Developer Guide: Android Object API Applications

SAP Mobile Platform 2.3 SP03

DOCUMENT ID: DC01908-01-0233-01
LAST REVISED: September 2013

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

Contents

Getting Started with Android Development

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

- *SAP Mobile WorkSpace - Mobile Business Object Development*
- *Supported Hardware and Software*
- *Troubleshooting*.
- A complete Client Object API reference is available in *SMP_HOME* `\MobileSDK23\ObjectAPI\apidoc\android`
- *Fundamentals* contains high-level mobile computing concepts, and a description of how SAP Mobile Platform implements the concepts in your enterprise.
- *Developer Guide: Migrating to SAP Mobile SDK* contains information for developers who are migrating device applications to a newer software version, and changes to MBOs, projects, and the SAP Mobile Server.

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. SAP Mobile 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 packages, that are deployed to the SAP Mobile 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 SAP Mobile Platform is used in a stand-alone manner for data synchronization (without Data Orchestration Engine), it utilizes an efficient bulk transfer and data insertion technology between the middleware cache and the device database.

In an SAP Mobile 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, SAP Mobile 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 SAP Mobile 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 SAP Mobile 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 SAP Mobile Server employs several components in the synchronization chain.

Mobile Channel Interfaces

Two main channel interfaces provide notifications and data transport to and from remote devices.

- 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.

  The messaging channel sends these types of communications:
  - Application registration - the messaging channel is used for application registration before establishing a connection to the SAP Mobile Server.
  - Change notifications - when the SAP Mobile Server detects changes in the back-end EIS, the SAP Mobile 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.
  - Operation replay records - when synchronizing, these records are sent to the SAP Mobile Server and the messaging channel sends a notification of `replayFinished`. The application must call another synchronize method to retrieve the result.
  - SAP Data Orchestration Engine (DOE) application synchronization - the messaging channel is used for synchronization for DOE applications.
- The synchronization channel sends data to keep the SAP Mobile 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 SAP Mobile 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 EIS. Data that changes on the EIS as a result of device changes, or those originating elsewhere, are replicated to the device database.

Documentation Roadmap for SAP Mobile Platform

SAP® Mobile 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.

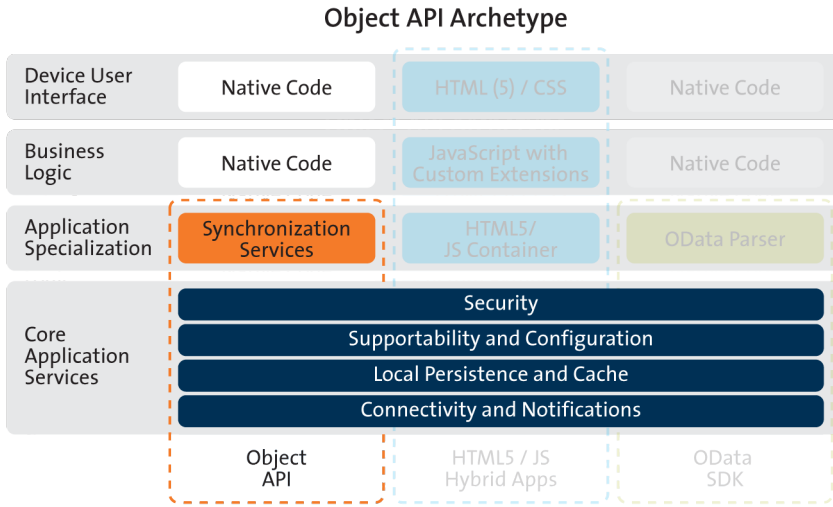
See *Documentation Roadmap* in *Fundamentals* for document descriptions by user role.

Check the Product Documentation Web site regularly for updates: *http://sybooks.sybase.com/sybooks/sybooks.xhtml?id=1289&c=firsttab&a=0&p=categories*, 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 SAP Mobile 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 SAP Mobile 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 applications, 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. *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 SAP Mobile WorkSpace, or by using a command line utility for generating code.
3. *Creating a Project*

   Build a device application project.
4. *Developing the Application Using the Object API*

   Use the Object API to develop 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 SAP Mobile WorkSpace*

   Install the supported version of Android Development Tools (ADT) directly in the SAP Mobile 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

- *Generating Java Object API Code* on page 8

Installing the Android SDK

Install the Android SDK.

1. Confirm that your system meets the requirements at *http://developer.android.com/sdk/requirements.html*.
2. Download and install the supported version of the Android SDK starter package.

   See the *Google Android Versions* topic for your platform in *Supported Hardware and Software* at *http://sybooks.sybase.com/sybooks/sybooks.xhtml?id=1289&c=firsttab&a=0&p=categories*. Select the appropriate version of the SAP Mobile Platform document set.
3. Launch the Android SDK Manager and install the Android tools (SDK Tools and SDK Platform-tools) and the Android API.
4. Launch the **Android Virtual Device Manager**, and create an Android virtual device to use as your emulator.

Installing ADT in SAP Mobile WorkSpace

Install the supported version of Android Development Tools (ADT) directly in the SAP Mobile WorkSpace Eclipse environment.

Follow the instructions for installing the ADT Plugin for Eclipse at *http://developer.android.com/sdk/installing/installing-adt.html*.

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`

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 SAP Mobile WorkSpace, or by using a command line utility for generating code.

Generated code can be used to leverage SAP Mobile Platform capabilities and services, and access MBO-related data: calling the mobile business object operations, object queries, and so on. This code can then be imported into an integrated development environment (IDE) of your choice to create the device application (define the user interface, application logic, and so on.

See also

- *Installing the Android Development Environment* on page 6
- *Creating a Project* on page 15

Generating Java Object API Code Using SAP Mobile WorkSpace

Use SAP Mobile 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

SAP Mobile 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
Code generation configuration	A table lists all existing named configurations plus the most recently used configuration. You can select any of these, click **Next**, and proceed. Additionally, you can: • Create new configuration – click **Add** and enter the **Name** and optional **Description** of the new configuration and click **OK** to save the configuration for future sessions. You can also select **Copy from** to copy an existing configuration which can then be modified. • Most recent configuration – if you click **Next** the first time you generate code without creating a configuration, the configuration is saved and displays as the chosen configuration the next time you invoke the code generation wizard. If the most recent configuration used is a named configuration, it is saved as the first item in the configuration table, and also "Most recent configuration", even though it is still listed as the original named configuration.

3. Click **Next**.

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

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

   SAP Mobile WorkSpace automatically computes the default page size after you choose the MBOs based on total attribute size. If an MBO's accumulated attribute size is larger than the page size setting, a warning displays.

5. 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. • Java • Android
SAP Mobile Server	Specify a default SAP Mobile Server connection profile to which the generated code connects at runtime.
Server domain	Choose the domain to which the generated code will connect. If you specified an SAP Mobile Server to which you previously connected successfully, the first domain in the list is chosen by default. You can enter a different domain manually. **Note:** This field is only enabled when an SAP Mobile Server is selected.

Option	Description
Page size	(Optional) Select the page size for the generated client code. If the page size is not set, the default page size is 4KB at runtime. The default is a proposed page size based on the selected MBO's attributes. The page size should be larger than the sum of all attribute lengths for any MBO that is included with all the MBOs selected, and must be valid for the database. If the page size is changed, but does not meet these guidelines, object queries that use string or binary attributes with a **WHERE** clause may fail. See *MBO Attributes* in *Mobile Data Models: Using Mobile Business Objects* for more information. 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). 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 SAP Mobile Server is selected.
Package, Namespace, or Name Prefix	• Package – enter a package name for Java. The package name must follow Java naming conventions for packages. For example, no leading or trailing spaces and no special characters such as §&/, except that the first letter may be upper-case.

Option	Description
Destination	Specify the destination of the generated device client files. Enter (or **Browse**) to either a **Project path** (Mobile Application project) location or **File system path** location. Select **Clean up destination before code generation** to clean up the destination folder before generating the device client files. **Note:** If you select Java as the language, enter a project path, specify a mobile application project folder, and select `Generated Code` as the destination. JAR files are automatically added to the destination for the platform that supports compiling of the generated client code.
Third-party jar file	Enter or browse to the location of the third party jar file. For example, `net_rim_api.jar` for BlackBerry, or `android.jar` for Android. If you select Java as the language, and if the BlackBerry or Android third-party JAR file has not been added, the warning `The dependent third-party class 'net.rim.device.api.system.ApplicationDescriptor' cannot be found` or `The dependent third-party class 'android.content.Context' cannot be found` displays.

6. 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.
7. Select **Including object manager classes** to generate both the metadata for the attributes and operations of each generated client object and an object manager for the generated metadata.

   The **Including object manager classes** option is enabled only for BlackBerry and C# if you select **Generate metadata classes**. The object manager allows you to retrieve the metadata of packages, MBOs, attributes, operations, and parameters during runtime using the name instead of the object instance.

   Note: When generating code for Android, "Generate metadata classes" is automatically selected and cannot be unselected. The "Including object manager classes" option is unavailable and unsupported.

8. Click **Finish**.
9. Examine the generated code location and contents.
10. 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 SAP Mobile WorkSpace to develop and package your mobile business objects. See *SAP Mobile WorkSpace - Mobile Business Object Development* > *Develop* > *Developing a Mobile Business Object*.
- Deploy the package to the SAP Mobile Server, creating files required for code generation from the command line. See *SAP Mobile WorkSpace - Mobile Business Object Development* > *Develop* > *Packaging and Deploying Mobile Business Objects* > *Automated Deployment of SAP Mobile WorkSpace Projects*.

Task

1. Locate `<domain name>_package.jar` in your mobile project folder. For the SMP101 example, the project is deployed to the default domain, and the deploy jar file is in the following location: `SMP101\Deployment\.pkg.profile \My_SAP_Mobile_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 *`SMP_HOME`*`\MobileSDK23\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 *`SMP_HOME`*`\MobileSDK23\ObjectAPI \Utils\genfiles` directory, assuming `codegen.bat` is run from the *`SMP_HOME`*`\MobileSDK23\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 SAP Mobile 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, generated object API code is stored in the *SMP_HOME*`\MobileSDK23\ObjectAPI\Utils\genfiles` folder after you generate code.

The contents of the folder is determined by the options you selected in the Generate Code wizard in SAP Mobile 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.
- DatabaseClass – package level class that handles subscription, login, synchronization, and other operations for the package.
- 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.

SAP Mobile 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 **SAP AG > 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.

Creating a Project

Build a device application project.

1. *Creating a Project in SAP Mobile WorkSpace*

   Create a project for your Android device application in SAP Mobile WorkSpace.
2. *Downloading the Latest Afaria Libraries*

   Afaria® provides provisioning of configuration data and certificates for your SAP Mobile Platform client application. Afaria libraries are packaged with SAP Mobile Platform, but may not be the latest software available. To ensure you have the latest Afaria libraries, download Afaria software.
3. *Importing Libraries and Code*

   Create a package with the same name as the package of the code generated in SAP Mobile Workspace.

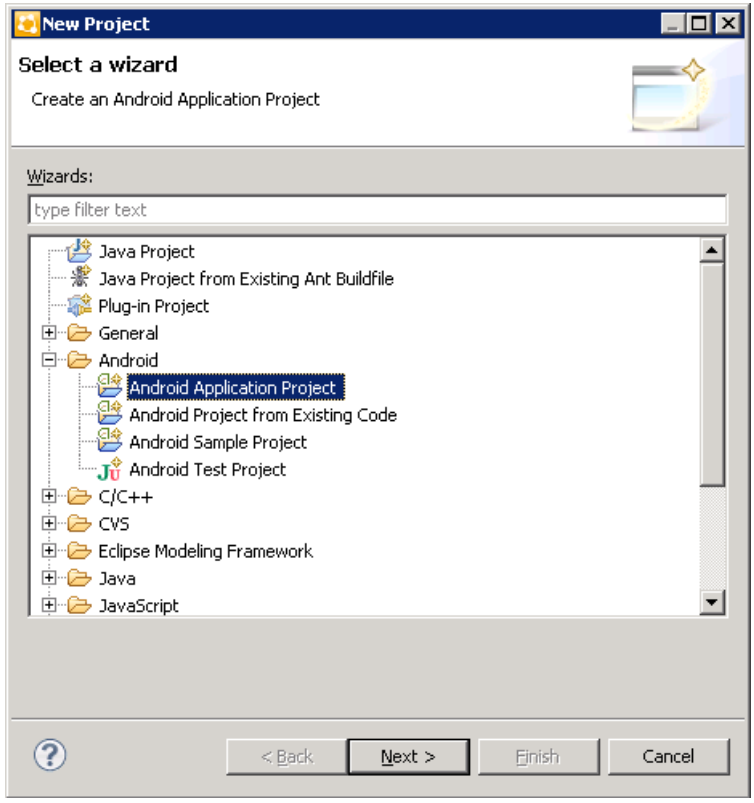
See also

- *Generating Java Object API Code* on page 8
- *Developing the Application Using the Object API* on page 29

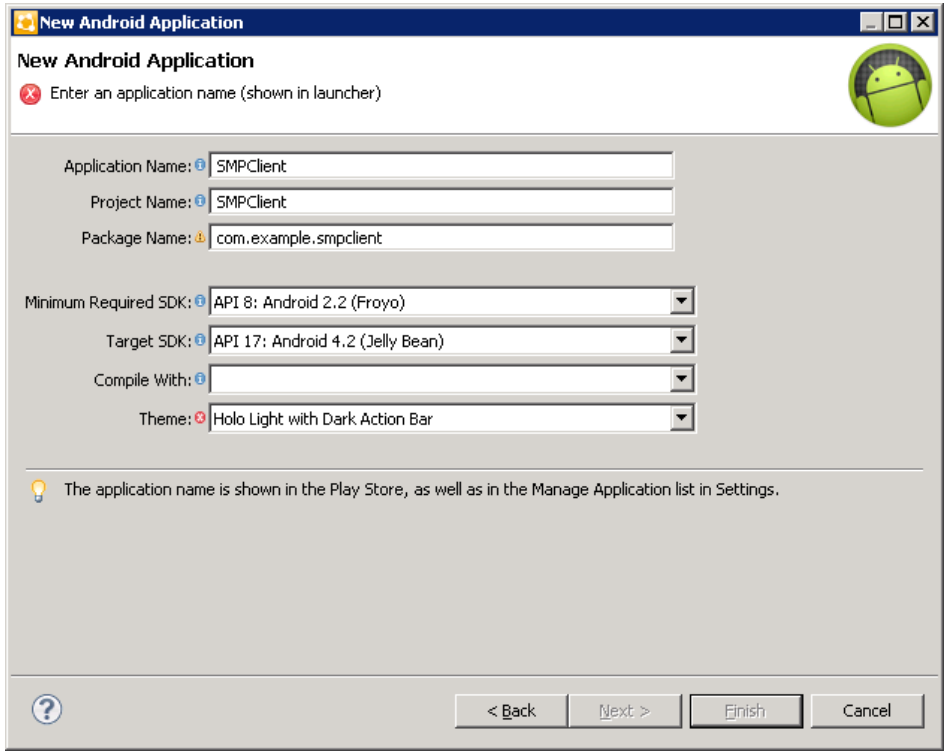
Creating a Project in SAP Mobile WorkSpace

Create a project for your Android device application in SAP Mobile WorkSpace.

1. In SAP Mobile WorkSpace, select **File > New > Project**.
2. Select **Android > Android Application Project**.



3. In the **New Android Project** wizard, enter these values and click **Finish**:
   - **Application Name: –** enter the name of the application.
   - **Project Name: –** enter a unique project name in Eclipse. This can be the same as the application name.
   - **Package Name: –** enter the name of the package as a unique identifier for the application. Must be a valid Java package.
   - **Minimum Required SDK: –** enter the lowest version of the SDK that the application supports.
   - **Target SDK: –** enter the highest API level that the application is known to work with.
   - **Compile With: –** choose a target API to compile your code against, from your installed SDKs.
   - **Theme: –** choose the base theme to use for the application.



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>
```

   a) Open the `manifest` file.
   b) Select the **Permissions** tab.
   c) Click**Add**.
   d) Select **Uses Permission** and click **OK**.
   e) In the **Name** field, enter the name of the permission. In this case, add five **Uses Permission** entries and name them:

