	replayCounter=**33422977**
	replayPending=**0**
	replayFailure=0

Possible result: The Unwired Server rejects the update, sends a `replayFailure` for the entity, and refreshes the entity from the database	isNew=false
	isCreated=false
	isDirty=false
	isDeleted=false
	isUpdated=true
	pending=true
	pendingChange='U'
	replayCounter=33424981
	replayPending=**0**
	replayFailure=**33424981**

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();
```

```
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.

For example:

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

Generated Package Database APIs

The generated package database APIs include methods that exist in each generated package database.

Client Database APIs

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

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

Typically, `createDatabase` does not need to be called since it is called internally when necessary. An application may use `deleteDatabase` when uninstalling the application.

Use the transaction API to group several transactions together for better performance.

```
public static com.sybase.persistence.LocalTransaction
beginTransaction()
```

```
Customer customer = Customer.findByPrimaryKey(101);
   // Use one transaction to save and submit pending
   LocalTransaction tx = SUP101DB.beginTransaction();
   // modify customer information
   customer.save();
   customer.submitPending();
   tx.commit();
```

Large Attribute APIs

Use large string and binary attributes.

You can import large messages containing binary objects (BLOBs) to the client, send new or changed large objects to the server, and efficiently handle large attributes on the client.

The large attribute APIs allow clients to import large messages from the server or send a replay message without using excessive memory and possibly throwing exceptions. Clients can also access or modify a large attribute without reading the entire attribute into memory. In addition, clients can execute queries without having large attribute valuies automatically filled in the returned MBO lists or result sets.

BigBinary

An object that allows access to a persistent binary value that may be too large to fit in available memory. A streaming API is provided to allow the value to be accessed in chunks.

close

Closes the value stream.

Closes the value stream. Any buffered writes are automatically flushed. Throws a `StreamNotOpenException` if the stream is not open.

Syntax

```
public void close()
```

Examples

- **–** Writes a binary book cover image and closes the image file.

  ```
  Book book = Book.findByPrimaryKey(bookID);
  com.sybase.persistence.BigBinary image = book.getCover();
  image.openForWrite(0);
  // ...
  image.close();
  ```

copyFromFile

Overwrites this `BigBinary` object with data from the specified file.

Any previous contents of the file will be discarded. Throws an `ObjectNotSavedException` if this `BigBinary` object is an attribute of an entity that has not yet been created in the database. Throws a `StreamNotClosedException` if the object is not closed.

Syntax

```
public void copyFromFile(java.lang.String filepath)
```

Parameters

- **filepath –** The file containing the data to be copied.

copyToFile

Overwrites the specified file with the contents of this `BigBinary` object.

Any previous contents of the file are discarded. Throws an `ObjectNotSavedException` if this `BigBinary` object is an attribute of an entity that has not yet been created in the database. Throws a `StreamNotClosedException` if the object is not closed.

Syntax

```
public void copyToFile(java.lang.String filepath)
```

Parameters

- **filepath –** The file to be overwritten.

flush

Flushes any buffered writes.

Flushes any buffered writes to the database. Throws a `StreamNotOpenException` if the stream is not open.

Syntax

```
public void flush()
```

openForRead

Opens the value stream for reading.

Has no effect if the stream was already open for reading. If the stream was already open for writing, it is flushed before being reopened for reading. Throws an `ObjectNotSavedException` if this `BigBinary` object is an attribute of an entity that has not yet been created in the database. Throws an `ObjectNotFoundException` if this object is null.

Syntax

```
public void openForRead()
```

Examples

- **–** Opens a binary book image for reading.

  ```
  Book book = Book.findByPrimaryKey(bookID);
  com.sybase.persistence.BigBinary image = book.getCover();
  image.openForRead();
  ```

openForWrite

Opens the value stream for writing.

Any previous contents of the value will be discarded. Throws an `ObjectNotSavedException` if this `BigBinary` object is an attribute of an entity that has not yet been created in the database.