```
android.permission.INTERNET
android.permission.READ_PHONE_STATE
android.permission.ACCESS_WIFI_STATE
android.permission.ACCESS_NETWORK_STATE
android.permission.WRITE_EXTERNAL_STORAGE
```

f) Save your changes.

Downloading the Latest Afaria Libraries

Afaria® provides provisioning of configuration data and certificates for your SAP Mobile Platform client application. Afaria libraries are packaged with SAP Mobile Platform, but may not be the latest software available. To ensure you have the latest Afaria libraries, download Afaria software.

1. Navigate to the Mobile Enterprise Technical Support website at *http://frontline.sybase.com/support/downloads.aspx*.
2. If not registered, register for an account.
3. Log into your account.
4. Select **Software Updates** and download the latest Static Link Libraries.
5. Extract the contents of the downloaded zip file.
6. Copy the Afaria library files into the Android development environment.
7. Include the Afaria library into your project.

Importing Libraries and Code

Create a package with the same name as the package of the code generated in SAP Mobile Workspace.

1. In your SAP Mobile WorkSpace project, create a `libs` directory.
2. Copy the following library and JAR files from *SMP_HOME*`\MobileSDK<version>\ObjectAPI\Android` into the `libs` directory within your project, retaining the same structure:

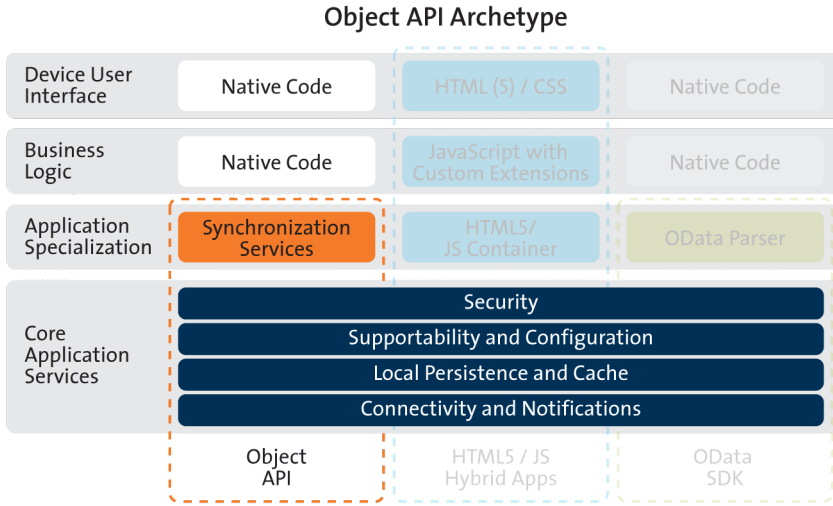
Location	Files
• *SMP_HOME*`\MobileSDK<version>\ObjectAPI\Android`	• `AfariaSSL.jar` • `ClientLib.jar` • `sup-client.jar` • `UltraLiteJNI12.jar`
• *SMP_HOME*`\MobileSDK<version>\ObjectAPI\Android\armeabi`	• `libmlcrsa12.so` • `libultralitej12.so`

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

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 SAP Mobile 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 SAP Mobile 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 DOE-based applications, connectivity and notifications use the Messaging channel.

1. *Installing the Android Development Environment*

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

2. *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.

3. *Creating a Project*

   Build a device application project.

4. *Developing the Application Using the Object API*

   Use the Object API to develop 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

- *Generating Java Object API Code* on page 22

Installing the Android SDK

Install the Android SDK.

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

2. Download and install the supported version of the Android SDK starter package.

   See the *Google Android Versions* topic for your platform in *Supported Hardware and Software* at *http://sybooks.sybase.com/sybooks/sybooks.xhtml?id=1289&c=firsttab&a=0&p=categories*. Select the appropriate version of the SAP Mobile Platform document set.
3. Launch the Android SDK Manager and install the Android tools (SDK Tools and SDK Platform-tools) and the Android API.
4. Launch the **Android Virtual Device Manager**, and create an Android virtual device to use as your emulator.

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`

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 SAP Mobile Platform package.
- Deploy the package to the SAP Mobile Server.

See *Create, Generate, and Download the ESDMA Bundle*, *Convert the ESDMA Bundle into an SAP Mobile Platform Package*, and *Deploy the SAP Mobile Platform Package* in *Mobile Data Models: Using Data Orchestration Engine*.

Task

1. Locate `<domain name>_package.jar` in your mobile project folder. For the SMP101 example, the project is deployed to the default domain, and the deploy jar file is in the following location: `SMP101\Deployment\.pkg.profile \My_SAP_Mobile_Server\default_package.jar`.
2. Make sure that the JAR file contains this file:
   - `deployment_unit.xml`
3. From *SMP_HOME*`\MobileSDK23\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 *SMP_HOME*`\MobileSDK23\ObjectAPI \Utils\genfiles` directory, assuming codegen.bat is run from the *SMP_HOME* `\MobileSDK23\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

- *Installing the Android Development Environment* on page 20
- *Creating a Project* on page 23

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.

Creating a Project

Build a device application project.

1. *Creating a Project in SAP Mobile WorkSpace*

   Create a project for your Android device application in SAP Mobile WorkSpace.
2. *Downloading the Latest Afaria Libraries*

   Afaria® provides provisioning of configuration data and certificates for your SAP Mobile Platform client application. Afaria libraries are packaged with SAP Mobile Platform, but may not be the latest software available. To ensure you have the latest Afaria libraries, download Afaria software.
3. *Importing Libraries and Code*

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

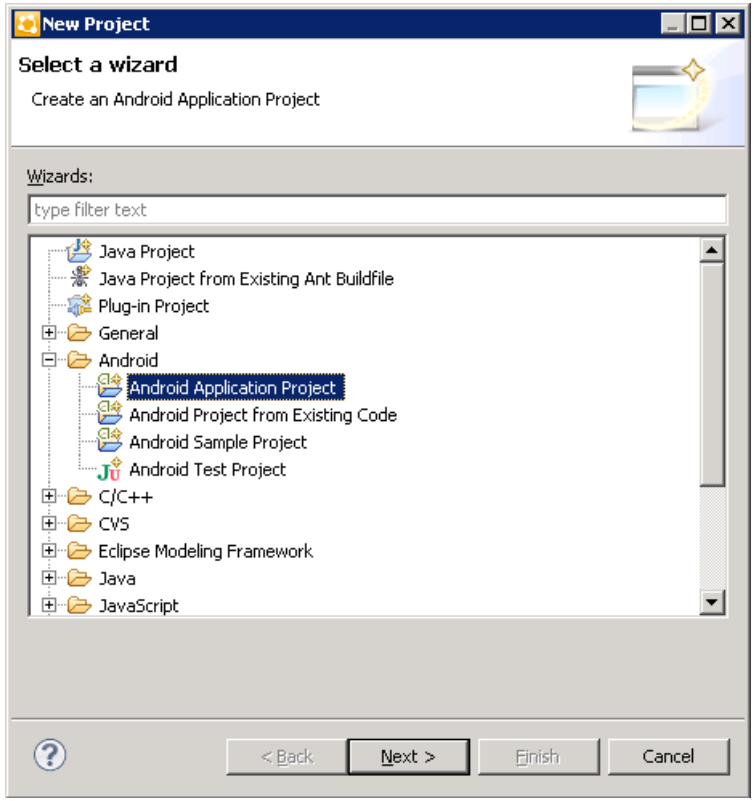
See also

- *Generating Java Object API Code* on page 22
- *Developing the Application Using the Object API* on page 29

Creating a Project in SAP Mobile WorkSpace

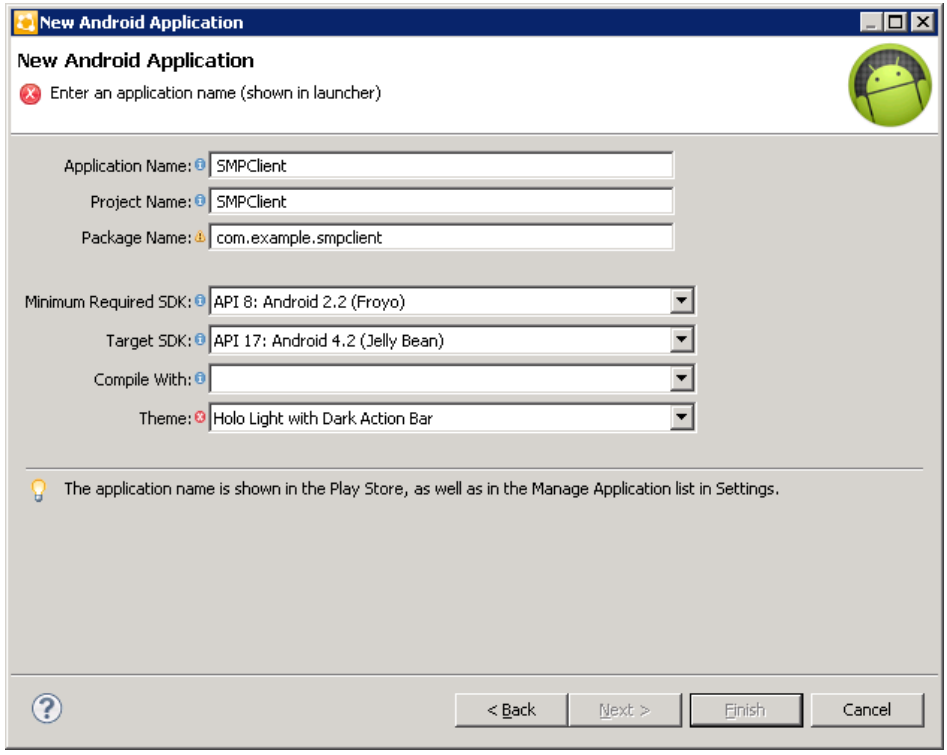
Create a project for your Android device application in SAP Mobile WorkSpace.

1. In SAP Mobile WorkSpace, select **File** > **New** > **Project**.
2. Select **Android** > **Android Application Project**.



3. In the **New Android Project** wizard, enter these values and click **Finish**:
   - **Application Name: –** enter the name of the application.
   - **Project Name: –** enter a unique project name in Eclipse. This can be the same as the application name.
   - **Package Name: –** enter the name of the package as a unique identifier for the application. Must be a valid Java package.
   - **Minimum Required SDK: –** enter the lowest version of the SDK that the application supports.
   - **Target SDK: –** enter the highest API level that the application is known to work with.
   - **Compile With: –** choose a target API to compile your code against, from your installed SDKs.

- **Theme: –** choose the base theme to use for the application.



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>
```

a) Open the `manifest` file.
b) Select the **Permissions** tab.
c) Click**Add**.
d) Select **Uses Permission** and click **OK**.
e) In the **Name** field, enter the name of the permission. In this case, add five **Uses Permission** entries and name them:

```
android.permission.INTERNET
android.permission.READ_PHONE_STATE
android.permission.ACCESS_WIFI_STATE
android.permission.ACCESS_NETWORK_STATE
android.permission.WRITE_EXTERNAL_STORAGE
```

f) Save your changes.

Downloading the Latest Afaria Libraries

Afaria® provides provisioning of configuration data and certificates for your SAP Mobile Platform client application. Afaria libraries are packaged with SAP Mobile Platform, but may not be the latest software available. To ensure you have the latest Afaria libraries, download Afaria software.

1. Navigate to the Mobile Enterprise Technical Support website at *http://frontline.sybase.com/support/downloads.aspx*.
2. If not registered, register for an account.
3. Log into your account.
4. Select **Software Updates** and download the latest Static Link Libraries.
5. Extract the contents of the downloaded zip file.
6. Copy the Afaria library files into the Android development environment.
7. Include the Afaria library into your project.

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 SAP Mobile WorkSpace project, create a `libs` directory.
2. Copy the following library and JAR files from *SMP_HOME*`\MobileSDK23\ObjectAPI\Android` into the `libs` directory within your project, retaining the same structure:

Location	Files
• *SMP_HOME*`\MobileSDK23\ObjectAPI\Android`	• `AfariaSSL.jar` • `ClientLib.jar` • `sup-client.jar` • `UltraLiteJNI12.jar`
• *SMP_HOME*`\MobileSDK23\ObjectAPI\Android\armeabi`	• `libmlcrsa12.so` • `libultralitej12.so`

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

4. Click on the active target, and modify the Header Search Path from Building Settings.

   Specify the path to the location where you copied the include files, and select the Recursive checkbox. The header files in the client library are grouped into subdirectories `public` and `internal`, so the recursive option is required.

Developing the Application Using the Object API

Use the Object API to develop 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

- *Creating a Project* on page 15
- *Testing Applications* on page 71
- *Creating a Project* 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 SAP Mobile 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 UltraLite®J runtime tuning values.
4. *Setting Up Connectivity*

   Store connection information to the SAP Mobile 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.

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 85
- *Connection APIs* on page 121

Setting Up Application Properties

The Application instance contains the information and authentication credentials needed to register and connect to the SAP Mobile 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 SAP
Mobile Server
app.setApplicationIdentifier("SMP101");
// 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 SAP Mobile Server
ConnectionProperties connProps = app.getConnectionProperties();
```

```
connProps.setServerName("server.mycompany.com");
// if you are using Relay Server, then use the correct port number
for the Relay Server.
// if connecting using http without a relay server, use the messaging
administration port, by default 5001.
// if connecting using https without a relay server, then use a new
port for https, for example 9001.
connProps.setPortNumber(5001);

// if connecting using https without a relay server, set the network
protocol
connProps.setNetworkProtocol("https");

//  Set FarmId and UrlSuffix when connecting through the Relay
Server.

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

// Initialize generated package database class with this Application
instance
SMP101DB.setApplication(app);
```

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

If you are using a Relay Server, specify the connection as follows:

```
// specify Relay Server Host
connProps.setServerName("relayserver.mycompany.com");
// specify Relay Server Port (port 80 by default)
connProps.setPortNumber(80);
// specify the Relay Server MBS Farm, for example MBS_Farm
connProps.setFarmId("MBS_FARM");
```

Optionally, you can specify the Relay Server URL suffix.

Using a Reverse Proxy for Object API applications

The Object API application communicates with SAP Mobile Server through two ports:

1. Application registration (default 5001)
2. Application synchronization (default 2480)

The SAP Mobile Server Administrator configures two ports with each port serving one SAP Mobile Server port, so that:

- The root context of http://reverseProxy:5001 maps to http://server-name:5001
- The root context of http://reverseProxy:2480 maps to http://server-name:2480

Set Object API application connection properties just as you would to directly connect to SAP Mobile Server.

The SAP Mobile Server Administrator configures two contexts for one SAP Mobile Server port, so that:

- The "/smp/message" context of http://reverseProxy:8080 maps to http://server-name:5001
- The "/smp/mobilink" context of http://reverseProxy:8080 maps to http://server-name:2480

Set the URL suffix for the Object API application to "/smp/message" for registering applications and "/smp/mobilink" for synchronization, just as you would if connecting to SAP Mobile Server through a Relay Server which is not installed at the default location. The difference is that you do not include a FarmId for the reverse proxy.

Note: When using an Apache server as a reverse proxy without SAP Hosted Relay Server to proxy Object API Applications against SAP Mobile Server, if a custom URL suffix is used, clients should specify a custom URL suffix including a trailing forward slash "/". For example, "/myApp/" instead of "/myApp". If not, the client may report connection failures.

See also

- *Registering an Application* on page 33
- *Application APIs* on page 85

Communicating with SAP Mobile Server Through a Reverse Proxy

Connect to SAP Mobile Server through a Reverse Proxy using Relay Server or the Apache Web server.

The Object API application can communicate with SAP Mobile Server:

- Directly
- Through Relay Server by specifying:
  - Just the FarmID
  - Just the URL suffix
  - Both the FarmID and URL suffix
- Through a Reverse Proxy by specifying only the URL suffix - you can configure the reverse proxy in such a way that the device communicates to http://server:port/customcontext/tm or /tm2 when communicating with SAP Mobile Server. "FarmID" has no meaning when using a Reverse Proxy. In this case use only the URL suffix and leave "FarmID" empty.

  Note: You can also use the custom context with the Relay Server when using Apache.

Connecting to SAP Mobile Server through a Reverse Proxy using Relay Server:

```
// Register:
    Application.getInstance().getConnectionProperties().setUrlSuffix
       ("/ias_relay_server/client/rs_client.dll/$messaging-farmId$")
    // (e.g. /ias_relay_server/client/rs_client.dll/mega-vm008.msg)
```

```
//Synchronize:

TestDB.getSynchronizationProfile().getStreamParams().setUrl_Suffix
       ("/ias_relay_server/client/rs_client.dll/$replication-farmId
$")
    // (e.g. /ias_relay_server/client/rs_client.dll/mega-vm008.rep)
```

Connecting to SAP Mobile Server through a Reverse Proxy using Apache Web server as proxy server:

```
Add to the httpd.conf file:
content: Listen 80 <VirtualHost *:80>
ServerName proxy-server
<Location /app1/>
ProxyPass http://sup-server:5001/ ProxyPassReverse
http://sup-server:5001/
</Location>
<Location /app2/>
ProxyPass http://sup-server:2480/
ProxyPassReverse http://sup-server:2480/
</Location>
</VirtualHost>

// Register:
Application.getInstance().getConnectionProperties().setUrlSuffix("/
app1");

//Synchronize:
TestDB.getSynchronizationProfile().getStreamParams().setUrl_Suffix(
"/app2");
```

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 start-ups. The `registerApplication` method automatically starts the connection to complete the registration process.

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 SAP Mobile Server
app.setApplicationIdentifier("SMP101");
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
...

SMP101DB.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

- *Setting Up Application Properties* on page 30
- *Application APIs* on page 85

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 UltraLite®J 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. Retrieve the connection profile object using the SAP Mobile Platform database's `getConnectionProfile` method.

```
// Initialize the device database connection profile (if needed)
ConnectionProfile connProfile = SMP101DB.getConnectionProfile();

// Store the database in an SD card
connProfile.setProperty("databaseFile",
android.os.Environment.getExternalStorageDirectory().getPath() + "/
SMP1011_0.ulj");

// encrypt the database
connProfile.setEncryptionKey("your encryption key"); //Encryption
key can be of arbitary length, but generally the longer, the better.
// You can also automatically generate a encryption key and store it
```

```
inside a data vault.

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

An application can have multiple threads writing to the database during synchronization by enabling the connection profile property, `allowConcurrentWrite`. Setting the property to "true" allows multiple threads to perform create, read, update, or delete operations at the same time in a package database. For example:

```
SMP101DB.getConnectionProfile().setProperty("allowConcurrentWrite",
"true");
```

Note: Multiple threads are allowed to write to the database at the same time. However, there will be errors when multiple threads write to the same row of one MBO. Avoid writing to the same MBO row in your application.

See also

- *ConnectionProfile* on page 121

Setting Up Connectivity

Store connection information to the SAP Mobile Server data synchronization channel.

See also

- *Creating and Deleting a Device's Local Database* on page 36

Setting Up the Synchronization Profile

You can set SAP Mobile 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 SAP Mobile 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 SAP Mobile Server to connect to the SAP Mobile 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 SAP Mobile Platform database's `getSynchronizationionProfile` method.

   ```
   ConnectionProfile cp = SMP101DB.getSynchronizationProfile();
   ```

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

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

See also

- *Synchronization Profile* on page 124

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.

1. Connect to the generated database by calling the generated database instance's `openConnection` method.

   ```
   SMP101DB.openConnection();
   ```

   If the database does not already exist, the `openConnection` method creates it.

2. Optionally, you can include code in your application to check if an instance of the generated database exists by calling the generated database instance's `databaseExists` method.

   If an instance of the generated database does not exist, call the generated database instance's `createDatabase` method.

   ```
   if (!SMP101DB.databaseExists())
       {
           SMP101DB.createDatabase();
       }
   ```

3. When the local database is no longer needed, delete it by calling the generated database instance's `deleteDatabase` method.

   ```
   SMP101DB.deleteDatabase();
   ```

See also

- *Setting Up Connectivity* on page 35

Logging In

Use online authentication with the server.

Authenticate the user for data synchronization by calling the generated database API `onlineLogin` method.

Use the `SynchronizationProfile` to store the username and password.

```
ConnectionProfile syncProfile =
SMP101DB.getSynchronizationProfile();
syncProfile.setUserName("user");
syncProfile.setPassword("password");
SMP101DB.onlineLogin();
```

Turn Off API Logger

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

```
SMP101DB.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 SAP Mobile 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. To 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 42
- *Callback and Listener APIs* on page 171

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 register 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:
   ```
   SMP101DB.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)
   {
      SMP101DB.registerCallbackHandler(new MyCallbackHandler());
      GenericList<SynchronizationGroup> sgs = new
GenericList<SynchronizationGroup>();
      sgs.add(SMP101DB.getSynchronizationGroup("sg1"));
      sgs.add(SMP101DB.getSynchronizationGroup("sg2"));
      SMP101DB.beginSynchronize(sgs, "my test synchronization
context");
   }
}

class MyCallbackHandler extends DefaultCallbackHandler
{
   //The onSynchronize method overrides the
   //onSynchronize method from 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())
      // The application is waiting for input from the user.
      // This section demonstrates that you can stop the
synchronization or
      // let it proceed depending on the status of the application.
      {
         case SynchronizationStatus.STARTING:
            if (waitForMoreChanges())
            {
               return SynchronizationAction.CANCEL;
            }
            else
            {
               return SynchronizationAction.CONTINUE;
            }
         default:
            return SynchronizationAction.CONTINUE;
      }
   }
}
```

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 in the background. 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.

When SAP Mobile Platform does an `update` operation replay, if the primary key or foreign key of the MBO is generated by the EIS and the MBO's content coming from the device has no primary key or foreign key, the SAP Mobile Server loads the primary key or foreign key from the CDB to merge the incoming values with the CDB content so that a full row (graph) can be communicated to the EIS.

```
oneMBO mbo = new oneMBO();
mbo.setXX(xx);
....
mbo.create();
mbo.submitPending();
mbo.setXX(yy);
....
mbo.update();
mbo.submitPending();
DBClass.synchronize()
```

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:

```
SMP101DB.getSynchronizationProfile().setAsyncReplay(false);
```

When the application is connected (by `Application.startConnection()` or `Application.registerApplication`), it may receive background notifications and trigger a synchronize or other database operation. If you try to delete the database, you may receive database exceptions.

Before deleting the database, stop the application connection (`Application.stopConnection()`).

You can specify an upload-only synchronization where the client sends its changes to the server, but does not download other changes from the server. This type of synchronization conserves device resources when receiving changes from the server.

```
public static void
beginSynchronize(com.sybase.collections.GenericList<com.sybase.pers
istence.SynchronizationGroup> sgs,Object context, boolean
uploadOnly)
```

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

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.

```
SMP101DB.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 37

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*.

See also

- *Specifying Personalization Parameters* on page 45
- *Synchronization APIs* on page 131
- *Specifying Synchronization Parameters* on page 45

Configuring Data Synchronization Using SSL Encryption

Enable SSL encryption by configuring the synchronization HTTPS port.

1. In the left navigation pane of SAP Control Center for SAP Mobile 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 SAP Mobile 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. Set the synchronization parameters if there are any.

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

2. Make a blocking synchronize call to SAP Mobile Server to pull in all MBO data:

```
SMP101DB.synchronize();
```

3. 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
}
```

4. 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();
```

5. Submit the pending changes. The changes are ready for upload, but have not yet been uploaded to the SAP Mobile Server.

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

6. Use non-blocking synchronize call to upload the pending changes to the SAP Mobile 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(SMP101DB.getSynchronizationGroup("default")); // Customer
MBO is in "default" sync group
SMP101DB.beginSynchronize(sgs, "mycontext");
```

Enabling Change Notifications

A synchronization group can enable or disable its change notifications.

By default, change notifications are disabled for synchronization groups. To enable change notifications, you must synchronize, then call the `SynchronizationGroup` object's `setEnableSIS` method.

```
com.sybase.persistence.SynchronizationGroup sg =
SMP101DB.getSynchronizationGroup("PushEnabled");

if (!sg.getEnableSIS())
{
  sg.setEnableSIS(true);
  sg.setInterval(2);
  sg.save();
  SMP101DB.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 =
   SMP101DB.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 SAP Mobile Server:

   ```
   SMP101DB.synchronize();
   ```

See also

- *Synchronizing Applications* on page 43
- *Personalization APIs* on page 129

Specifying Synchronization Parameters

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

Note: The `getSynchronizationParameters` method has been deprecated.

Assign the synchronization parameters of an MBO before a synchronization session. The next synchronize sends the updated synchronization parameters to the server.

1. List all the synchronization parameters.

   ```
   com.sybase.collections.GenericList<CustomerSubscription> r =
   Customer.getSubscriptions();
   ```

2. Add synchronization parameters. This call adds and saves the synchronization parameters:

   ```
   CustomerSubscription sp = new CustomerSubscription();
   sp.setName("example");
   Customer.addSubscription(sp);
   ```

3. Synchronize to download the data:

   ```
   SMP101DB.synchronize();
   ```

For DOE-based applications, call `SMP101DB.beginSynchronize()` to download the data.

See also

- *Synchronizing Applications* on page 43
- *Synchronization APIs* on page 131

Subsequently Starting an Application

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

Starting an application on subsequent occasions:

1. Use the `getRegistrationStatus` API in the `Application` class to determine if the application has already been registered. if it has been registered, then only perform the following steps:
   a. Get the application instance.
   b. Set the `applicationIdentifier`. The `applicationIdentifier` must be the same as the one used for initial registration.
   c. Initialize the generated package database class with this application instance.

   Note: Once the application is registered, changes to any of the application connection properties do not take effect. To modify the connection properties, unregister the application, change the connection properties and then register again. Unregistering the application also removes the user from the server.
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 using the existing connection parameters and registration information.
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
SMP101DB.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 85

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 181
- *Object Queries* on page 47
- *Dynamic Queries* on page 48
- *MBOs with Complex Types* on page 49
- *Relationships* on page 49

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<SMP101.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<SMP101.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 47
- *Query APIs* on page 181

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 = SMP101DB.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 47
- *Query APIs* on page 181

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, SAP Mobile 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 SAP Mobile Server and get replayed.

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

See also

- *Accessing MBO Data* on page 47
- *Query APIs* on page 181

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 47
- *Query APIs* on page 181

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 192

Creating, Updating, and Deleting MBO Records

Perform create, update, and delete operations on the MBO instances that you have created.

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();
   SMP101DB.synchronize();
   ```

   For DOE-based applications, omit `SMP101DB.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();
   SMP101DB.synchronize();
   ```

   For DOE-based applications, omit `SMP101DB.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();
SMP101DB.synchronize();
```

For DOE-based applications, omit `SMP101DB.synchronize;` above.

For an object tree with MBOs in a composite (cascading) relationship, `submitPending` submits changes found in the entire hierarchy. If each MBO in the hierarchy has its own CUD operations, the submitted object tree replays in this order:

- Create and Update: a preorder traversal, i.e. parent -> left child -> right child. That is, create the parent before the children.
- Delete: a postorder traversal, i.e. left child ->right child->parent.

Left and right in this context means from the first child in the children list to the last child. For a tree with multiple operation types, for example root (update) and two children (one create and one update) and each child has two children, the order of the operation is: root (update), child one(create), children of child one(create), children of child two (delete), child two (delete).

See also

- *Operations APIs* on page 192

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 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:

   ```
   SMP101AddOperation op = new SMP101AddOperation(); //Convention 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();
   SMP101DB.synchronize();
   ```

   With DOE-based applications, omit `SMP101DB.Synchronize();` above.

See also

- *Operations APIs* on page 192

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, the MBOs enter a replay pending state. The next synchronization will submit those changes to the EIS.

Note that **submitPendingOperations** APIs are expensive. SAP 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. SAP 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. SAP 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.
5. For a single MBO, you must call the `refresh()` method of the MBO instance before you use this instance again.

   ```
   customer.create();
   customer.submitPending();
   ```

```
// must call refresh() here
customer.refresh();
customer.update();
customer.submitPending();
```

6. For related MBOs, you must call the `refresh()` method of the MBO instance before you use this instance again, even if the MBO's child or parent has called `submitPending`.

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. Use the `closeConnection` method to close all database connections for this package and release all resources allocated for those connections. This is recommended to be part of the application shutdown process.

Debugging Runtime Errors and Performance Analysis

To handle occurrences of exceptions and special conditions that change the normal flow of the program execution, you must perform error handling.

End to End Tracing

End to end tracing enables an application developer and end user to trace a request that is sent from the client to the back-end. This spans the entire landscape where you can derive a correlation of traces at the client, server and back-end.

These correlated traces help in performance analysis and are centrally monitored on SAP Solution Manager. These are displayed as reports where you can extract information on failure of delivering a request, time taken for a request to reach a component and so on.

On the client side, the client framework enables an application developer to switch on the trace for messages. The client traces the request at predefined points and all these transactions/ requests are recorded in a Business Transaction XML. Additionally, the client maintains a unique identifier in the HTTP header called the SAP Passport that is used to correlate traces

across various components. This Business Transaction XML can later be uploaded to the SAP Solution Manager which is a central location to correlate all logging information.

Using Tracing APIs

Use these APIs to enable the application user to use End-to-End tracing.

The API consists of the following interfaces or classes:

- **com.sybase.mobile.e2etrace.util.E2EtraceService –** A public interface for use by the application's user interface developers.
- **com.sybase.mobile.e2etrace.util.E2EtraceLevel –** Defines an enumeration of the trace levels that you can set to a passport. Trace levels control the amount of logging done on the server side.

Getting an Instance of the E2E Trace Service

Get an instance of the E2ETraceService interface.

```
E2ETraceService traceService = E2ETraceServiceImpl.getInstance();
```

Initializing the Trace

Set the trace level and start the trace. The SAP Mobile Server administrator sets the trace level from SAP Control Center.

Set the passport trace level to one of the following values.

Trace Level	Description
0 (NONE)	0 (NONE) Do not use. Not Supported. (Specific to trace analysis on the client. No traces are triggered on the server.)
1 (LOW)	Corresponds to response time- distribution analysis. This helps to analyse the time taken on each server component.
2 (MEDIUM)	Corresponds to performance analysis. Performance traces are triggered on the server side. Example: Introscope Transaction Trace, ABAP Trace, SQL Traces and so on.
3 (HIGH)	Corresponds to functional analysis.

```
...buttonClickEvent()
{
  E2ETraceLevel level = null;
    If(val == 0)
      level = E2EtraceLevel.NONE;
    else if(val == 1)
      level = E2EtraceLevel.LOW;
...
//Get an instance of the E2E trace handler
E2ETraceService e2eTraceService = E2ETraceServiceImpl.getInstace();
```

```
//Set the trace level to the E2ETraceService instance
e2eTraceService.setTraceLevel(level);
//Call the 'startTrace' method of the E2E trace service
...
e2eTraceService.startTrace ();
}
```

When you call the `startTrace()` method, the `E2ETraceService` initializes the trace and sets appropriate flags to indicate the trace has started. The method may perform other tasks as required by SAP's BTX API, such as getting a handle to the BTX writer from the BTX API.

Stopping the Trace

Stop appending trace data to the business transaction (BTX) and finish creating the BTX.

The `stopTrace()` method also retrieves the BTX byte array from the BTX writer and returns it to the calling code for further use (upload). Because the `stopTrace()` call clears the BTX from memory, you must make sure to save the BTX for further use, such as uploading the trace.

Uploading the BTX

Upload the business transaction to the server.

Upload the business transaction by calling `uploadTrace(byte[] btx)` and passing the BTX byte array. The method returns true if the upload succeeds, otherwise it throws an `E2ETraceUploadException`.

Call this blocking method in a separate thread other than the main application thread.

```
//ensure this blocking call gets executed in a separate thread
try
{
boolean status = traceService.upload(btx);
}
catch(E2EtraceUploadException eue){ }
```

Tracking KPI

Access performance libraries for tracing or collecting key performance indicators (KPIs).

User interactions are measured in intervals of these types: HttpRequest, PersistenceRead, PersistenceWrite, SubmitPending, CancelPending, and Transaction. All intervals measure Wallclock Time, CPU Time, and Memory Max.

Specific interval types measure some additional KPIs:

- HttpRequest
  - Roundtrips
  - Total Bytes
  - Sent Bytes

    - Received Bytes
    - Total Packets
    - Sent Packets
    - Received Packets
- PersistenceRead
    - PersistenceReads
- PersistenceWrite
    - PersistenceWrites

After the interaction is stopped, a summary log in `csv` format and a detailed log in txt format is written to the device.The summary log contains sums of each of the KPI types. For example, total Wallclock Time, total CPU Time, total number of roundTrips, total number of PersistenceRead, total CPU Time of PersistenceWrite, and so on. The detailed log also contains a summary line, as well as KPI values for each interval.

The administrator can invoke a Get Trace request through SAP Control Center to send the performance log to the server domain log.

To start collecting performance metrics, call the `startInteraction` method:

```
public void startInteraction(String interactionName)
```

To stop collecting performance metrics and output a summary to the reporting target, call the `stopInteraction` method:

```
public void stopInteraction();
```

Example of application interactions for collecting KPI:

```
// get the instance
PerformanceAgentService pa =
PerformanceAgentServiceImpl.getInstance();
pa.startInteraction("Interaction 1");
    // application interaction
    // ...
    // ...
pa.stopInteraction();

    pa.startInteraction("Interaction 2");
    // application interaction
    // ...
    // ...
    pa.stopInteraction();
```

The following limitations apply:

- On Android devices, Memory Max can only be measured on devices running Android 2.x.
- On Android devices, the HttpRequest KPIs (Total Packets, Sent Packets, Received Packets) are not supported by all Android devices.

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. Unregister the application by invoking the `Application` instance's `unregisterApplication` method.

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

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

   ```
   SMP101DB.deleteDatabase();
   ```

Recovering From SAP Mobile Server Failures

Add application code to check for and recover from from SAP Mobile Server failures.

It is highly recommended that you add a catch call to all synchronize methods (synchronize(), begingSynchronize(), and so on) within your applications to allow the application to recover if SAP Mobile Server fails and needs to be restored from an older database. If not, you may have to reinstall the application manually for all users so they can resynchronize with SAP Mobile Server.

See *Restoring from an Older Backup Database File (Data Loss)* in the *System Administration Guide* for information about SAP Mobile Server recovery.

As a best practice, and not included in these examples, application developers should include code that informs mobile application users about:

- What is going to happen (for example, reregistering, recreating the local database, and so on). And,
- The reason for the action (for example, lost registration, server is restored, and so on).

And prompt them for confirmation before executing the code.

When the SAP Mobile Server is restored to a previous state, it may be inconsistent with the state of the client. For example:

- The client synchronizes with the server after the database is backed up. In this case the client cannot synchronize successfully with the server after the database is restored.
- The client registers with the server after the database is backed up. In this case the client registration is lost when the database was restored.

The following sample code illustrates how the client can recover from these errors.

1. After SAP Mobile Server is restored, client application connection information may be lost if the registration was created after the database was backed up. This client application calls `startConnection` to connect to the server, the `onConnectionStatusChanged` callback returns error code 580 with a message that authentication failed. The user can reregister the application with the `ApplicationCallback` implementation after encountering this error code. If the server is restored to a point-in-time when the client application has registered, the application runs as normal without receiving this error code. These examples illustrate both automatic and manual registration recovery.

   a) Automatic registration recovery:

```
public void startApplication( )
  {
      Application app = Application.getInstance();
      Application.getInstance().setApplicationCallback(new
MyApplicationCallback());
      try
      {
           ConnectionProperties connProperties =
app.getConnectionProperties();
           connProperties.setServerName (“mega-vm008” );
           connProperties.setPortNumber (5001);
           connProperties.setLoginCredentials(new
com.sybase.persistence.LoginCredentials(“test@admin”,
“test123”));
           if (app.getRegistrationStatus() ==
RegistrationStatus.UNREGISTERED)
           {
               app.registerApplication(100); // or call
app.registerApplication();
           }
           else
           {
               app.startConnection(100); // or call
app.startConnection();
           }
      }
      catch (ApplicationRuntimeException ex)
      {
          System.out.println(ex);
      }
      catch (ApplicationTimeoutException ex)
      {
          System.out.println(ex);
      }
      while(app.getConnectionStatus() !=
ConnectionSatus.CONNECTED || app.getRegistrationStatus() !=
                RegistrationStatus.REGISTED)
      {
            try
           {
                Thread.sleep(100);
           }
```

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

       }

  }
 public class MyApplicationCallback  extends
com.sybase.mobile.DefaultApplicationCallback
  {
      boolean callFlag = false;
     //override this method to check if need to reregister
      public  void onConnectionStatusChanged(int
connectionStatus, int errorCode, String errorMessage)
      {
          if (errorCode == 580 && !callFlag)
          {
            //this callback will be invoked multiple times when
this error occures, but we just call once reregister, so set the
             //callFlag to be true.
              callFlag = true;
              Thread registerThread = new Thread("reregister")
               {
                    public void run()
                   {
                   //do not unregister application, because the
application connection info has been deleted from server side.
We can
                       //call register application directly.

Application.getInstance().registerApplication();
                   }
               }.start();
          }
      }

  }
```

b) Manual registration recovery. If error code 580 is encountered, the Administrator must first manually register the application in SAP Control Center, or else the reregistered application fails the first time. Manual registration requires setting of the activation code:

```
public void startApplication( )
{
    Application app = Application.getInstance();
    Application.getInstance().setApplicationCallback (new
MyApplicationCallback());

    try
    {
         ConnectionProperties connProperties =
app.getConnectionProperties();
         connProperties.setServerName("mega-vm008" );
         connProperties.setPortNumber (5001);
```

```
         connProperties.setActivationCode ("100" );
         connProperties.setLoginCredentials (new
com.sybase.persistence.LoginCredentials("test@admin", null));
         if (app.getRegistrationStatus() ==
RegistrationStatus.UNREGISTERED)
         {
             app.registerApplication(100); // or call
app.registerApplication();
         }
         else
         {
             app.startConnection(100); // or call
app.startConnection();
         }
    }
    catch (ApplicationRuntimeException ex)
    {
        System.out.println(ex);
    }
    catch (ApplicationTimeoutException ex)
    {
        System.out.println(ex);
    }
    while(app.getConnectionStatus() !=
ConnectionSatus.CONNECTED || app.getRegistrationStatus() !=
              RegistrationStatus.REGISTED)
    {
         try
         {
              Thread.sleep(100);
         }
         catch (Exception ex)
         {
             System.out.println(ex);
        }
    }

  }
public class MyApplicationCallback  extends
com.sybase.mobile.DefaultApplicationCallback
{
    boolean callFlag = false;
   //override this method to check if need to reregister
    public  void onConnectionStatusChanged(int
connectionStatus, int errorCode, String errorMessage)
    {
        if (errorCode == 580 && !callFlag)
        {
           //this callback will be invoked multiple times when
this error occures, but we just call once reregister, so set the
           //callFlag to be true.
            callFlag = true;
            Thread registerThread = new Thread("reregister")
             {
                  public void run()
                  {
```

```
                    //do not unregister application, because the
application connection info has been deleted from server side.
We can
                       //call register application directly.

Application.getInstance().registerApplication();
                    }
                }.start();
            }
        }
 }
```

2. Client Application RBS synchronization recovery.

   If the server is restored to a point-in-time of an application's previous synchronization, the client synchronization gets `com.sybase.persistence.SynchronizeException` with an error code of SQLE_UPLOAD_FAILED_AT_SERVER. This error code indicates the need to recover the client database. If the server is restored to a point-in-time of the application's last synchronization or the application has never synchronized, the client application can synchronize as normal without the exception. For example:
   - Time1: application registered and has not synchronized.
   - Time2: application synchronized for the first time.
   - Time3: application synchronized for the second time.
   - Time4: application synchronized for the last time.

   If the server is restored to time1 or time4, the client can synchronize successfully. If the server is restored to time2 or time3, client synchronization fails with `com.sybase.persistence.SynchronizeException`. You have three methods to recover the client database and synchronize successfully again:
   a. Before synchronization recovery, the application needs to complete application registration recovery if necessary.
   b. Once the client application starts, the application checks if the last recovery failed in the middle by checking the saved flag. If the last recovery failed, the application needs to resume the recovery first.
   c. Mark the recovery state. For example, the application can save the recovery state to a file. In recovery method two, the old client database is copied and used as a recovery state flag.

   a) Recreate database without copying old data (all data lost)

      This is the simplest recovery method, but the old data, such as synchronization parameters, SIS, and Local MBOs are not copied to the new database. The application user needs to reenter them in the application's GUI.

```
try
 {
      End2end_rdbDB.synchronize();
 }
 catch (Exception ex)
 {
    //if meet this error, the server has been restored, we need
```

```
to recover client database
       if (ex instanceof
com.sybase.persistence.SynchronizeException)
       {
          if (((com.sybase.persistence.SynchronizeException)
ex).getErrorCode() ==

com.sybase.persistence.SynchronizeException.SQLE_UPLOAD_FAILED
_AT_SERVER)
          {
               recoverClientDatabase();
          }
       }
    }

    private void recoverClientDatabase()
    {
        setRecoveringInPlaceFlag();  //Like save a flag into
FileSystem
        End2end_rdbDB.closeDBConnection();
        End2end_rdbDB.deleteDatabase();
        cleanRecoveringInPlaceFlag();
    }
```

b) Recreate database and copying old database (local transaction lost)

Copies old database data to a new database; this example includes personalization keys, subscription information, SIS info, local BO. Unsubmitted transactions like an MBO's pending state are lost. This sample code checks if a copy of the database is available to determine if a recovery was interrupted.

```
if(isRecoverFailed())
{
     recoverClientDatabase();
}
else
{
     try
     {
        End2end_rdbDB.synchronize();
     }
     catch (Exception ex)
     {
        //if meet this error, the server has been restored, we
need to recover client database
        if (ex instanceof
com.sybase.persistence.SynchronizeException)
       {
           if (((com.sybase.persistence.SynchronizeException)
ex).getErrorCode() ==

com.sybase.persistence.SynchronizeException.SQLE_UPLOAD_FAILED
_AT_SERVER)
          {
               recoverClientDatabase();
          }
       }
```

```
     }
}

private void isRecoverFailed()
{
    String dbFile =
End2end_rdbDB.getConnectionProfile().getProperty("databaseFile
");
    String recoverDbFile = dbFile + ".recover.ulj";
    if (new File(recoverDbFile).exists())
    {
       //todo
       //implement to copy recoverDbFile content to recover
dbFile
       return true;
    }
     return false;
}

private void recoverClientDatabase()
 {
    String dbFile = End2end_rdbDB.
getConnectionProfile().getProperty("databaseFile");
    String recoverDbFile = dbFile + ".recover.ulj";
    //todo
    //implement to copy dbFile content to recover recoverDbFile

    //retrieve all the subscriptions from client database
    GenericList<CustomerWithParamSubscription>
_customerWithParamSubscriptions =
    CustomerWithParam.getSubscriptions();
    GenericList<SISSubscription> _sisSubs =
SISSubscription.findAll();
    GenericList<String> syncedPublication = new
GenericList<String>();

   // check all the synchronization group, if is synchronized,
add to new sync group to synchronize
    if (End2end_rdbDB.isSynchronized("synchronizationGroup"))
    {
        syncedPublication.add("synchronizationGroup ");
    }
    //retrieve  all local BO from client database
    GenericList< LocalMbo > localBoList = LocalBo.findAll();
    End2end_rdbDB.closeDBConnection();
    //subscribe with new database file
    End2end_rdbDB.deleteDatabase();
    End2end_rdbDB.subscribe();

    //merge old local BO data to new database
    for(LocalBo lc :  localBoList)
    {
        LocalBo localBo = new LocalBo ();
        localBo.copyAll(lc);
        localBo.create();
    }
```

```
    //add all the subscriptions from old database to new
database
    for (CustomerWithParamSubscription sub :
_customerWithParamSubscriptions)
    {
        CustomerWithParam.addSubscription(sub);
    }
    for (SISSubscription sub :  _sisSubs)
    {
        ISynchronizationGroup sg =
End2end_rdbDB.getSynchronizationGroup(sub.getSyncGroup());
        sg.setEnableSIS (sub.getEnable());
        sg.save();
    }
    //do sync
    String syncgroups = "";
    for(String pub :  syncedPublication)
    {
        syncgroups += pub + ",";
    }
    syncgroups = syncgroups.substring(0, syncgroups.length()
-1 );
    End2end_rdbDB.synchronize(syncgroups);

    //finally delete the backup recover database file
    new File(recoverDbFile).delete();
}
```

c) Resending local transaction

This is a complete recovery method. Both methods above lose the local transaction. To prevent the lose of the local transaction when encountering the `SQLE_UPLOAD_FAILED_AT_SERVER` exception, the SAP Mobile Server Administrator must remove the client remote id information. The Administrator can locate the remote id from the server's `mlsrv_err.log` and call the `ml_delete_remote_id` procedure in the CDB to remove the remote id. The user can then continue to synchronize using the old database to upload all pending operations. But once uploaded, the user\application must recreate the database using either of the two methods described above, and must **not** reuse the old database anymore. The `mlsrv_err.log` logs remote id errors similar to this:

```
 I. 2013-04-14 14:13:39. <3> The sync sequence ID in the
consolidated database:
        95bd47691098419cbf8539e8151bcf00; the remote previous
sequence ID:
        95bd47691098419cbf8539e8151bcf97, and the current
sequence ID:
       401be536e6e7417fb01b196276ec11c2E. 2013-04-14 14:13:39.
<3> [-10400] Invalid sync sequence ID for remote ID
        'ed2ae448-a597-4f17-ad72-c6c61a6075a5'
```