Syntax

```
public void openForWrite(long newLength)
```

Parameters

- **newLength –** The new value length in bytes. This parameter is required for some platforms, but for Android the parameter value is ignored, and can be specified as 0.

Examples

- **–** Opens a binary book image for writing.

  ```
  Book book = Book.findByPrimaryKey(bookID);
  com.sybase.persistence.BigBinary image = book.getCover();
  image.openForWrite(0);
  ```

read

Reads a chunk of data from the stream.

Reads and returns the specified number of bytes, or fewer if the end of stream is reached. Throws a `StreamNotOpenException` if the stream is not open for reading.

Syntax

```
public byte[] read(int length)
```

Parameters

- **length –** The maximum number of bytes to be read into the chunk.

Returns

`read` returns a chunk of binary data read from the stream, or a null value if the end of the stream has been reached.

Examples

- **–** Reads in a binary book image.

  ```
  Book book = Book.findByPrimaryKey(bookID);
  com.sybase.persistence.BigBinary image = book.getCover();
  ```

```
int bufferLength = 1024;
image.openForRead();
byte[] binary = image.read(bufferLength);
while (binary != null)
{
    binary = image.read(bufferLength);
}
image.close();
```

readByte
Reads a single byte from the stream.

Throws a `StreamNotOpenException` if the stream is not open for reading.

Syntax

```
public int readByte()
```

Returns

`readByte` returns a byte of data read from the stream, or -1 if the end of the stream has been reached.

seek
Changes the stream position.

Throws a `StreamNotOpenException` if the stream is not open for reading.

Syntax

```
public void seek(long newPosition)
```

Parameters

- **newPosition –** The new stream position in bytes. Zero represents the beginning of the value stream.

write
Writes a chunk of data to the stream.

Writes data to the stream, beginning at the current position. The stream may be buffered, so use `flush` or `close` to be certain that any buffered changes have been applied. Throws a `StreamNotOpenException` if the stream is not open for writing. Throws a `WriteAppendOnlyException` if the platform only supports appending to the end of a value and the current stream position precedes the end of the value. Throws a `WriteOverLengthException` if the platform requires the length to be predetermined before writing and this write would exceed the predetermined length.

Syntax

```
public void write(byte[] data)
```

Parameters

- **data –** The data chunk to be written to the stream.

Examples

- **–** Opens a binary book image for writing.

  ```
  Book book = Book.findByPrimaryKey(bookID);
  com.sybase.persistence.BigBinary image = book.getCover();
  image.openForWrite(0);
  byte[] binary = new byte[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
  image.write(binary);
  ```

writeByte

Writes a single byte to the stream.

Writes a byte of data to the stream, beginning at the current position. The stream may be buffered, so use `flush` or `close` to be certain that any buffered changes have been applied. Throws a `StreamNotOpenException` if the stream is not open for writing. Throws a `WriteAppendOnlyException` if the platform only supports appending to the end of a value and the current stream position precedes the end of the value. Throws a `WriteOverLengthException` if the platform requires the length to be predetermined before writing and this write would exceed the predetermined length.

Syntax

```
public void writeByte(byte data)
```

Parameters

- **data –** The byte value to be written to the stream.

BigString

An object that allows access to a persistent string value that might be too large to fit in available memory. A streaming API is provided to allow the value to be accessed in chunks.

close

Closes the value stream.

Closes the value stream. Any buffered writes are automatically flushed. Throws a `StreamNotOpenException` if the stream is not open.

Syntax

```
public void close()
```

Examples

- **–** Writes to the biography file, and closes the file.

  ```
  Author author = Author.findByPrimaryKey(authorID);
  BigString text = author.getBiography();
  text.openForWrite(0);
  text.write("something");
  text.close();
  ```

copyFromFile

Overwrites this `BigString` object with data from the specified file.