3. Client application RBS beginSynchronize recovery

`beginSynchronize` is an async pattern, requiring the user to override the `com.sybase.persistence. DefaultApplicationCallbackHandler`

class `onSynchronize` method to check the `SQLE_UPLOAD_FAILED_AT_SERVER` error using the same three methods as described above to recover the client database. This sample code uses the second method and implements the `AsyncCallbackHandler`:

```
if(isRecoverFailed())
{
     recoverClientDatabase();
}
else
{
    GenericList<String> syncList = new GenericList<String>();
    syncList.add("default");
    synchronize(syncList);
}

 private void isRecoverFailed()
 {
    String dbFile =
End2end_rdbDB.getConnectionProfile().getProperty("databaseFile");
    String recoverDbFile = dbFile + ".recover.ulj";
    if (new File(recoverDbFile).exists())
    {
       //todo
       //implement to copy recoverDbFile content to recover dbFile
       return true;
    }
    return false;
}

private void synchronize(GenericList<String> syncGroup)
{
    AsyncCallbackHandler callback = new AsyncCallbackHandler();
    GenericList<ISynchronizationGroup> sgs = new
    GenericList<ISynchronizationGroup>();
    for(String sg : syncGroup)
    {
        sgs.add(End2end_rdbDB .getSynchronizationGroup(sg));
    }
    callback.userContext = System.nanoTime() + "";
    End2end_rdbDB.registerCallbackHandler(callback);
    End2end_rdbDB.beginSynchronize(sgs, callback.userContext);

    int waitCount = 0;
    while (!callback.asyncDone())
    {
        if (waitCount++ > maxWaitTime)
        {
           throw new Exception("Asyn relay test failed because no
response returned from server after waiting for 60 seconds.");
        }
        try
        {
            Thread.sleep(1000);
        }
        catch (Exception e)
```

```
        {
        }
    }

    try
   {
         Thread.sleep(4000);
   }
    catch (Exception e)
   {
   }

   if (callback.errorMessage != null)
    {
        throw new Exception(callback.errorMessage);
    }
    callback.userContext = null;
}

 private void recoverClientDatabase()
 {
    String dbFile = End2end_rdbDB.
getConnectionProfile().getProperty("databaseFile");
    String recoverDbFile = dbFile + ".recover.ulj";
    //todo
    //implement to copy dbFile content to recover recoverDbFile
    //retrieve all the subscriptions from client database
    GenericList<CustomerWithParamSubscription>
_customerWithParamSubscriptions =
    CustomerWithParam.getSubscriptions();
    GenericList<SISSubscription> _sisSubs =
SISSubscription.findAll();
    GenericList<String> syncedPublication = new
GenericList<String>();

   // check all the synchronization group, if is synchronized, add
to new sync group to synchronize
    if (End2end_rdbDB.isSynchronized("synchronizationGroup"))
    {
        syncedPublication.add("synchronizationGroup ");
    }
    //retrieve  all local BO from client database
    GenericList< LocalMbo > localBoList = LocalBo.findAll();
    End2end_rdbDB.closeDBConnection();
    //subscribe with new database file
    End2end_rdbDB.deleteDatabase();
    GenericList<String> syncList = new GenericList<String>();
    syncList.add("default");
    synchronize(syncList);

    //merge old local BO data to new database
    for(LocalBo lc : localBoList)
    {
        LocalBo localBo = new LocalBo ();
        localBo.copyAll(lc);
        localBo.create();
```

```
    }

    //add all the subscriptions from old database to new database
    for (CustomerWithParamSubscription sub :
_customerWithParamSubscriptions)
    {
        CustomerWithParam.addSubscription(sub);
    }
    for (SISSubscription sub : _sisSubs)
    {
        ISynchronizationGroup sg =
End2end_rdbDB.getSynchronizationGroup(sub.getSyncGroup());
        sg.setEnableSIS(sub.getEnable());
        sg.save();
    }
    //do sync
    synchronize(syncedPublication);
    //finally delete the backup recover database file
    new File(recoverDbFile).delete();
}

class AsyncCallbackHandler  extends  DefaultCallbackHandler
    {
         private volatile boolean asyncCompleted = false;
         private volatile boolean asyncUploaded = false;
         public volatile String userContext = null;
         public volatile String errorMessage = null;

         public boolean asyncDone()
        {
             return asyncCompleted;
        }

        public  SynchronizationAction
onSynchronize(GenericList<ISynchronizationGroup> groups,
                SynchronizationContext context)
       {
            Exception ex = context.getException();
             if (ex instanceof
com.sybase.persistence.SynchronizeException)
           {
              if (((com.sybase.persistence.SynchronizeException)
ex).getErrorCode() ==

com.sybase.persistence.SynchronizeException.SQLE_UPLOAD_FAILED_AT
_SERVER)
               {
                    recoverClientDatabase();
               }
           }

           if (context.getStatus() ==
SynchronizationStatus.ASYNC_REPLAY_UPLOADED)
           {
              if (!End2end_rdbDB.isReplayQueueEmpty())
              {
```

```
                throw new Exception("need sync  is not correct!");
              }
              asyncUploaded = true;
          }
          if (context.getStatus() ==
SynchronizationStatus.ASYNC_REPLAY_COMPLETED)
          {
              if (userContext != null && !
userContext.equals(context.getUserContext())) //Not for this
round
              {
                  return SynchronizationAction.CANCEL;
              }

            userContext = null;
            End2end_rdbDB.synchronize("so");
            if (!asyncUploaded)
            {
               errorMessage = "ASYNC_REPLAY_COMPLETED is received
without ASYNC_REPLAY_UPLOADED";
            }
            asyncCompleted = true;
            return SynchronizationAction.CANCEL;
        }
        return SynchronizationAction.CONTINUE;
     }
 }
```

4. MBS for DOEC client Application recovery

   MBS for DOEC clients run on C# and Android platforms. The registration recovery is the same as RBS registration recover. But the Subscribe recovery is different. There are two scenarios:

   - **SAP Mobile Server restored to a point-in-time where the client unsubscribed –** after SAP Mobile Server is restored, the client status is unsubscribed on the server-side, But the client status may be subscribed or unsubscribed. The client application should check the client subscription status; if subscribed, call unsubscribe(this leads to unsubscribe failure message because the server does not have client subscribe information), then call subscribe. If unsubscribed, call Subscribe. Sample code:

     ```
      If(MyDatabase.isSubscribed())
     MyDatabase.unsubscribe();
     _rh.waitForMessage("UnsubscribeFailure",
     "UnsubscribeSuccess");
     MyDatabase.subscribe();
     _rh.waitForMessage("SubscribeSuccess");
     ```

   - **SAP Mobile Server is restored to a point-in-time where the client subscribed –** after SAP Mobile Server is restored, the client status is subscribed on the server-side. In this scenario, the client can check if it is subscribed and, if not, call Subscribe to recover. After SAP Mobile Server is restored, the SAP Control Center Administrator can also delete the subscription from server-side. This time, SAP Mobile Server sends an unsubscribe message to the client application. When the client application is running, it receives the unsubscribe message, at which time the client generated

database calls `DB.cleanAllData()` to clean the client data and update the status to unsubscribed. The client application needs to call subscribe:

```
Thread.sleep(10 * 1000);
//sleep some time to receive server unsubscribe message when
application start up
If(!MyDatabase.isSubscribed())
MyDatabase.subscribe();
```

Testing Applications

Test native applications on a device or simulator.

For additional information about testing applications, see these topics in the Mobile Application Life Cycle collection:

- *Recommended Test Methodologies*
- *Best Practices for Testing Applications on a Physical Device*

See also

- *Developing the Application Using the Object API* on page 29
- *Localizing Applications* on page 81

Testing an Application Using a Emulator

Run and test the application on an emulator and verify that the application automatically registers to the SAP Mobile 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 the SAP Mobile Server using the default application connection template.
2. In SAP 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. The Pending Items column is used only for messaging applications.
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:

- SAP Mobile 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 SAP 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 SAP 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,requestId,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` – SAP Mobile 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 the SAP Mobile 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
SMP101DB.getConnectionProfile().enableTrace(true, true);
```

Server-Side Debugging

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

Problems on the SAP Mobile 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 SAP Mobile Server side:

- Check the SAP Mobile 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 SAP Control Center. See *SAP Control Center for SAP Mobile Platform* > *Administer* > *SAP Mobile Server* > *Server Log* > *SAP Mobile Server Runtime Logging* > *Configuring SAP Mobile Server Log Settings*.

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 *SMP_HOME*`\Servers\UnwiredServer\logs\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.

Improve Synchronization Performance by Reducing the Log Record Size

Improve synchronization performance and free SAP Mobile Server resources by deleting log records from SAP Mobile Server and the client when no longer needed.

A large log record table can negatively impact client synchronization performance. Each package contains a single log record table that consists of:

- **SAP Mobile Server operation replay logs –** downloaded to the device when the application synchronizes. SAP Mobile Server generates a log record if the operation replay fails, or succeeds but results in a warning.
- **Client logs generated by the application –** uploaded from the device to SAP Mobile Server for audit and logging purposes.

If the application and SAP Mobile Server do not delete these log records, the log record table continues to grow.

Unrestricted growth of the log record table eventually affects synchronization performance. You can view client log records from SAP Control Center; however, this displays only active log records (that is, those that have not been logically deleted). A logically deleted log record is marked for deletion but retained until the application downloads the delete record and deletes the copy from the device. Once SAP Mobile Server confirms that the application has

downloaded the delete, the inactive log record can be physically removed from SAP Mobile Server.

Determining the Log Record Size

Use Sybase® Central to query the database of a given SAP Mobile Server to determine the size of the log record.

Prerequisites

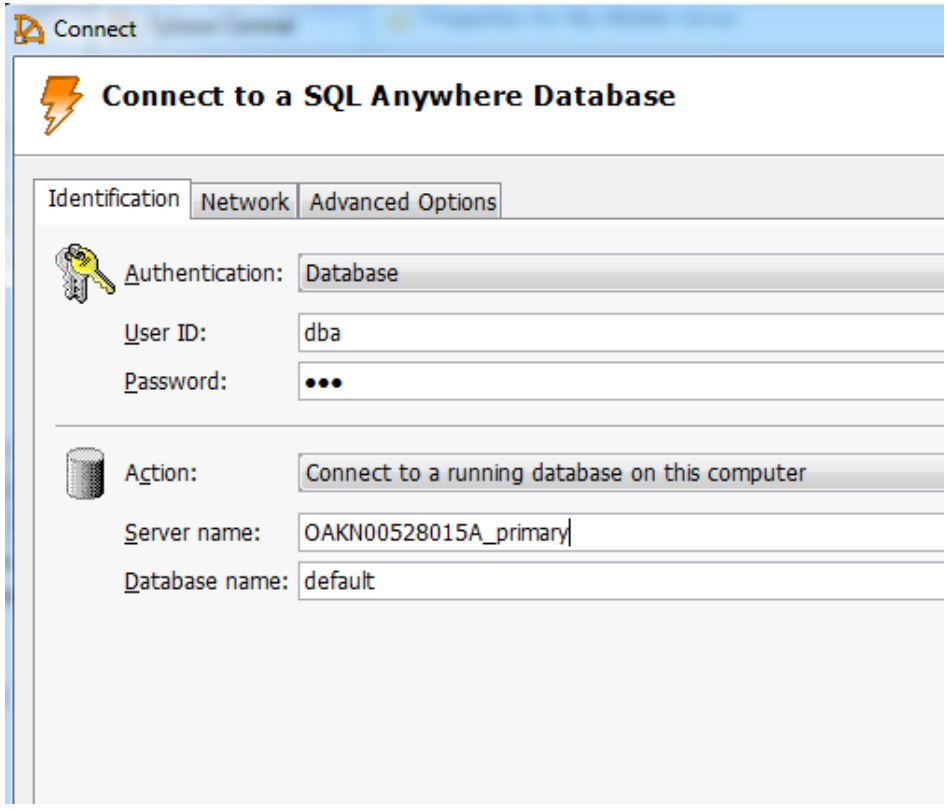
SAP Mobile Platform services must be running and at least one Mobile Application project deployed to SAP Mobile Server.

Task

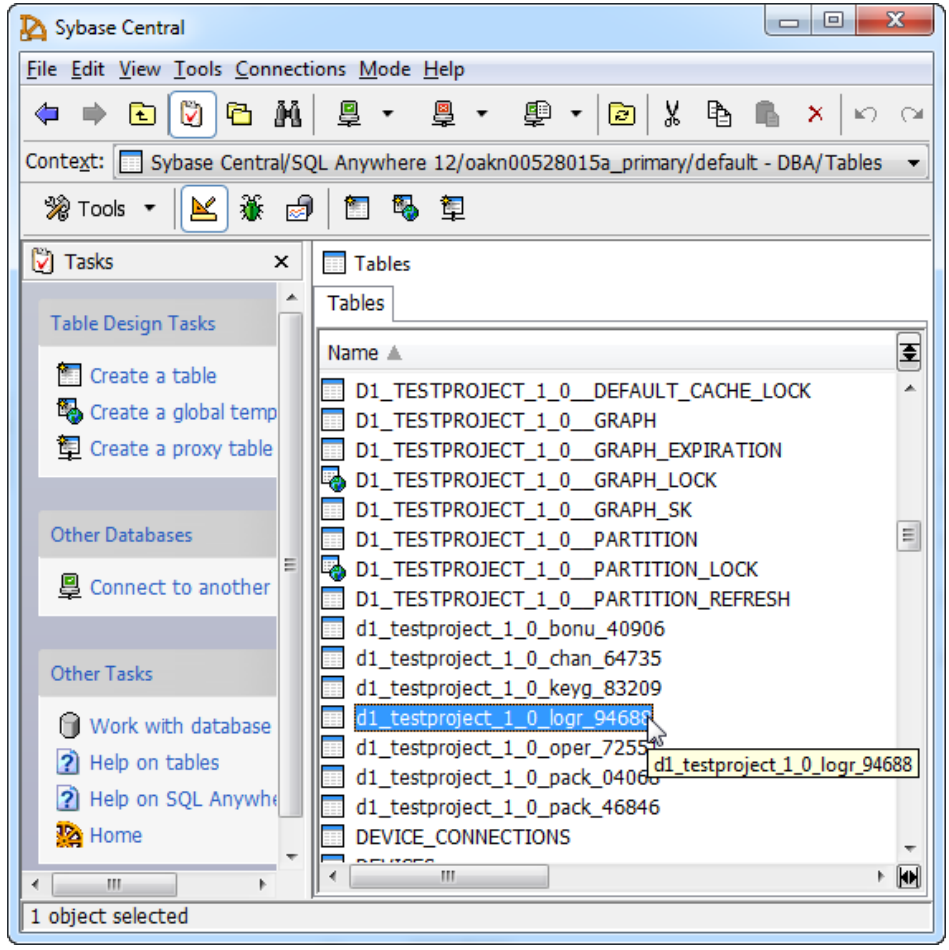
1. Launch Sybase Central (`scjview.exe`) to manage SQL Anywhere and Ultralite databases.

   The default installation location of the Sybase Central executable is *`SMP_HOME`* `\Servers\SQLAnywhere12\BIN32\scjview.exe`.
2. From Sybase Central connect to the database server by selecting **Connections > Connect with SQL Anywhere 12** .
3. Provide connection details and click **Connect**.

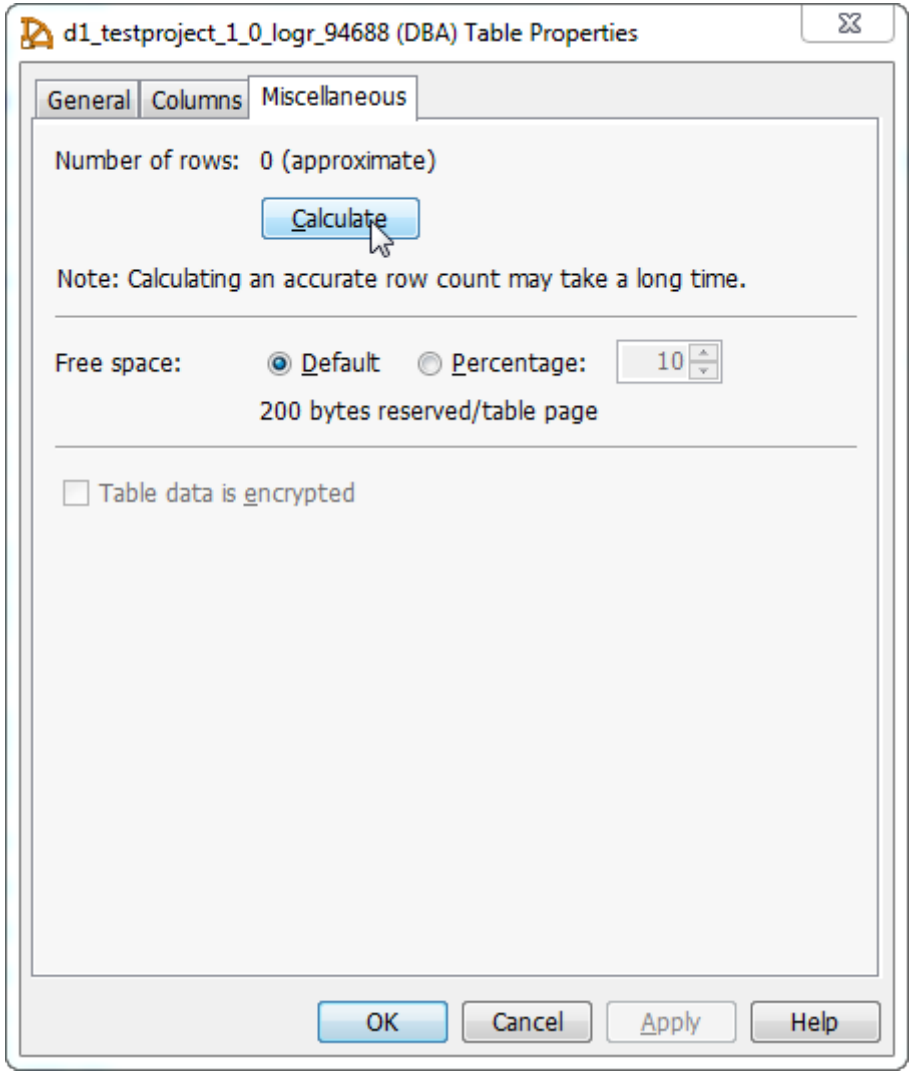
   For example, select **Connect to a running database on this computer** and enter:
   - **User ID and Password –** `dba` and `sql` respectively
   - **Server name –** *hostName*_primary
   - **Database name –** `default`



4. Double-click the **Tables** folder and search for the log record table. The log record name is typically *packageName*_logr… where *packageName* is the name of the deployed package.



5. Right-click the log record table and select **Properties**.
6. In the Properties dialog, select the **Miscellaneous** tab, then click **Calculate**.



The number returned includes logically deleted rows.The returned number of rows depends on the number of application users of the package, and the retention window setting. As a general guideline, the number of rows should be fewer than 10,000.

Reducing the Log Record Size

Use SAP Control Center to delete log record entries by setting a date range window.

The SAP Mobile Server does not remove any logically deleted rows until it receives confirmation that the device hosting the application has synchronized after the record is logically deleted from SAP Mobile Server.

1. Clean up the client log data:
   a) Expand **Domains** > **default** > **Packages**.
   b) Select *packageName* then select the **Client Log** tab.
   c) Select **Clean**, then enter starting and ending dates.

      The LOGICAL_DEL flag is set to true for records within the range.

      Note: Allow time for clients to synchronize. Logically deleted records are retained until the client synchronizes and downloads the delete records that clean up the client database. The length of time to wait for synchronization to complete depends on the clients' activities.

   d) Click **OK** to clean the client log data.
2. Clean the logically deleted records from SAP Mobile Server:
   a) Select the **General** tab.
   b) Select **Error Cleanup**.

      This starts a cleanup task that asynchronously removes all logically deleted records from clients that have performed a synchronization after the time specified in the Clean operation.

      For example, if the Clean operation is performed at 1:00am on Feb 27, all clients that synchronize after that time have their records physically removed. As a result, it takes time to reduce the size of the log record table.

      Note: Clean up the client log data (step one) during periods of low client activity: when a single transaction processing a large log record table is active, client synchronization is blocked, degrading client responses and performance. As a best practice, once the log record table has been cleaned to a reasonable size, schedule the clean/error cleanup tasks on a daily basis.

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 71

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 SAP Mobile 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 SAP Mobile Platform Client Object API consists of generated business object classes that represent mobile business objects (MBOs) that are designed and built in the SAP Mobile 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* > *Developing the Application Using the Object API*.

Client Object API Reference

Use the SAP Mobile Platform Client Object API Javadocs as a Client Object API reference.

Review the reference details in the Client Object API documentation, located in *SMP_HOME* `\MobileSDK23\ObjectAPI\apidoc`.

There is a subdirectory for `android`.

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

Note: Due to an Ultralite limitation, the first client object API call must be on the main thread in the application.

Application APIs

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

Note: SAP 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 29
- *Setting Up Application Properties* on page 30
- *Registering an Application* on page 33
- *Subsequently Starting an Application* on page 46

Application

Methods or properties in the `Application` class.

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` or `registerApplication`.

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 `SMP101`.

  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 SAP Mobile Server
app.setApplicationIdentifier("SMP101");
```

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)`.

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`.

The maximum length of the Application ID is 64 characters. The total length of the Application Connection ID cannot exceeds 128 characters. The Application Connection ID format is `deviceId__applicationId`. The `applicationId` separator is two underscores.

```
Application app = Application.getInstance();
// set Application ID - need to match as the server side Application
ID
app.setApplicationIdentifier("SMP101");
app.setApplicationCallback(new MyApplicationCallbackHandler());
ConnectionProperties props = app.getConnectionProperties();
props.setServerName("server.mycompany.com");
props.setPortNumber(5001);
LoginCredentials loginCred = new LoginCredentials("supAdmin",
"supPwd");
props.setLoginCredentials(loginCred);

SMP101DB.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.

As a best practice, if a timeout exception occurs in registerApplication or startConnection, the application should wait for the appropriate callback, and optionally add a user message to the application, "please wait" for example, instead of closing the application. This prevents a build up of start up requests by needlessly restarting the application which can adversely affect performance.

Wait for the application callback, such as onConnectionStatusChanged() if `ApplicationTimeoutException` is encountered when calling registerApplication (int timeout), instead of closing the application. This allows the application code to catch `ApplicationTimeoutException` and does not throw an exception.

```
try
{ Application.GetInstance().RegsiterApplication(100); }
catch(ApplicationTimeoutException ex)
{
while (Application.GetInstance().ConnectionStatus ==
ConnectionStatus.CONNECTING)
{ Thread.Sleep(100); }
}
```

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). 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`.

The maximum length of the Application ID is 64 characters. The total length of the Application Connection ID cannot exceeds 128 characters. The Application Connection ID format is `deviceId__applicationId`. The `applicationId` separator is two underscores.

```
Application app = Application.getInstance();
// set Application ID - need to match as the server side Application
ID
app.setApplicationIdentifier("SMP101");
app.setApplicationCallback(new MyApplicationCallbackHandler());
ConnectionProperties props = app.getConnectionProperties();
props.setServerName("server.mycompany.com");
props.setPortNumber(5001);
LoginCredentials loginCred = new LoginCredentials("supAdmin",
"supPwd");
props.setLoginCredentials(loginCred);

SMP101DB.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 SAP Mobile Server
app.setApplicationIdentifier("SMP101");
ApplicationCallback appCallback = new MyApplicationCallback();
app.setApplicationCallback(appCallback);
```

getApplicationCallback

Get the current callback handler.

Syntax

```
public ApplicationCallback getApplicationCallback();
```

Examples

- **Get the current ApplicationCallback handler**

```
ApplicationCallback currentCallback =
application.getApplicationCallback();
```

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)
```

Parameters

- **context –** The Android application 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()
  ```

Usage

If you delete an application from SAP Control Center, when the client application calls `startConnection()`, the following callback is triggered inside the `ApplicationCallback` handler:

```
void onConnectionStatusChanged(int connectionStatus, int errorCode,
String errorMessage);
errorCode = 580
errorMessage = "Error: 580 Message: 'TM
Error:InvalidAuthenticationParameters'"
```

To continue using the application, call `unregisterApplication()` to clean up the client state, and re-register using `registerApplication()`. You lose the previous

subscription on the server side. Delete the client database and perform another initial synchronization.

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). 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).

Returns

None.

Examples

- **Start the Application**

```
startConnection(timeout)
```

getConnectionStatus

Return current status of the mobile application connection.

Syntax

```
public int getConnectionStatus()
```

Returns

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

`ConnectionStatus` has the following possible values:

- **ConnectionStatus.CONNECTED –** The connection has been successfully started.
- **ConnectionStatus.CONNECTING –** The connection is currently being started.
- **ConnectionStatus.CONNECTION_ERROR –** The connection could not be started, or was previously started and subsequently an error occurred. Use `onConnectionStatusChanged` to capture the associated `errorCode` and `errorMessage`.
- **ConnectionStatus.DISCONNECTED –** The connection been sucessfully stopped, or there was no previous connection.
- **ConnectionStatus.DISCONNECTING –** The connection is currently being stopped.
- **ConnectionStatus.NOTIFICATION_WAIT –** The connection has been suspended and is awaiting a notification from the server. This is a normal situation for those platforms which can keep connections closed when there is no activity, since the server can reawaken the connection as needed with a notification.

Examples

- **Get the Application Connection Status**

```
getConnectionStatus()
```

getConnectionProperties

Retrieves the connection parameters from the application's connection properties instance. You must set connection properties before calling `startConnection`, `registerApplication` or `unregisterApplication`.

Syntax

```
public ConnectionProperties getConnectionProperties()
```

Parameters
None.

Returns

Returns the connection properties instance.

getApplicationSettings
Return application settings that have been received from the SAP Mobile Server after application registration and connection.

Syntax
```
public ApplicationSettings getApplicationSettings()
```

Returns
Application settings that have been received from the SAP Mobile Server.

Examples

- **Get the application settings**

```
 ApplicationSettings applicationSettings =
Application.getInstance().getApplicationSettings();
```

beginDownloadCustomizationBundle (java.io.OutputStream out)
Start downloading the default resource bundle associated with the application, and save it into an output stream.

The resource bundle is saved into the output stream that you provide. An application can only have one default resource bundle.

Syntax
```
public void beginDownloadCustomizationBundle (java.io.OutputStream
out)
```

Parameters

- **out –** An output stream that you provide.

Returns

None.

Examples

- **Download default resource bundle**

```
java.io.OutputStream out = new java.io.FileOutputStream("/data/
data/com.app/files/Example.zip");
Application.getInstance().beginDownloadCustomizationBundle(out);
```

beginDownloadCustomizationBundle (String customizationBundleID java.io.OutputStream out)

Start downloading the specified resource bundle named into the output stream.

The resource bundle is saved into the output stream that you provide.

Syntax

```
public void beginDownloadCustomizationBundle (String
customizationBundleID java.io.OutputStream out)
```

Parameters

- **customizationBundleID –** The resource bundle name.
- **out –** An output stream of bytes that you provide.

Returns

None.

Examples

- **Download specified resource bundle**

```
java.io.OutputStream out = new java.io.FileOutputStream("/data/
data/com.app/files/Example.zip");
Application.getInstance().beginDownloadCustomizationBundle("Examp
le:2.0",out);
```

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. 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).

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.

Make sure the synchronization process has ended before calling this method.

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. 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

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

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

ConnectionProperties

A class that supports the configuration of properties to enable application registrations and connections.

getActivationCode

Retrieves the activation code.

Syntax

```
public String getActivationCode()
```

Parameters

None.

Returns

Returns the activation code.

setActivationCode

Sets the activation code. If you register an application manually, you must set an activation code.

Syntax

```
public void setActivationCode(String value)
```

Parameters

- **value –** The activation code.

Returns

None.

getNetworkProtocol

Retrieves the network protocol for the server connection URL, which is also known as the URL scheme.

Syntax

```
public String getNetworkProtocol()
```

Parameters

None.

Returns

Returns the network protocol for the server connection URL.

setNetworkProtocol

Sets the network protocol for the server connection URL, which is also known as the URL scheme. Defaults to HTTP.

Syntax

```
public void setNetworkProtocol(String value)
```

Parameters

- **value –** The network protocol for the server connection URL, which is also known as the URL scheme.

Returns

None.

getLoginCertificate

Retrieves the login certificate.

Syntax

```
public LoginCertificate getLoginCertificate()
```

Parameters

None.

Returns

Returns the login certificate.

setTrustedCertificates

Sets the CA certificates that are trusted by the client.

Syntax

```
public void setTrustedCertificates(X509Certificate[] values)
```

Parameters

- **values –** List of CA certificates trusted by the client.

Returns

None.

setLoginCertificate

Sets the login certificate to enable authentication by a digital certificate.

Syntax

```
public void setLoginCertificate(LoginCertificate value)
```

Parameters

- **value –** The login certificate.

Returns

None.

getLoginCredentials

Retrieves the login credentials.

Syntax

```
public LoginCredentials getLoginCredentials()
```

Parameters

None.

Returns

Returns the login credentials.

setLoginCredentials

Sets the login credentials to enable authentication by username and password.

Syntax

```
public void setLoginCredentials(LoginCredentials value)
```

Parameters

- **value –** The login credentials.

Returns

None.

getPortNumber

Retrieves the port number for the server connection URL.

Syntax

```
public int getPortNumber()
```

Parameters

None.

Returns

Returns the port number.

setPortNumber

Sets the port number for the server connection URL.

Syntax

```
public void setPortNumber(int value)
```

Parameters

- **value –** The port number for the server connection URL.

Returns

None.

getServerName

Retrieves the server name for the server connection URL.

Syntax

```
public String getServerName()
```

Parameters
None.

Returns

Returns the server name.

setServerName
Sets the server name for the server connection URL.

Syntax
```
public void setServerName(String value)
```

Parameters

- **value –** The server name for the server connection URL.

Returns

None.

getSecurityConfiguration
Retrieves the security configuration for the connection profile.

Syntax
```
public String getSecurityConfiguration()
```

Parameters
None.

Returns

Returns the security configuration.

setSecurityConfiguration
Sets the security configuration for the connection profile. If not specified, the server selects the correct security configuration by matching an application connection template with the `applicationIdentifier`. If you have two application connection templates with the same application ID but different security configurations, you must set the security configuration. Otherwise, a 'template not found' exception will be thrown.

Syntax
```
public void setSecurityConfiguration(String value)
```

Parameters

- **value –** The security configuration for the connection profile.

Returns

None.

getUrlSuffix

Retrieves the URL suffix for the server connection URL.

If the URL Suffix is left blank, then the client will attempt to discover the correct URL using default Relay Server URLs. If a valid `urlSuffix` is discovered, the value will be saved and used exclusively.

Note: If an incorrect URL is configured, it must be cleared or corrected before the client is able to connect.

Syntax

```
public String getUrlSuffix()
```

Parameters

None.

Returns

Returns the URL suffix.

setUrlSuffix

Sets the URL suffix for the server connection URL. This optional property is only used when connecting through a proxy server or Relay Server.

Note: If you provide an incorrect URL suffix, the server uses the default URL suffix when registering.

Syntax

```
public void setUrlSuffix(String value)
```

Parameters

- **value –** The URL suffix for the server connection URL.

Returns

None.

Usage

The suffix "`/%cid%/tm`" is appended if the URL does not already end in "`/tm`". If the URL ends in "`/`", then only "`%cid%/tm`" is appended.

You can optionally code a Content-ID (CID) into the URL.

For example, if the CID is "XYZ" then any of these URL suffixes:

- /ias_relay_server/client/rs_client.dll
- /ias_relay_server/client/rs_client.dll/
- /ias_relay_server/client/rs_client.dll/%cid%/tm
- /ias_relay_server/client/rs_client.dll/XYZ/tm

result in the following URL suffix:

- /ias_relay_server/client/rs_client.dll/XYX/tm

getFarmId

Retrieves the Farm ID for the server connection URL. This optional property is used in the URL discovery process when connecting through a proxy server or Relay Server. The `farmId` is substituted into the default URL templates for Relay Server on into a configured `urlSuffix`. The `farmId` is used only until a connection is successfully made and the permanent `urlSuffix` is stored.

Syntax

```
public String getFarmId()
```

Parameters

None.

Returns

Returns the Farm ID.

setFarmId

Sets the Farm ID for the server connection URL (the default is 0). This optional property is only used when connecting through a proxy server or Relay Server.

Syntax

```
public void setFarmId(String value)
```

Parameters

- **value –** The Farm ID for the server connection URL.

Returns

None.

getHttpHeaders

Retrieves any custom headers for HTTP network communications with a proxy server or Relay Server.

Syntax

```
public StringProperties getHttpHeaders()
```

Parameters

None.

Returns

Returns the HTTP headers.

setHttpHeaders

Sets the HTTP headers for network communications through a proxy server or Relay Server.

Syntax

```
public void setHttpHeaders(StringProperties oHeaders)
```

Parameters

- **oHeaders –** Optional headers for HTTP network communication with a proxy server or Relay Server.

Returns

None.

getHttpCookies

Retrieves any custom HTTP cookies for network communications with a proxy server or Relay Server.

Syntax

```
public StringProperties getHttpCookies()
```

Parameters

None.

Returns

Returns the HTTP cookies.

setHttpCookies

Sets the HTTP cookies for network communications through a proxy server or Relay Server.

Syntax

```
public void setHttpCookies(StringProperties oCookies)
```

Parameters

- **oCookies –** Optional cookies for HTTP network communication with a proxy server or Relay Server.

Returns

None.

getHttpCredentials

Retrieves the credentials for HTTP basic authentication with a proxy server or Relay Server.

Syntax

```
public LoginCredentials getHttpCredentials()
```

Parameters

None.

Returns

Returns credentials for HTTP basic authentication with a proxy server or Relay Server.

setHttpCredentials

Sets the HTTP credentials for basic authentication through a proxy server or Relay Server.

Syntax

```
public void setHttpCredentials(LoginCredentials httpCredentials)
```

Parameters

- **httpCredentials –** credentials for HTTP basic authentication with proxy/relay server.

Returns

None.

ApplicationSettings

Methods or properties in the `ApplicationSettings` class.

isApplicationSettingsAvailable

Checks whether the application settings are available from the SAP Mobile Server.

Syntax

```
public boolean isApplicationSettingsAvailable()
```

Parameters

None.

Returns

Returns true if the application settings are available.

Examples

- **Check if application settings are available**

```
boolean isSettingsAvailable =
Application.getInstance().getApplicationSettings().isApplicationS
ettingsAvailable();
```

getStringProperty

Retrieves a string property from the `applicationSettings`.

Syntax

```
public String getStringProperty(ConnectionPropertyType type)
```

Parameters

- **type –** Type of ConnectionPropertyType.

Returns

Returns a string property value.

Examples

- **Get string property**

```
String user_name =
appSettings.getStringProperty(ConnectionPropertyType.UserName);
```

getIntegerProperty

Retrieves an integer property from the `applicationSettings`.

Syntax

```
public Integer getIntegerProperty(ConnectionPropertyType type)
```

Parameters

- **type –** Type of ConnectionPropertyType.

Returns

Returns an integer property value.

Examples

- **Get integer property**

```
int min_length =
appSettings.getIntegerProperty(ConnectionPropertyType.PwdPolicy_L
ength);
```

getBooleanProperty

Retrieves a boolean property from the `applicationSettings`.

Syntax

```
public Boolean getBooleanProperty(ConnectionPropertyType type)
```

Parameters

- **type –** Type of ConnectionPropertyType.

Returns

Returns a boolean property value.

Examples

- **Get boolean property**

```
boolean pwdpolicy_enabled =
appSettings.getBooleanProperty(ConnectionPropertyType.PwdPolicy_E
nabled);
```

getCustom1
A custom application setting for use by the application code.

Syntax
```
public String getCustom1()
```

Parameters
None.

Returns

Returns a custom application setting.

getCustom2
A custom application setting for use by the application code.

Syntax
```
public String getCustom2()
```

Parameters
None.

Returns

Returns a custom application setting.

getCustom3
A custom application setting for use by the application code.

Syntax
```
public String getCustom3()
```

Parameters
None.

Returns

Returns a custom application setting.

getCustom4

A custom application setting for use by the application code.

Syntax

```
public String getCustom4()
```

Parameters

None.

Returns

Returns a custom application setting.

getDomainName

Syntax

```
public String getDomainName()
```

Parameters

None.

Returns

Returns the domain name.

getConnectionId

Syntax

```
public String getConnectionId()
```

Parameters

None.

Returns

Returns a Connection ID for this application setting.

ConnectionPropertyType

Methods or properties in the `ConnectionPropertyType` class.

See the generated API reference provided with the Mobile SDK for a complete list of methods in the `ConnectionPropertyType` class.

PwdPolicy_Enabled
Indicates whether the password policy is enabled.

Syntax
```
ConnectionPropertyType PwdPolicy_Enabled
```

Parameters
None.

Returns

Examples

- **PwdPolicy_Enabled**

```
boolean pwdpolicy_enabled =
appSettings.getBooleanProperty(ConnectionPropertyType.PwdPolicy_E
nabled);
```

PwdPolicy_Default_Password_Allowed
Indicates whether the client application is allowed to use the default password for the data vault.

Syntax
```
ConnectionPropertyType PwdPolicy_Default_Password_Allowed
```

Parameters
None.

Returns

None.

Examples

- **PwdPolicy_Default_Password_Allowed**

```
boolean default_password_allowed =
appSettings.getBooleanProperty(ConnectionPropertyType.PwdPolicy_D
efault_Password_Allowed);
```

PwdPolicy_Length

Defines the minimum length for a password.

Syntax

```
ConnectionPropertyType PwdPolicy_Length
```

Parameters

None.

Returns

Returns an integer value for the minimum length for a password.

Examples

- **PwdPolicy_Length**

```
int min_length =
appSettings.getIntegerProperty(ConnectionPropertyType.PwdPolicy_L
ength);
```

PwdPolicy_Has_Digits

Indicates if the password must contain digits.

Syntax

```
ConnectionPropertyType PwdPolicy_Has_Digits
```

Parameters

None.

Returns

Returns true if the password must contain digits.

Examples

- **PwdPolicy_Has_Digits**

```
boolean has_digits =
appSettings.getBooleanProperty(ConnectionPropertyType.PwdPolicy_H
as_Digits);
```

PwdPolicy_Has_Upper
Indicates if the password must contain at least one upper case character.

Syntax
```
ConnectionPropertyType PwdPolicy_Has_Upper
```

Parameters
None.

Returns
Returns true if the password must contain at least one upper case character.

Examples
- **PwdPolicy_Has_Upper**

```
boolean has_upper =
appSettings.getBooleanProperty(ConnectionPropertyType.PwdPolicy_H
as_Upper);
```

PwdPolicy_Has_Lower
Indicates if the password must contain at least one lower case character.

Syntax
```
ConnectionPropertyType PwdPolicy_Has_Lower
```

Parameters
None.

Returns
Returns true if the password contains at least one lower case character.

Examples
- **PwdPolicy_Has_Lower**

```
boolean has_lower =
appSettings.getBooleanProperty(ConnectionPropertyType.PwdPolicy_H
as_Lower);
```

PwdPolicy_Has_Special

Indicates if the password must contain at least one special character. A special character is a character in the set "~!@#$%^&*()-+".

Syntax

```
ConnectionPropertyType PwdPolicy_Has_Special
```

Parameters

None.

Returns

Returns true if the password must contain at least one special character.

Examples

- **PwdPolicy_Has_Special**

```
boolean has_special =
appSettings.getBooleanProperty(ConnectionPropertyType.PwdPolicy_H
as_Special);
```

PwdPolicy_Expires_In_N_Days

Specifies the number of days in which the password expires from the date of setting the password.

Syntax

```
ConnectionPropertyType PwdPolicy_Expires_In_N_Days
```

Parameters

None.

Returns

Returns an integer value for the number of days in which the password expires.

Examples

- **PwdPolicy_Expires_In_N_Days**

```
int expires_in_n_days =
appSettings.getIntegerProperty(ConnectionPropertyType.PwdPolicy_E
xpires_In_N_Days);
```

PwdPolicy_Min_Unique_Chars

Specifies the minimum number of unique characters in the password.

Syntax

```
ConnectionPropertyType PwdPolicy_Min_Unique_Chars
```

Parameters

None.

Returns

An integer specifying the minimum number of unique characters in the password.

Examples

- **PwdPolicy_Min_Unique_Characters**

```
int min_unique_characters =
appSettings.getIntegerProperty(ConnectionPropertyType.PwdPolicy_M
in_Unique_Chars);
```

PwdPolicy_Lock_Timeout

Specifies the timeout value (in seconds) after which the vault is locked from the unlock time. A value of 0 indicates no timeout.

Syntax

```
ConnectionPropertyType PwdPolicy_Lock_Timeout
```

Parameters

None.

Returns

An integer specifying the timeout value.

Examples

- **PwdPolicy_Lock_Timeout**

```
int lock_timeout =
appSettings.getIntegerProperty(ConnectionPropertyType.PwdPolicy_L
ock_Timeout);
```

PwdPolicy_Retry_Limit

Specifies the number of failed unlock attempts after which the data vault is deleted. A value of 0 indicates no retry limit.

Syntax

```
ConnectionPropertyType PwdPolicy_Retry_Limit
```

Parameters

None.

Returns

An integer specifying the number of failed unlock attempts after which the data vault is deleted.

Examples

- **PwdPolicy_Retry_Limit**

```
int retry_limit =
appSettings.getIntegerProperty(ConnectionPropertyType.PwdPolicy_R
etry_Limit);
```

Afaria APIs

Use the Afaria APIs to provision your SAP Mobile Platform application with configuration data for connecting to the SAP Mobile Server, and certificates.

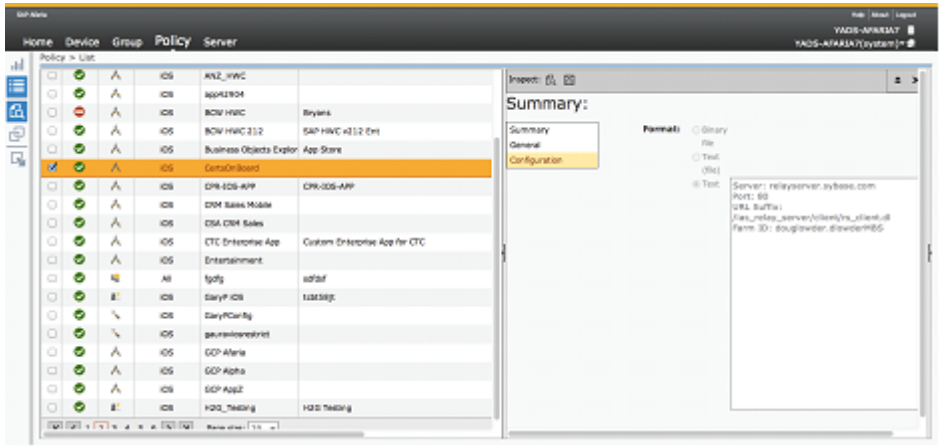
Using Afaria to Provision Configuration Data

You can use Afaria to provision configuration data for a SAP Mobile Platform application, including the SAP Mobile Server server name, port number, and other parameters.

To use these APIs you must provide the application to the device through an Afaria application policy. When setting up such an application policy, the Afaria administration interface provides an option to add configuration data to the policy as text or binary.

The following is an an example of the Afaria administration screen for an application policy that provides an application named "CertsOnBoard" to an enrolled device. The "Configuration" tab shows the configuration data provided to the application.

In this case, the configuration information is added using the administration user interface, but it can also be provided as a text or binary file. The example shows plain text, but you can also provide the information as XML or JSON text for easier parsing by the application.



You can obtain configuration data for your application using Afaria by calling the following API from the `com.sybase.afaria.SeedDataAPI` class (in `AfariaSSL.jar`).