Any previous contents of the value will be discarded. Throws an `ObjectNotSavedException` if this `BigString` object is an attribute of an entity that has not yet been created in the database. Throws a `StreamNotClosedException` if the object is not closed.

Syntax

```
public void copyFromFile(java.lang.String filepath)
```

Parameters

- **filepath –** The file containing the data to be copied.

copyToFile

Overwrites the specified file with the contents of this `BigString` object.

Any previous contents of the file are discarded. Throws an `ObjectNotSavedException` if this `BigString` object is an attribute of an entity that has not yet been created in the database. Throws a `StreamNotClosedException` if the object is not closed.

Syntax

```
public void copyToFile(java.lang.String filepath)
```

Parameters

- **filepath –** The file to be overwritten.

flush

Flushes any buffered writes.

Flushes any buffered writes to the database. Throws a `StreamNotOpenException` if the stream is not open.

Syntax

```
public void flush()
```

openForRead

Opens the value stream for reading.

Has no effect if the stream was already open for reading. If the stream was already open for writing, it is flushed before being reopened for reading. Throws an `ObjectNotSavedException` if this `BigString` object is an attribute of an entity that has not yet been created in the database.

Syntax

```
public void openForRead()
```

Examples

- **–** Opens the biography file for reading.

  ```
  Author author = Author.findByPrimaryKey(authorID);
  BigString text = author.getBiography();
  text.openForRead();
  ```

openForWrite

Opens the value stream for writing.

Any previous contents of the value will be discarded. Throws an `ObjectNotSavedException` if this `BigString` object is an attribute of an entity that has not yet been created in the database.

Syntax

```
public void openForWrite(long newLength)
```

Parameters

- **newLength –** The new value length in bytes. Some platforms may allow this parameter to be specified as 0, with the actual length to be determined later, depending on the amount of data written to the stream. Other platforms require the total amount of data written to the stream to match the specified value.

Examples

- **–** Opens the biography file for writing.

  ```
  Author author = Author.findByPrimaryKey(authorID);
  BigString text = author.getBiography();
  text.openForWrite(0);
  ```

read
Reads a chunk of data from the stream.

Reads and returns the specified number of characters, or fewer if the end of stream is reached. Throws a `StreamNotOpenException` if the stream is not open for reading.

Syntax

```
public String read(int length)
```

Parameters

- **length –** The maximum number of characters to be read into the chunk.

Returns

`read` returns a chunk of string data read from the stream, or a null value if the end of the stream has been reached.

Examples

- – Reads in the biography file.

```
Author author = Author.findByPrimaryKey(authorID);
BigString text = author.getBiography();
text.openForRead();
int bufferLength = 1024;

String something = text.read(bufferLength); //null if EOF
while (something != null)
{
    something = text.read(bufferLength);
}
text.close();
```

readChar
Reads a single character from the stream.

Throws a `StreamNotOpenException` if the stream is not open for reading.

Syntax

```
public int readChar()
```

Returns

`readChar` returns a single character read from the stream, or -1 if the end of the stream has been reached.

seek

Changes the stream position.

Throws a `StreamNotOpenException` if the stream is not open for reading.

Syntax

```
public void seek(long newPosition)
```

Parameters

- **newPosition –** The new stream position in characters. Zero represents the beginning of the value stream.

write

Writes a chunk of data to the stream.

Writes data to the stream, beginning at the current position. The stream may be buffered, so use `flush` or `close` to be certain that any buffered changes have been applied. Throws a `StreamNotOpenException` if the stream is not open for writing. Throws a `WriteAppendOnlyException` if the platform only supports appending to the end of a value and the current stream position precedes the end of the value. Throws a `WriteOverLengthException` if the platform requires the length to be predetermined before writing and this write would exceed the predetermined length.

Syntax

```
public void write(java.lang.String data)
```

Parameters

- **data –** The data chunk to be written to the stream.