```
String
com.sybase.afaria.SeedDataAPI.retrieveSeedData(SeedDataCredentials
arg0) throws SeedDataAPIException
```

To access this data, the application provides `SeedDataCredentials` to the `retrieveSeedData` API. If the device is correctly enrolled to Afaria, the API returns a string which contains the full path to a file in the application's sandbox with the seed data.

```
SeedDataCredentials sdc = new
SeedDataCredentials("supadmin","xnetqa","abc");
String result = SeedDataAPI.retrieveSeedData(sdc);
resultText.append("the seed data file: " + result);
BufferedReader reader = null;
Map<String, String> keyValues = null;
try
{
  reader = new BufferedReader(new FileReader(result));
  String line = null;
  keyValues = new java.util.HashMap<String, String>();
  while ((line = reader.readLine()) != null)
  {
    resultText.append(line + "\r\n");
    String[] strs = line.split(":");
    if(strs.length == 2)
    {
      keyValues.put(strs[0], strs[1]);
    }
  }

}

catch(Exception ex)
{
  throw new RuntimeException(ex);
```

```
}
finally
{
  if(reader != null)
  {
    reader.close();
  }
}

//set the download configuration to application connectionProperties
Application app = Application.getInstance();
ConnectionProperties appConnections = app.getConnectionProperties();
appConnections.setServerName(keyValues.get("server"));
appConnections.setPortNumber(Integer.parseInt(keyValues.get("port")
));
appConnections.setUrlSuffix(keyValues.get("URL Suffix"));
appConnections.setFarmId(keyValues.get("Farm ID"));

resultText.append("server name is set to: " +
appConnections.getServerName() + "\r\n");
resultText.append("server port is set to: " +
appConnections.getPortNumber() + "\r\n");
resultText.append("url suffix is set to: " +
appConnections.getUrlSuffix() + "\r\n");
resultText.append("farm id is set to: " + appConnections.getFarmId()
+ "\r\n");
```

The `Textview` output is:

```
the seed data file: data/data/com.app/files/seedData/
SUPOnboardingSeedData.txt
server: relayserver.sybase.com
port: 80
URL Suffix: /ias_relay_server/client/rs_client.dl
Farm ID: example.exampleMBS
server name is set to: relayserver.sybase.com
server port is set to: 80
url suffix is set to: /ias_relay_server/client/rs_client.dl
farm id is set to: example.exampleMBS
```

For more information on the Afaria APIs and the meanings of return codes, see the Afaria documentation.

Using Certificates from Afaria for Authentication

One of the features of Afaria is the ability to provide a device with a signed certificate that could be used as an authentication credential for SAP Mobile Platform. This note explains how to take a certificate provided by Afaria and convert it into a form suitable for use with SAP Mobile Platform.

Prerequisites:

- The application has been built using the SAP Mobile Platform generated code and framework headers and libraries.

- The application has been registered with the Afaria server as an application policy and made available to the client device.

In SAP Mobile Platform, a certificate can be used for authentication by creating a LoginCertificate object (the LoginCertificate class), and setting that as the certificate property in the client's synchronization profile.

After calling the Afaria APIs to get initial settings and configuration data, an application using Afaria may obtain a signed certificate using this API:

```
X509Certificate
com.sybase.afaria.SeedDataAPI.retrieveCertificate(RSAPublicKey
arg0, RSAPrivateKey arg1,  String arg2, String arg3,
SeedDataCredentials arg4) throws SeedDataAPIException
```

After this, the application will have an X509Certificate object. The certificate data in the X509Certificate object cannot be used as a LoginCertificate. It must be converted into a LoginCertificate.

This sample code shows how to get the Afaria certificate, create a `LoginCertificate` object, and attach it to a SAP Mobile Platform synchronization profile.

The part of the code from the top through the section which retrieves the `LoginCertificate` object is performed only once during application initialization where you are obtaining the certificate through Afaria. The `LoginCertificate` is next stored in the data vault. Each time the application runs thereafter, it retrieves the `LoginCertificate` from the data vault and sets it into the `connProperties.setLoginCertificate(lc);` as shown, before synchronizing.

```
String commonName = "SMP-SSO";
String passWord = "smp";
String pkcsFile = "/mnt/sdcard/SMP-SSO.pfx";
//first, initialize SeedDataAPI using current Android Activity
context
SeedDataAPI.initialize(this);

//generate a key pair using java.security API
KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("RSA");
keyPairGen.initialize(1024);
KeyPair keyPair = keyPairGen.generateKeyPair();
RSAPublicKey publicKey = (RSAPublicKey) keyPair.getPublic();
RSAPrivateKey privateKey = (RSAPrivateKey) keyPair.getPrivate();

//get the X509Certificate object from Afaria server by Afaria API
X509Certificate cer = SeedDataAPI.retrieveCertificate(publicKey,
privateKey, commonName, passWord, null);

//we need to wrap the X509Certificate and private key to a PKCS12
Certificate
java.security.KeyStore ks =
java.security.KeyStore.getInstance("PKCS12");
ks.load(null, passWord.toCharArray());
ks.setCertificateEntry(commonName, cer);
```

```
Certificate[] chain = {cer};
ks.setKeyEntry(commonName, privateKey, passWord.toCharArray(),
chain);
FileOutputStream out = new FileOutputStream(pkcsFile);
ks.store(out, passWord.toCharArray());

//call API to get LoginCertificate object from the PKCS12 certificate
file
LoginCertificate lc =
CertificateStore.getDefault().getSignedCertificateFromFile(pkcsFile
, passWord);

//use the loginCertificate to register Application
Application app = Application.getInstance();
ConnectionProperties connProperties = app.getConnectionProperties();
connProperties.setLoginCertificate(lc);
```

Connection APIs

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

See also

- *Initially Starting an Application* on page 29

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 = SMP101DB.getConnectionProfile();
profile.setPageSize( 4*1024 );
profile.setEncryptionKey("Your key of more than 16 characters");
```

Note: If you set the page size to a negative value, the framework uses a default value of 4K as the page size.

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.

You can use the `cacheSize` API to control the size of the memory cache used by the database.

```
public void setCacheSize(int cacheSize)
```

See also

- *Setting Up the Connection Profile* on page 34

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 SAP 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 =
SMP101DB.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 = SMP101DB.getConnectionProfile();
cp.setProperty("databaseFile", "SMP101.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/smp101.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: SAP 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 SAP Mobile 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 SAP Mobile 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 SAP Mobile Server to connect to the SAP Mobile 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 = SMP101DB.getSynchronizationProfile();
profile.setDomainName( "default" );
profile.setServerName( "smp.example.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 SAP Mobile Server by including "compression=zlib" into the sync parameters:

```
SMP101DB.getSynchronizationProfile().getStreamParams().setZlibCompr
ession(true);
```

See also

- *Setting Up the Synchronization Profile* on page 35

Connect the Data Synchronization Channel Through a Relay Server

To enable your client application to connect through a Relay Server, you can enter the related configuration in the application connection template through SAP Control Center, and/or setup the configuration properties in the synchronization profile using the object API.

Edit SMP101DB by modifying the values of the Relay Server properties for your Relay Server environment.

To update properties for a Relay Server installed on Apache:

```
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 a 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 *SAP Control Center for SAP Mobile Platform*.

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.

```
SMP101DB.getSynchronizationProfile().setAsyncReplay(false);
```

Note: Synchronous operation replay against MBOs using an EIS managed cache group policy are automatically treated as asynchronous replay by the SAP Mobile Platform Runtime.

Authentication APIs

You can log in to the SAP Mobile 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 SAP Mobile Server connection profile and server domain selected during code generation. You can log in to the SAP Mobile Server with your user name and credentials.

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

- **`onlineLogin()`** – authenticates credentials against the SAP Mobile Server.

Sample Code: Setting Up Login Credentials

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

```
/// SMP101DB is a generated database class
///First put the p12 certificate file on the SDCard of the device,
for example "/mnt/sdcard/sap101.p12"

//Get certificate from certificate file
CertificateStore myStore =
certificateStore.getDefault();
String certFile = "/mnt/sdcard/sap101.p12";
String password = "my p12 password";
LoginCertificate lc = myStore.getSignedCertificateFromFile(certFile,
password);

// Save the login certificate to your synchronization profile
SMP101DB.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);

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

Sample Code: Mutual Authentication

Illustrates client configuration to support mutual authentication, as well as other APIs related to certificate handling:

```
//Step 1: Set the CA certificates trusted by the client for mutual
authentication

CertificateFactory certificate_factory;
X509Certificate serverCA = null;
try {
    certificate_factory = CertificateFactory .getInstance("X.509");
    FileInputStream file_inputstream = new FileInputStream("/data/
data/" +
    this.getPackageName() + "/server.cer");
    serverCA = (X509Certificate)
certificate_factory .generateCertificate(file_inputstream);
    }

   catch (Exception e) {
      // TODO Auto-generated catch block
       e.printStackTrace();
       }

........

ConnectionProperties pro = app.getConnectionProperties();
pro.setTrustedCertificates(new X509Certificate[] {serverCA});

//Step 2: Get the login certificate from a certificate store

CertificateStore myStore = CertificateStore.getDefault();
LoginCertificate lc = myStore.getSignedCertificateFromFile("/data/
data/com.test/client1-old.pfx", "changeit");
Application app = Application.getInstance();

app.setApplicationContext(this);
app.setApplicationIdentifier("customer.service");
ConnectionProperties pro = app.getConnectionProperties();
pro.setLoginCertificate(lc);

//Step 3:  Register the application

pro.setServerName("10.0.0.2");
pro.setNetworkProtocol("HTTPS");
pro.setFarmId("0");
pro.setUrlSuffix("");
pro.setSecurityConfiguration("cert");
```

```
if (Application.getInstance().getRegistrationStatus() ==
RegistrationStatus.UNREGISTERED)
{
    Application.getInstance().registerApplication(100);
}

......

//Step 4: Get ready to synchronize

DsTestDB.getSynchronizationProfile().setNetworkProtocol("HTTPS");
DsTestDB.getSynchronizationProfile().setPortNumber(2482);
DsTestDB.getSynchronizationProfile().setServerName("10.0.0.2");
DsTestDB.getSynchronizationProfile().setCertificate(lc);
DsTestDB.getSynchronizationProfile().setNetworkStreamParams("truste
d_certificate=serverCA.crt");

//Step 5: Synchronize
.......
DsTestDB.synchronize();
```

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 SAP Mobile 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 SAP Mobile Server Connections

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

```
LoginCertificate cert = LoginCertificate.load("myCert", vault);
ConnectionProfile syncProfile =
SMP101DB.getSynchronizationProfile();
syncProfile.setCertificate(cert);
```

Connecting to SAP Mobile 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.

```
SMP101DB.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 45

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 SAP Mobile 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 the client and the SAP Mobile 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 the SAP Mobile Server. Two built-in (session) personalization keys — username and password — can be used to perform single sign-on from the device application to the SAP Mobile Server, authentication and authorization on the SAP Mobile 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 SAP Mobile 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 =
SMP101DB.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 43
- *Specifying Synchronization Parameters* on page 45

Managing Synchronization Parameters

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

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

To add a synchronization parameter:

```
CustomerSubscription sp = new CustomerSubscription();
sp.setName("example");
Customer.addSubscription(sp);
```

To list all synchronization parameters:

```
com.sybase.collections.GenericList<CustomerSubscription> r =
Customer.getSubscriptions();
```

To remove a synchronization parameter:

```
com.sybase.collections.GenericList<CustomerSubscription> r =
Customer.getSubscriptions();
CustomerSubscription sub = r.item(0);
Customer.removeSubscription(sub);
```

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:

```
SMP101DB.synchronize (string synchronizationGroup)
```

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

```
SMP101DB.synchronize("my-sync-group");
```

Or, you can synchronize all synchronization groups:

```
SMP101DB.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:
`SMP101DB.beginSynchronize("default");` or
`SMP101DB.synchronize("default");`

If there is no other synchronization group, call `SMP101DB.beginSynchronize();` or `SMP101DB.synchronize();`

To synchronize a synchronization group asynchronously:

```
GenericList<SynchronizationGroup> syncGroups = new
GenericList<SynchronizationGroup>();
syncGroups.add(SMP101DB.getSynchronizationGroup("my-sync-group"));
SMP101DB.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 (!SMP101DB.isReplayQueueEmpty())
{
   // There are OperationReplay not uploaded to server
   GenericList<SynchronizationGroup> sgs = new
GenericList<SynchronizationGroup>();
   sgs.add(SMP101DB.getSynchronizationGroup("system"));
   SMP101DB.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 SAP Mobile Server, to suspend or resume requests to the SAP Mobile Server, and to recover data related to the package from the server.

Note: The `beginOnlineLogin`, `suspendSubbscription`, `resumeSubscription`, and `vacuumDatabase` methods are for use with DOE-based applications only.

beginOnlineLogin

Sends a login message to the SAP Mobile 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 SAP Mobile Server with your username and credentials.

beginOnlineLogin sends a message to the SAP Mobile Server with the username and password. The SAP Mobile 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.

  ```
  SMP101DB.beginOnlineLogin("supAdminID", "supPwd");
  ```

subscribe

Subscribes to a server package. A subscription message is sent to the SAP Mobile Server and the application receives a subscription request result notification from the the SAP Mobile 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 SAP Mobile Platform client process.
- The device application has already configured SAP Mobile 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 SMP101DB.

  ```
  SMP101DB.subscribe();
  ```

unsubscribe

Removes an existing subscription to a server package. An unsubscription message is sent to the SAP Mobile Server and the application receives a subscription request result notification from the SAP Mobile 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 SMP101DB.

```
SMP101DB.unsubscribe();
```

suspendSubscription

Sends a suspend request to the SAP Mobile Server to notify the server to stop delivering data changes. A suspend subscription message is sent to the SAP Mobile Server and the application receives a suspend subscription request result notification from the SAP Mobile 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 suspendSubscription()
```

Parameters

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

Returns

None.

Examples

- **Suspend a Subscription –** Suspend the subscription to SMP101DB.

```
SMP101DB.suspendSubscription();
```

beginSynchronize

Sends a message to the SAP Mobile 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

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 SMP101DB for all synchronization groups.

```
SMP101DB.beginSynchronize(null, "my context");
```

resumeSubscription

Sends a resume request to the SAP Mobile Server.

The resume request notifies the SAP Mobile 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 SMP101DB.

```
SMP101DB.resumeSubscription();
```

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.

SAP Mobile 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 =
SMP101DB.getSynchronizationGroup("TCNEnabled");

if (!sg.getEnableSIS())
{
  sg.setEnableSIS(true);
  sg.setInterval(2); // 2 minutes
  sg.save();
  SMP101DB.synchronize("TCNEnabled");
}
```

If you see that setInterval is set to 0, then change detection is disabled, and notifications will not be delivered. Enable change detection and notification delivery by setting an appropriate value. For recommendations, see *Configuring Synchronization Groups* in *SAP Control Center for SAP Mobile Platform*.

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 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)
```

Resumable Synchronization

The resumable synchronization APIs enable an interrupted synchronization with partial data downloaded to the device to resume from the point of interruption, without restarting the entire download again.

Resumable synchronization is particularly useful when you synchronize large data sets, and do not want to have to cancel or roll back any partially downloaded data. You can resume the synchronization that meets all of these conditions:

- It must be within the lifetime of the application. You cannot resume the session after the application terminates, for example if the application is closed.
- The resumable option must be set before calling synchronization.
- The synchronization must be performed asynchronously in background mode.
- There is partial data downloaded to the device when the interruption occurs.

Synchronization can be interrruped by either environmental changes (for example, a dropped connection) or when the application code disables or cancels the synchronization (for example, by returning true in the SyncStatusListener objectSyncStatus method).

- **Application code disables or cancels the synchronization for any reason –** for example,

Note: You cannot resume synchronization if the MobiLink Server restarts, since all pending downloads are lost.

If your callback handler is registered, and if some data has downloaded data before the interruption, this status is returned in the CallbackHandler onSynchronize method at the time of the interruption:

```
SynchronizationStatus.INTERRUPTED_WITH_PARTIAL_DATA
```

The system locks the synchronization queue in a pending synchronization mode that waits for the response to either resume or cancel.

The application determines when to resume the synchronization. The resume Synchronization API first releases the locked synchronization queue, then sends a resume

synchronization request to the server. If the request is too early, the resumption may fail immediately if the connection is not yet available. However, if the request is sent too late, the download data may no longer be available on the server.

See the topic *Configuring Resumable Synchronization* in the *SAP Control Center for SAP Mobile Server* collection for information about setting a cache storage size for resumable download data. The default cache size is 5 GB.

Resumable Synchronization API

Resume an interrupted synchronization where partial data has been downloaded to the device.

Syntax

To resume a pending synchronization, call:

```
SMP101DB.resumePendingSynchronization();
```

The application can also determine when to cancel synchronization. For example, you could add logic to the application that sets a max retry resume to cancel the pending synchronization when the application reaches the max setting. The cancel pending synchronization API also unlocks the locked synchronization queue first, but instead of sending a synchronization request, it rolls back any partial data that has already downloaded to the device. To not resume the pending synchronization, call:

```
SMP101DB.cancelPendingSynchronization();
```

1. When the system is in a pending synchronization state, waiting for either a resume pending synchronization request by calling resumePendingSynchronization API or a cancellation request by calling cancelPendingSynchronization, the synchronization queue is locked. Any other new blocking or non-blocking synchronization requests are not permitted.
2. If partial data has not downloaded to the device when the interruption occurs, for example, the device is uploading data to the server, or the server is still fetching data from the EIS, the synchronization request terminates immediately and the client reverts to standard behavior. If the interruption is due to:
   - **Environmental changes –** instead of receiving a SynchronizationStatus.INTERRUPTED_WITH_PARTIAL_DATA status, the exception in the sync context is returned.
   - **Application code disables or cancels the synchronization for any reason –** SyncStatusListener's objectSyncStatus method returns true.