Examples

- **–** Writes to the biography file, and closes the file.

  ```
  Author author = Author.findByPrimaryKey(authorID);
  BigString text = author.getBiography();
  text.openForWrite(0);
  text.write("something");
  text.close();
  ```

writeChar

Writes a single character to the stream.

Writes a character of data to the stream, beginning at the current position. The stream may be buffered, so use `flush` or `close` to be certain that any buffered changes have been applied. Throws a `StreamNotOpenException` if the stream is not open for writing. Throws a

`WriteAppendOnlyException` if the platform only supports appending to the end of a value and the current stream position precedes the end of the value. Throws a `WriteOverLengthException` if the platform requires the length to be predetermined before writing and this write would exceed the predetermined length.

Syntax

```
public void writeChar(char data)
```

Parameters

- **data –** The character value to be written to the stream.

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.

MetaData classes are generated automatically. However, you must use the `-rm` option to generate the object manager class.

DatabaseMetaData

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

Any entity for which "allow dynamic queries" is enabled generates attribute metadata. Depending on the options selected in the Eclipse IDE, metadata for attributes and operations may be generated for all classes and entities.

```
DatabaseMetaData dmd = SUP101DB.getMetaData();
com.sybase.collections.StringList syncGroups =
dmd.getSynchronizationGroups();
for(int i=0; i<syncGroups.size(); i++)
{
String syncGroup = syncGroups.item(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());
```

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

A server-side exception occurs when a client tries to update or create a record and the Unwired Server throws an exception.

A server-side exception results in a stack trace in the server log, and a log record (`LogRecordImpl`) imported to the client with information on the problem.

HTTP Error Codes

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

These tables list recoverable and unrecoverable error codes. All error codes that are not explicitly considered recoverable are considered unrecoverable.

Table 4. Recoverable Error Codes

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

Table 5. Unrecoverable Error Codes

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

Error code 401 is not treated as a simple recoverable error. If the `SupThrowCredentialRequestOn401Error` context variable is set to true (the default), error code 401 throws a `CredentialRequestException`, which sends a credential request notification to the user's inbox. You can change this behavior by modifying the value of the `SupThrowCredentialRequestOn401Error` context variable in Sybase Control Center. If `SupThrowCredentialRequestOn401Error` is set to false, error code 401 is treated as a normal recoverable exception.

Mapping of EIS Codes to Logical HTTP Error Codes

A list of SAP® error codes mapped to HTTP error codes. By default, SAP error codes that are not listed map to HTTP error code 500.

Note: These JCO error codes are not applicable for DOE-based applications.

Table 6. Mapping of SAP Error Codes to HTTP Error Codes

Constant	Description	HTTP Error Code
JCO_ERROR_COMMUNICATION	Exception caused by network problems, such as connection breakdowns, gateway problems, or unavailability of the remote SAP system.	503
JCO_ERROR_LOGON_FAILURE	Authorization failures during login. Usually caused by unknown user name, wrong password, or invalid certificates.	401
JCO_ERROR_RESOURCE	Indicates that JCO has run out of resources such as connections in a connection pool.	503
JCO_ERROR_STATE_BUSY	The remote SAP system is busy. Try again later.	503

Client-Side Exceptions

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

Note: See *Callback Handlers*.

Exception Classes

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

- **SynchronizeException –** thrown when an error occurs during synchronization.
- **PersistenceException –** thrown when trying to access the local database.
- **ObjectNotFoundException –** thrown when trying to load an MBO that is not inside the local database.
- **NoSuchOperationException –** thrown when trying to call a method (using the Object Manager API) but the method is not defined for the MBO.
- **NoSuchAttributeException –** thrown when trying to access an attribute (using the Object Manager API) but the attribute is not defined for the MBO.
- **ApplicationRuntimeException –** thrown when a call to start the connection, register the application, or unregister the application cannot be completed due to an error.
- **ConnectionPropertyException –** thrown when a call to start the connection, register the application, or unregister the application cannot be completed due to an error in a connection property value or application identifier.

Index