Examples

- **Sample code for setting the resumable option then synchronizing**

```
...
    ...
    ...
    //turn on the resume synchronization option

SMP101DB.getSynchronizationProfile().setResumeSynchronization(tru
```

```
e);
    MyCallbackHandlerForResumeSync callback = new
MyCallbackHandlerForResumeSync();
    SMP101DBRegisterCallbackHandler(callback);
    MyResumeSyncStatusListener listener = new
MyResumeSyncStatusListener();
    GenericList<SynchronizationGroup> groups = new
GenericList<SynchronizationGroup>();
    groups.add(SMP101DB.getSynchronizationGroup("default"));

    //submit a non-blocking sync request
    SMP101DB.beginSynchronize(groups, null, false, listener);
```

- **CallbackHandler sample code in receiving SynchronizationStatus.INTERRUPTED_WITH_PARTIAL_DATA status**

```
//in the implementation of SUPDefaultCallbackHandler class
    ...
    ...
    ...
    int onSynchronize(GenericList<SynchronizationGroup> groups,
SynchronizationContext context)
    {
        switch (context.getStatus())
        {
        ...
        case SynchronizationStatus.INTERRUPTED_WITH_PARTIAL_DATA:
            _isInterruptedWithPartialData = true;

          // By returning continue, the system waits for a request
to resume or cancel the pending synchronization and the
synchronization queue is locked.
           // If returning SynchronizationAction.CANCEL instead,
the system automatically cancels the pending synchronization.
          // Application cannot resume the synchronization and the
synchronization queue is not locked.

            return SynchronizationAction.CONTINUE;
        ...
        }
    }
```

- **Sample code for resuming or canceling synchronization on sync callback status changed notification**

```
void handleInterruptedWithPartialData()
    {
        if (MAX_INTERRUPTED_STATUS == _interruptedStatusCount)
        {
            cancelPendingSync();
            System.out.println("Resume sync is canceled.");
            System.out.println("Interrupted status count reached
the max: " + MAX_INTERRUPTED_STATUS + ".");
        }
        else
        {
            _isInterruptedWithPartialData = true;
```

```
            _interruptedStatusCount++;

            int waitCount = 0;
            while(!_isDeviceConnectedToInternet)
            {
                Thread.sleep(1000);
                waitCount++;

                if (waitCount > MAX_WAIT)
                {
                    System.out.println("There is no network
connection " + MAX_WAIT + " seconds.");
                    break;
                }
            }

            if (_isDeviceConnectedToInternet)
            {
                resumeSync();
            }
            else
            {
                cancelPendingSync();

                System.out.println("Resume sync is canceled.");
              System.out.println("there's no network connectivity
for too long.");
            }
        }
    }

    void cancelPendingSync()
    {
        System.out.println("going to cancel pending sync...");

        if (SMP101DB.cancelPendingSynchronization())
        {
            System.out.println("failed to cancel pending sync.");
        }
        else
        {
           System.out.println("pending sync has been cancelled.");
        }
    }

    void resumeSync()
    {
        if (!_isInterruptedWithPartialData)
        {
          System.out.println("no resume as there is no interrupted
with partiald data.");
            return;
        }

        System.out.println("going to resume sync...");
        bool resumed = SMP101DB.resumePendingSynchronization();
```

```
        _isInterruptedWithPartialData = false;
        if (!resumed)
        {
            System.out.println("failed to resume pending syn.");
        }
        else
        {
           _resumeSyncCount++;
           System.out.println("resume sync count: " +
_resumeSyncCount + ".");
        }
    }
```

Resumable Synchronization Recommendations

Consider these recommendations when implementing resumable synchronization.

- Set a max resume-attempt in your code so the application does not continuously try to resume and fail for the same reason. If the max count is reached, the sync is canceled.
- Check for connectivity before attempting to resume the synchronization. The code can be implemented to wait a short time to recheck connectivity, along with a timeout period to wait a sufficient amount of time.
- Add code that obtains a flag indicating that the sync is disabled when interruption occurs because the application code disables or cancels the synchronization for any reason. For example, the application is running in the background, and you want to resume only when the application is running in the foreground.

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.

Log records are automatically created when an operation replay fails in the SAP Mobile Server. If an operation replay succeeds, there is no `LogRecord` created by default (note that an SAP default result checker may write a log record even when the SAP operation succeeds). To get the confirmation when an operation replay succeeds, register a CallbackHandler and implement the `CallbackHandler.onReplaySuccess` method.

See *Developer Guide: Android Object API Applications* > *Client Object API Usage* > *Callback and Listener APIs*.

LogRecord API

`LogRecord` stores two types of logs.

- Operation logs on the SAP Mobile Server. These logs can be downloaded to the device.
- Client logs. These logs can be uploaded to the SAP Mobile 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();
SMP101DB.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 SAP Mobile Server only creates a logRecord when an operation fails or completes with warnings.

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 = SMP101DB.getLogRecords(query);
  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();
    }
   SMP101DB.beginSynchronize(null, null);
}
```

A `LogRecord` is not generated for a successful operation replay. SAP Mobile Server only creates one when an operation fails or completes with warnings. The client is responsible for removing operation replay log records. SAP Mobile Server typically allows a period of time for the client to download and act on the operation replay log record. Therefore, the client should proactively remove these log records when they are consumed. Failure to do so may result in accumulation of operation replay log records until SAP Mobile Server removes them. This sample code illustrates how to find the corresponding MBO with the `LogRecord` and delete the log record when it is processed.

```
private void processLogs()
{
   Query query = new Query();
   GenericList<LogRecord> logRecords =
SMP101DB.getLogRecords(query);
   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();
      }
   }
```

SAP Mobile Server is responsible for deleting client log records uploaded by the application. These application logs are used for audit and/or support services. Determine and set the retention policy from SAP Control Center after consulting with the application's developers. If there are multiple applications using the same package, retain them based on the maximum required time for each application. Client log records are removed that are outside the retention window, and deleted records removed from the client database the next time the application synchronizes. See *Improve Synchronization Performance by Reducing the Log Record Size* in *Troubleshooting* for details about reducing the Log Record size.

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. The `Logger` is an abstraction over the LogRecord API to write records of various log levels into the LogRecord MBO on the client database.

```
Logger logger = SMP101DB.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");

// Prepare all outstanding client generated log records for upload
SMP101DB.submitLogRecords();
```

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 Operation 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 Root 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**

  ```
  SMP101DB.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 =
  SMP101DB.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**

```
SMP101DB.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 simultaneously.

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**

```
SMP101DB.disableChangeLog();
```

Code Samples

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

```
SMP101DB.enableChangeLog();
SMP101DB.synchronize();

// Retrieve all change logs
GenericList<ChangeLog> logs = SMP101DB.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(SMP101.EntityType.Customer),
               AttributeTest.EQUAL);
query.setTestCriteria(at);
logs = SMP101DB.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 = SMP101DB.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.

```
SMP101DB.generateEncryptionKey();
// store the encryption key at somewhere for reuse later
ConnectionProfile profile = SMP101DB.getConnectionProfile();
String key = profile.getEncryptionKey();
...
SMP101DB.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 enable E2EE for the SMP101 package:

```
ConnectionProfile cp=SMP101DB.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();
SMP101DB.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-128. The `DataVault` class allows you create a named vault, and specify a password and salt used to unlock it. The password can be of arbitrary length and can include any characters. The password and salt together 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 occur. If the user enters an incorrect password a configurable number of times, the vault is deleted and any data stored within it becomes unrecoverable. The vault can also relock itself after a configurable amount of time.

Typical usage of the `DataVault` is to implement an application login screen. Upon application start, the user is prompted for a password, which unlocks the vault. If the unlock attempt is successful, the user is allowed into the rest of the application. User credentials for synchronization can also be extracted from the vault so the user need not reenter 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** *SMP_HOME*`\MobileSDK23\ObjectAPI\Android\SybaseDataProvider.apk`

init

Initialization function that you must call with the application's context before you call any of the other vault methods. In addition to saving the context for later use, this method also initializes static member variables (such as encryption objects).

Syntax

```
public static void init(android.content.Context oContext)
```

Parameters

- **oContext –**
  `Valid application context.`

Returns

None.

Examples

- **Initialize**

```
DataVault.init(oContext);
```

createVault

Creates a new secure store (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 with the same name already exists, this method throws an exception. A newly created vault is in the unlocked state.

Syntax

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

Parameters

- **name –** an arbitrary name for a `DataVault` instance on this device. This name is effectively the primary key for looking up `DataVault` instances on the device, so it cannot use the same name as any existing instance. If it does, this method throws an exception with error code INVALID_ARG. The name also cannot be empty or null.

- **password –** the initial encryption password for this DataVault. This is the password needed for unlocking the vault. If null is passed, a default password is computed and used.
- **salt –** the encryption salt value for this DataVault. This value, combined with the password, creates the actual encryption key that protects the data in the vault. If null is passed, a default salt is computed and used.

Returns

Returns the newly created instance of the DataVault with the provided ID. The returned DataVault is in the unlocked state with default configuration values. To change the default configuration values, you can immediately call the "set" methods for the values you want to change.

If a vault already exists with the same name, a `DataVaultException` is thrown 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, returns true if it does and false if the datavault is locked, does not exist, or is inaccessible for any other reason.

Syntax

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

Parameters

- **name –** the vault name.

Returns

Returns true if the vault exists; otherwise returns false.

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");
}
```

vaultExists2

Tests whether the specified vault exists, returns true if the vault exists; otherwise returns false. If an error occurs while reading the keychain, throws an kDataVaultExceptionReasonIORead exception.

Syntax

```
+ (BOOL)vaultExists2:(NSString*)dataVaultID;
```

Parameters

- **dataVaultID –** the vault name.

Returns

Returns true if the vault exists; otherwise returns false. If an error occurs while reading the keychain, throws an kDataVaultExceptionReasonIORead exception.

Examples

- **Check if a data vault exists –** checks if a data vault called `myVault` exists, and if so, deletes it.

```
@try {
   if ([SUPDataVault vaultExists2:@"myVault"]) {
      [SUPDataVault deleteVault:@"myVault"];
   }
}
@catch ( SUPDataVaultException *exception ) {
   //handle the exception
}
```

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.

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");
}
```

getDataNames

Retrieves information about the data names stored in the vault.

The application can pass the data names to `getValue` or `getString` to retrieve the data values.

Syntax

```
public abstract DataVault.DVDataName[] getDataNames()
```

Parameters

None.

Returns

Returns a `DVPasswordPolicy` object, as an array of `DVDataName` structure objects.

Examples

- **Get data names**

```
// Call getDataNames to retrieve all stored element names from our
data vault.
DataVault.DVDataName[] dataNameArray = oDataVault.getDataNames();
for ( int i = 0; i < dataNameArray.length; i++ )
{
  if ( dataNameArray[i].iType == DataVault.DV_DATA_TYPE_STRING )
  {
    String thisStringValue =
oDataVault.getString( dataNameArray[i].sName );
  }
  else
  {
    byte[] thisBinaryValue =
oDataVault.getValue( dataNameArray[i].sName );
  }
}
```

setPasswordPolicy

Stores the password policy and applies it when `changePassword` is called, or when validating the password in the `unlock` method.

If the application has not set a password policy using this method, the data vault does not validate the password in the `createVault` or `changePassword` methods. An exception is thrown if there is any invalid (negative) value in the `passwordPolicy` object.

Syntax

```
public abstract void setPasswordPolicy(DataVault.DVPasswordPolicy
oPasswordPolicy)
```

Parameters

- **oPasswordPolicy –** the password policy constraints.

Examples

- **Set a password policy**

```
// SetPasswordPolicy() locks the vault to ensure the old password
// conforms to the new password policy settings.
oDataVault.setPasswordPolicy( oPasswordPolicy );
```

Password Policy Structure

A structure defines the policy used to generate the password.

Table 1. Password Policy Structure

Name	Type	Description
defaultPasswordAllowed	Boolean	Indicates if client application is allowed to use default password for the data Vault. If this is set to TRUE and if client application uses default password then minLength, hasDigits, hasUpper, hasLower and hasSpecial parameters in the policy are ignored.
minimumLength	Integer	The minimum length of the password.
hasDigits	Boolean	Indicates if the password must contain digits.
hasUpper	Boolean	Indicates if the password must contain uppercase characters.
hasLower	Boolean	Indicates if the password must contain lowercase characters.
hasSpecial	Boolean	Indicates if the password must contain special characters. The set of special characters is: "~!@#$%^&*()-+".
expirationDays	Integer	Specifies password expiry days from the date of setting the password. 0 indicates no expiry.
minUniqueChars	Integer	The minimum number of unique characters in the password. For example, if length is 5 and minUniqueChars is 4 then "aaate" or "ababa" would be invalid passwords. Instead, "aaord" would be a valid password.

Name	Type	Description
lockTimeout	Integer	The timeout value (in seconds) after which the vault will be locked from the unlock time. 0 indicates no timeout. This value overrides the value set by `setLockTimeout` method.
retryLimit	Integer	The number of failed unlock attempts after which data vault is deleted. 0 indicates no retry limit. This value overrides the value set by the `setRetryLimit` method.

Settings for Password Policy

The client applications use these settings to fill the PasswordPolicy structure. The default values are used by the data vault when no policy is configured. The defaults are also used in SAP Control Center in the default template. The SAP Mobile Platform administrator can modify these settings through SAP Control Center. The application must set the password policy for the data vault with the administrative (or alternative) settings.

Note: Setting the password policy locks the vault. The password policy is enforced when `unlock` is called (because the password is not saved, calling `unlock` is the only time that the policy can be evaluated).

- **PROP_DEF_PWDPOLICY_ENABLED –** Boolean property with a default value of false. Indicates if a password policy is enabled by the administrator.
- **PROP_DEF_PWDPOLICY_DEFAULT_PASSWORD_ALLOWED –** Boolean property with a default value of false. Indicates if the client application is allowed to use the default password for the data vault.
- **PROP_DEF_PWDPOLICY_MIN_LENGTH –** Integer property with a default value of 0. Defines the minimum length for the password.
- **PROP_DEF_PWDPOLICY_HAS_DIGITS –** Boolean property with a default value of false. Indicates if the password must contain digits.
- **PROP_DEF_PWDPOLICY_HAS_UPPER –** Boolean property with a default value of false. Indicates if the password must contain at least one uppercase character.
- **PROP_DEF_PWDPOLICY_HAS_LOWER –** Boolean property with a default value of false. Indicates if the password must contain at least one lowercase character.
- **PROP_DEF_PWDPOLICY_HAS_SPECIAL –** Boolean property with a default value of false. Indicates if the password must contain at least one special character. A special character is a character in this set “~!@#$%^&*()-+”.

- **PROP_DEF_PWDPOLICY_EXPIRATION_DAYS –** Integer property with a default value of 0. Specifies the number of days in which password will expire from the date of setting the password. Password expiration is checked only when the vault is unlocked.
- **PROP_DEF_PWDPOLICY_MIN_UNIQUE_CHARS –** Integer property with a default value of 0. Specifies minimum number of unique characters in the password. For example, if minimum length is 5 and minUniqueChars is 4 then “aaate” or “ababa” would be invalid passwords. Instead, “aaord” would be a valid password.
- **PROP_DEF_PWDPOLICY_LOCK_TIMEOUT –** Integer property with a default value of 0. Specifies timeout value (in seconds) after which the vault is locked from the unlock time. 0 indicates no timeout.
- **PROP_DEF_PWDPOLICY_RETRY_LIMIT –** Integer property with a default value of 0. Specifies the number of failed unlock attempts after which data vault is deleted. 0 indicates no retry limit.

Password Errors

Password policy violations cause exceptions to be thrown.

Table 2. Password Errors

Name	Value	Description
PASSWORD_REQUIRED	50	Indicates that a blank or null password was used when the password policy does not allow default password.
PASSWORD_UNDER_MIN_LENGTH	51	Indicates that the password length is less than the required minimum.
PASSWORD_REQUIRES_DIGIT	52	Indicates that the password does not contain digits.
PASSWORD_REQUIRES_UPPER	53	Indicates that the password does not contain upper case characters.
PASSWORD_REQUIRES_LOWER	54	Indicates that the password does not contain lower case characters.
PASSWORD_REQUIRES_SPECIAL	55	Indicates that the password does not contain one of these special characters: ~!@#$%^&*()-+.

Name	Value	Description
PASSWORD_UNDER_MIN_UNIQUE	56	Indicates that the password contains fewer than the minimum required number of unique characters.
PASSWORD_EXPIRED	57	Indicates that the password has been in use longer than the number of configured expiration days.

getPasswordPolicy

Retrieves the password policy set by `setPasswordPolicy`.

Use this method once the DataVault is unlocked.

Syntax

```
public abstract DataVault.DVPasswordPolicy getPasswordPolicy()
```

Parameters

None.

Returns

Returns a `passwordPolicy` structure that contains the policy set by `setPasswordPolicy`.

Returns a `DVPasswordPolicy` object with the default values if no password policy is set.

Examples

- **Get the current password policy**

```
// Call getPasswordPolicy() to return the current password policy
settings.
      DataVault.DVPasswordPolicy oCurrentPolicy =
oDataVault.getPasswordPolicy();
```

isDefaultPasswordUsed

Checks whether the default password is used by the vault.

Use this method once the DataVault is unlocked.

Syntax

```
public boolean isDefaultPasswordUsed()
```

Returns

Returns	Indicates
true	Both the default password and the default salt are used to encrypt the vault.
false	Either the default password or the default salt are not used to encrypt the vault.

Examples

- **Check if default password used**

```
// Call isDefaultPasswordused() to see if we are using an
automatically
// generated password (which we are).
boolean isDefaultPasswordUsed =
oDataVault.isDefaultPasswordUsed();
```

  This code example lacks exception handling. For a code example that includes exception handling, see *Developer Guide: Android Object API Applications> Client Object API Usage > Security APIs > DataVault > Code Sample*.

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, `lock` has no effect.

Syntax

```
public void lock()
```

Examples

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

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

isLocked

Checks whether the vault is locked.

Syntax

```
public boolean isLocked()
```

Returns

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 attempts exceeds the retry limit, the vault is deleted.

The password is validated against the password policy if it has been set using `setPasswordPolicy`. If the password is not compatible with the password policy, an `IncompatiblePassword` exception is thrown. In that case, call `changePassword` to set a new password that is compatible with the password policy.

Syntax

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

Parameters

- **password –** the encryption password for this DataVault. If null is passed, a default password is computed and used.
- **salt –** the encryption salt value for this DataVault. This value, combined with the password, creates the actual encryption key that protects the data in the vault. This value may be an application-specific constant. If null is passed, a default salt is computed and used.

Returns

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

Examples

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

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

setString

Stores a string object in the vault.

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 ensures 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.

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

Returns a string data value, associated with the specified name, from the vault.

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 ensures 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.

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 ensures 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.

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

Returns a binary data value, associated with the specified name, from the vault.

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 ensures 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();
}
```

deleteValue

Deletes the specified value.

Syntax

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

Parameters

- **name –** the name of the value to be deleted.

Examples

- **Delete a value –** deletes a value called `myValue`.

  ```
  DataVault.deleteValue("myValue");
  ```

changePassword (two parameters)

Changes the password for the vault. Use this method when the vault is unlocked.

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 ensures 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();
  }
  ```

changePassword (four parameters)

Changes the password for the vault. Use this method when the vault is locked

This overloaded method ensures the new password is compatible with the password policy, uses the current password to unlock the vault, and changes the password of the vault to a new password. If the current password is not valid an `InvalidPassword` exception is thrown. If the new password is not compatible with the password policy set in `setPasswordPolicy` then an `IncompatiblePassword` exception is thrown.

Syntax

```
public abstract void changePassword(string sCurrentPassword,
    string sCurrentSalt,
    string sNewPassword,
    string sNewSalt)
```

Parameters

- **currentPassword –** the current encryption password for this data vault. If a null value is passed, a default password is computed and used.
- **currentSalt –** the current encryption salt value for this data vault. If a null value is passed, a default password is computed and used.
- **newPassword –** the new encryption password for this data vault. If a null value is passed, a default password is computed and used.
- **newSalt –** the new encryption salt value for this data vault. This value, combined with the password, creates the actual encryption key that protects the data in the vault. This value may be an application-specific constant. If a null value is passed, a default password is computed and used.

Examples

- **Change the password for a data vault**

```
// Call changePassword with four parameters, even if the vault is
locked.
// Pass null for oldSalt and oldPassword if the defaults were
used.
oDataVault.changePassword( null, null, "password!1A",
"saltD#ddg#k05%gnd[!1A" );
```

Code Sample

Create a data vault for encrypted storage of application data.

```
public void testFunctionality(Context oContext)
  {
    try
    {
       DataVault oDataVault = null;
```

```
       DataVault.init( oContext );

       // If this dataVault already exists, then get it by calling
getVault()
       // Else create this new dataVault by calling createVault()
       if ( DataVault.vaultExists( "DataVaultExample" ) )
         oDataVault = DataVault.getVault( "DataVaultExample" );
       else
         oDataVault = DataVault.createVault( "DataVaultExample",
"password!1A", "saltD#ddg#k05%gnd[!1A" );

      // Call setLockTimeout(). This allows you to set the timeout of
the vault in seconds
       oDataVault.setLockTimeout( 1500 );
       int iTimeout = oDataVault.getLockTimeout();

      // Call setRetryLimit(). This allows you to set the number of
retries before the vault is destroyed
       oDataVault.setRetryLimit( 10 );
       int iRetryLimit = oDataVault.getRetryLimit();

      // Call setPasswordPolicy(). The passwordPolicy also includes
the retryLimit and LockTimeout that we set above.
       DataVault.DVPasswordPolicy oPasswordPolicy = new
DataVault.DVPasswordPolicy();
       oPasswordPolicy.setIsDefaultPasswordAllowed (true);
       oPasswordPolicy.setMinLength( 4 );
       oPasswordPolicy.setHasDigits( true );
       oPasswordPolicy.setHasUpper( true );
       oPasswordPolicy.setHasLower( true );
       oPasswordPolicy.setHasSpecial( true );
       oPasswordPolicy.setExpirationDays( 20 );
       oPasswordPolicy.setMinUniqueChars( 3 );
       oPasswordPolicy.setLockTimeout( 1600 );
       oPasswordPolicy.setRetryLimit( 20 );

      // SetPasswordPolicy() will always lock the vault to ensure the
old password
       // conforms to the new password policy settings.
       oDataVault.setPasswordPolicy( oPasswordPolicy );

      // We are now locked and need to unlock before we can access the
vault.
       oDataVault.unlock( "password!1A", "saltD#ddg#k05%gnd[!1A" );

       // Call getPasswordPolicy() to return the current password
policy settings.
       DataVault.DVPasswordPolicy oCurrentPolicy =
oDataVault.getPasswordPolicy();

      // Call setString() by giving it a name:value pair to encrypt
and persist
       // a string data type within your dataVault.
       oDataVault.setString( "stringName", "stringValue" );

      // Call getString to retrieve the string we just stored in our
```

```
data vault!
       String storedStringValue =
oDataVault.getString( "stringName" );

       // Call setValue() by giving it a name:value pair to encrypt
and persist
       // a binary data type within your dataVault.
       byte[] binaryValue = { 1, 2, 3, 4, 5, 6, 7  };
       oDataVault.setValue( "binaryName", binaryValue );

      // Call getValue to retrieve the binary we just stored in our
data vault!
       byte[] storedBinaryValue =
oDataVault.getValue( "binaryName" );

      // Call getDataNames to retrieve all stored element names from
our data vault.
       DataVault.DVDataName[] dataNameArray =
oDataVault.getDataNames();
       for ( int i = 0; i < dataNameArray.length; i++ )
       {
          if ( dataNameArray[i].getType() ==
DataVault.DV_DATA_TYPE_STRING )
          {
             String thisStringValue =
oDataVault.getString( dataNameArray[i].getName() );
          }
          else
          {
             byte[] thisBinaryValue =
oDataVault.getValue( dataNameArray[i].getName() );
          }
       }

       // Call changePassword with 2 parameters. Vault must be
unlocked.
      // If you pass null parameters as your new password or your new
salt,
       // it will generate a default password or default salt,
respectively.
       oDataVault.changePassword( null, null );

       // Call isDefaultPasswordused() to see if we are using an
automatically
       // generated password (which we are).
       boolean isDefaultPasswordUsed =
oDataVault.isDefaultPasswordUsed();

       // Lock the vault.
       oDataVault.lock();

      // Call changePassword with 4 parameters even if the vault is
locked.
      // Here, we pass null for oldSalt and oldPassword because
defaults were used.
       oDataVault.changePassword( null, null, "password!1A",
```

```
"saltD#ddg#k05%gnd[!1A" );

       // Call isDefaultPasswordused() and we will see that the
default password is NOT used anymore.
       isDefaultPasswordUsed = oDataVault.isDefaultPasswordUsed();
    }
    catch( Throwable exception )
    {
      exception.printStackTrace();
    }
    finally
    {
      try
       {
       // Because this is a test example, we will delete our vault at
the end.
        // This means we will forever lose all data we persisted in
our data vault.
         if ( DataVault.vaultExists( "DataVaultExample" ) )
           DataVault.deleteVault( "DataVaultExample" );
       }
       catch(Throwable t)
       {
         t.printStackTrace();
       }
    }
  }
```

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 37

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 3. 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. 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:** Only the `CallbackHandler` registered for package DB will be invoked.
`void onLoginSuccess()`	This method is invoked when login succeeds for a `beginOnlineLogin` call. **Note:** Only the CallbackHandler registered for package DB is invoked.

Callback	Description
`void onReplayFailure(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 onReplaySuccess(java.lang.Object entity)`	This method is invoked when a replay request succeeds. `onReplaySuccess` is an MBO object instance that contains the data prior to the synchronization. You can use the Change Log API to find records that occur after the synchronization. **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 onSearchFailure(java.lang.Object entity)`	This method is invoked when a back-end search fails. **Note:** CallbackHandlers registered for both the MBO class of the entity and the Package DB are invoked. **Parameters**: • **entity** – the back-end search object.
`void onSearchSuccess(java.lang.Object entity)`	This method is invoked when a back end search succeeds. **Note:** CallbackHandlers registered for both the MBO class of the entity and the Package DB are invoked. **Parameters**: • **entity** – the back-end search object.

Callback	Description
`void onSubscribeFailure()`	This method is invoked when subscribe fails. **Note:** CallbackHandlers registered for both the MBO class of the entity and the Package DB are invoked.
`void onSubscribeSuccess()`	This method is invoked when subscribe succeeds. **Note:** Only the CallbackHandler registered for the Package DB is invoked.
`int onSynchronize(GenericList<SynchronizationGroup> groups, SynchronizationContext context)`	This method is invoked at different stages of the synchronization. This method is called by the database class `synchronize` or `beginSynchronize` methods when the client initiates a synchronization, and is called again when the server responds to the client that synchronization has finished, or that synchronization failed. The status of the synchronization context, `context.Status`, 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.

Callback	Description
`void onSuspendSubscriptionFailure()`	This method is invoked when suspend subscription fails. **Note:** Only the CallbackHandler registered for the Package DB is invoked.
`void onSuspendSubscriptionSuccess()`	This method is invoked when suspend subscription succeeds. **Note:** Only the CallbackHandler registered for the Package DB is invoked.
`void onResumeSubscriptionFailure()`	This method is invoked when resume subscription fails. **Note:** Only the CallbackHandler registered for the Package DB is invoked.
`void onResumeSubscriptionSuccess()`	This method is invoked when resume subscription succeeds. **Note:** Only the CallbackHandler registered for the Package DB is invoked.
`void onUnsubscribeFailure()`	This method is invoked when unsubscribe fails. **Note:** Only the CallbackHandler registered for the Package DB is invoked.
`void onUnsubscribeSuccess()`	This method is invoked when unsubscribe succeeds. **Note:** Only the CallbackHandler registered for the Package DB is invoked.

Callback	Description
`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:** Only the CallbackHandler registered for the Package DB is invoked.
`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:** Only the CallbackHandler registered for the Package DB is invoked.
`void onResetSuccess()`	This method is invoked when all data is cleared by the reset. **Note:** Only the CallbackHandler registered for the Package DB is invoked.
`void onRecoverSuccess()`	This method is invoked when recover succeeds. **Note:** Only the CallbackHandler registered for the Package DB is invoked.
`void onRecoverFailure()`	This method is invoked when recover fails. **Note:** Only the CallbackHandler registered for the Package DB is invoked.

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:** Only the CallbackHandler registered for the Package DB is invoked.
`void onImportSuccess()`	This method is invoked when all data has been successfully imported. **Note:** Only the CallbackHandler registered for the Package DB is invoked.
`void beforeImport(java.lang.Object entity)`	This method is invoked before importing the specified entity. **Note:** Only the CallbackHandler registered for the Package DB is invoked. **Parameters**: • **entity** – the Mobile Business Object to be imported.
`void onMessageStart(int size, String method, String mbo);`	This method is called at the beginning of processing a message from the server, before the message transaction starts. Only the callback handler registered with the package database class is invoked. Parameters: • **size** – The size of the incoming message content in bytes. • **method** – The method string from the message header. • **mbo** – If this message is for a specific MBO, the name of the MBO; otherwise null. This method is for DOE-based applications only.

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.

Note: These callbacks are not triggered by changes or errors in Mobilink synchronization, which uses a different communication path than the one used for registration.

Table 4. 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. **Note:** Some of the connection status codes are not returned on certain client platforms due to platform operating system limitations.
`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.

Callback	Description
`void onHttpCommunicationError(int errorCode, String errorMessage, StringProperties httpHeaders);`	Invoked when an HTTP communication server/MobiLink rejects HTTP/MobiLink communication with an error code. • **errorCode –** Error code returned by the HTTP server or MobiLink. For example: code 401 for authentication failure, code 403 for authorization failure, and code 63 for MobiLink synchronization communication error. • **errorMessage –** Error message returned by the HTTP server or MobiLink. • **httpHeaders –** Response headers returned by the HTTP server or MobiLink.

`void onCustomizationBundleDownloadComplete(String customizationBundleID, int errorCode, String errorMessage);`	Invoked when the download of a resource bundle is complete.
	• **errorCode –** If download succeeds, returns 0. If download fails, returns an error code.
	• **errorMessage –** If download succeeds, returns "". If download fails, returns an error message.
	  • RESOURCE_BUNDLE_NOTFOUND = 14881
	  • DOWNLOAD_RESOURCE_BUNDLE_STREAM_IS_NULL = 14882
	  • DOWNLOAD_RESOURCE_BUNDLE_FAILURE = 14883
	• **customizationBundleID –** The name of the resource bundle. If null, the default application resource bundle is downloaded.

Callback	Description

`int onPushNotification (Hashtable notification);`	Invoked if a push notification arrives. You can add logic here to handle the notification. This callback is not called when a notification arrives when the application is not online.
	• **returns –** an integer to indicate if the notification has been handled. The return value is for future use. You are recommended to return NOTIFICATION_CONTINUE.
	  • 0: NOTIFICATION_CONTINUE if the notification was not handled by the callback method.
	  • 1: NOTIFICATION_CANCEL if the notification has already been handled by the callback method.

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();
SMP101DB.synchronize("sync_group", listener);
// or SMP101DB.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.

The method returns `false` to allow synchronization to continue. If the method returns true, the synchronization is aborted.

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 47

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 SAP Mobile WorkSpace. Object Query methods carry query names, parameters, and return types defined in SAP Mobile 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 SAP Mobile 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 5. 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.setValue("Beth");
query2.where(ts);
QueryResultSet qrs = SMP101DB.executeQuery(query2);
```

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
- NOT_START_WITH
- NOT_END_WITH
- 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 = SMP101DB.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
`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");
```

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);
SMP101DB.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 =
SMP101DB.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 =
SMP101DB.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);
```

Note: "Order By" is not supported for a long varchar field.

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 = SMP101DB.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 SAP Mobile 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 50

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 SAP Mobile 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 MBO Records* on page 51
- *Other Operations* on page 52

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( "SAP" );
cust.setPhone( "777-8888" );
cust.create();// or cust.save();
cust.submitPending();
SMP101DB.synchronize();
// or SMP101DB.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("SAP");
cust.setPhone("777-8888");
cust.save(); // or cust.update();
cust.submitPending();
SMP101DB.synchronize();
// or SMP101DB.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();
SMP101DB.synchronize();
// or SMP101DB.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();
SMP101DB.synchronize(); // or SMP101DB.synchronize (String
synchronizationGroup)
```

Pending Operation

You can manage the pending state.

- **submitPending –** submits the operation so that it can be replayed on the SAP Mobile Server. A request is sent to the SAP Mobile Server during a synchronization.
- **cancelPending –** cancels the previous `create`, `update`, or `delete` operations on the MBO. It cannot cancel submitted operations.

  `cancelPending` cancels pending changes for a particular instance or instances (via `cancelPendingObjects` from the database class). However, if `submitPending` has already been invoked, only the pending state and original state (for update) are removed. The operation replay record generated by the `submitPending` remains. This means that the operation replay record is uploaded to SAP Mobile Server upon synchronization. If the EIS honors the operation replay, the changes are propagated back to the device during the download. The Object API framework forgoes operation replay completion processing when it finds that there are no pending/original states for the instance. Hence, `cancelPending` is not the inverse operation of submitPending.
- **submitPendingOperations –** submits all the pending records for the entity to the SAP Mobile Server. This method internally invokes the `submitPending` method on each of the pending records.
- **cancelPendingOperations –** cancels all the pending records for the entity. This method internally invokes the `cancelPending` method on each of the pending records.

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

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

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 =
SMP101DB.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 SAP Mobile 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.
`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).

Name	Type	Description
`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.
`isPending`	`boolean`	Checks if the object's pending flag is turned on or not, i.e. has pending change or not. Returns true if there is a pending change, returns false if there is no pending change.
`isUpdated`	`boolean`	Returns true if this entity has been updated or changed in the database, and one of the following is true: • The entity has not yet been submitted to the server with a replay request. • The entity has been submitted to the server, but the server has not finished processing the request. • The server rejected the replay request (`replayFailure` message received).
`pending`	`boolean`	Returns true for any row that represents a pending `create`, `update`, or `delete` operation, or a row that has cascading children with a pending operation.
`pendingChange`	char	If pending is true, 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 `KeyGenerator.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 SAP Mobile Server is the value in the database before you call `submitPending`. After a successful replay, that value is imported from the SAP Mobile 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 SAP Mobile 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 SAP Mobile 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 SAP Mobile 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("SAP");
```

```
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 = SMP101DB.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

- **Close the value stream –** Writes a binary book cover image and closes the image file. In the following example, `book` is the instance of an MBO and `cover` is a BigBinary attribute

  ```
  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

- **Open for reading –** 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

- **Open for writing –** 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

- **Read –** 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

- **Write data –** 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

- **Close the value stream –** 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

- **Open for reading –** 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

- **Open for writing –** 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

- **Read –** 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

- **Write data –** 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 API

You can access metadata for database, classes, entities, attributes, operations, and parameters using the MetaData API.

MetaData 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 API.

These APIs allow retrieving the metadata of packages, MBOs, attributes, operations, and parameters during runtime.

MetaData classes are generated automatically.

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 = SMP101DB.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");
...
```

EntityMetaData

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

```
EntityMetaData customerMetaData = Customer.getMetaData();
    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. These sections do not contain error codes contained in the exception classes. See the Developer Guide: Device Client Error Reference for detailed information about SAP Mobile Platform error codes.

Exception Handling

An exception represents an unexpected condition hindering a method from completion. In some cases, the exception is transient and you can retry it at a later time. In most cases, you must resolve the underlying cause of the exception to allow the API to complete successfully. In rare cases, the exception encountered corrupts the application state and may require you to terminate and restart the application.

To use the localization features in exception handling:

- Register an exception message service implementation through the `ServiceRegistry`.

Base Exceptions

A base exception class is defined as the super class for all external exceptions. Specific exceptions always inherit from the base exception. To enable you, the Object API developer, to write a standard exception handler, all external exceptions have an error code and a single error message. Furthermore, the exception may contain another exception as the cause. See the Developer Guide: Device Client Error Reference for detailed information.

You can use the `getLocalizedMessage (String localeName)` method to retrieve an error message for a specified locale.

See the *Object API Applications* section of the *Developer Guide: Device Client Error Reference* for information about possible error codes and the corresponding error messages.

Exception Message Service

You can implement an exception message service for resolving localized messages using error codes. The exception class uses the exception message service to load resource bundles and look up error messages based on an error code. You can use a default message provider, `ExceptionMessageServiceImpl`, or create a custom provider by implementing your own `ExceptionMessageService`.

To resolve localized messages, implement the `ExceptionMessageService` interface.

```
public class CustomExceptionMessageService implements
ExceptionMessageService
{
  public String getMessage(int errorCode)
  {
    String msg = null;

    msg = "getMessage(" + errorCode + ")";

    return msg;
  }

  public String getMessage(int errorCode, String localeName)
  {
    String msg = null;

    msg = "getMessage(" + errorCode + "," + localeName + ")";

    return msg;
  }
}
```

A default implementation, `ExceptionMessageServiceImpl` allows the default English resource to look up an error message using an error code. You can follow these steps to add other localized resources for the Android platform without implementing a custom message service.

1. Get the default property file "SUPErrorMessage_en.properties" (included in the `resources` folder in the Mobile SDK for the Android platform) and localize it to another language file, for instance, "SUPErrorMessage_de.properties".
2. Add the new property file into the `src` folder of the application.
3. Register the default implementation "ExceptionMessageServiceImpl" in the application code:

   ```
   ServiceRegistry.getInstance().registerService(ExceptionMessageSer
   vice.class,
   ExceptionMessageServiceImpl.getInstance());
   ```
4. Use the localized error message in the application.
5. You can unregister the exception message service to cancel the use of the localized error message:

   ```
   ServiceRegistry.getInstance().unregisterService(ExceptionMessageS
   ervice.class);
   ```

Service Registry

You can register objects that implement the `ExceptionMessageService` interface using the `ServiceRegister` interface's `registerService` and `unregisterService` methods.

```
ServiceRegistry.getInstance().registerService(com.sybase.mobile.fra
mework.ExceptionMessageService.class, new
CustomExceptionMessageService());
...
ServiceRegistry.getInstance().unregisterService(com.sybase.mobile.f
ramework.ExceptionMessageService.class);
```

Example Code for Handling Exceptions

An example of registering your interface.

```
// Register ExceptionMessageServiceImpl
ServiceRegistry.getInstance().registerService(com.sybase.mobile.fra
mework.ExceptionMessageService.class,
ExceptionMessageServiceImpl.getInstance());
try
{
  // throw com.sybase.persistence.ObjectNotFoundException
}
catch (ObjectNotFoundException e)
{
  if (e.ErrorCode == ObjectNotFoundException.VALUE_IS_NULL)
  {
    String msg = e.getMessage();
    msg = e.getLocalizedMessage("fr");
    msg = e.getLocalizedMessage("de");
    msg = e.getLocalizedMessage("es");
  }
}
finally
{
  // Unregister ExceptionMessageServiceImpl
```

```
ServiceRegistry.getInstance().unregisterService(com.sybase.mobile.f
ramework.ExceptionMessageService.class);
}

// Register CustomExceptionMessageService
ServiceRegistry.getInstance().registerService(com.sybase.mobile.fra
mework.ExceptionMessageService.class, new
CustomExceptionMessageService());
try
{
  // throw com.sybase.persistence.ObjectNotFoundException
}
catch (ObjectNotFoundException e)
{
  if (e.ErrorCode == ObjectNotFoundException.VALUE_IS_NULL)
  {
    String msg = e.getMessage();
    msg = e.getLocalizedMessage("fr");
    msg = e.getLocalizedMessage("de");
    msg = e.getLocalizedMessage("es");
  }
}
finally
{

ServiceRegistry.getInstance().unregisterService(com.sybase.mobile.f
ramework.ExceptionMessageService.class);
}
```

Server-Side Exceptions

A server-side exception occurs when a client tries to update or create a record and the SAP Mobile 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.

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.

- **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
- **ApplicationTimeoutException –** thrown when a call to start the connection, register the application, or unregister the application times out.
- **LoginRequiredException –** thrown when the client application does not login to the server.
- **NoSuchOperationException –** thrown when trying to access operation metadata that does not exist in class metadata.
- **NoSuchAttributeException –** thrown when trying to access an attribute that does not exist in class or entity metadata and thrown by a dynamic query method (ExecuteQuery, etc.) if the Query passed in selects for an attribute that does not exist in the MBO queried.
- **ObjectNotFoundException –** thrown when trying to load an MBO that is not inside the local database.
- **ObjectNotSavedException –** thrown when a BigBinary or BigString method is called that requires the object to already exist in the database.
- **PersistenceException –** thrown when trying to access the local database.
- **ProtocolException –** thrown when an exception occurs during protocol version mismatch.
- **StreamNotOpenException –** thrown when a BigBinary or BigString method is called that requires the object to be open.
- **StreamNotClosedException –** thrown when a BigBinary or BigString method is called that requires the object to not be open.
- **SynchronizeException –** thrown when an error occurs during synchronization.
- **SynchronizeRequiredException –** thrown when synchronization is needed.
- **WriteAppendOnlyException –** thrown if a BigBinary or BigString method is called that writes to the middle of a value where only appending is allowed by the underlying database.
- **WriteOverLengthException –** thrown if the platform requires the length to be predetermined before write and a BigBinary or BigString method is called that writes past the predetermined length.

Error Codes

Codes for errors occuring during application execution.

HTTP Error Codes

The SAP Mobile 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 SAP Mobile Platform internal error, or an unrecognized EIS error.

The EIS code and HTTP error code values are stored in log records (`LogRecord.EisCode`, and `LogRecord.Code`, respectively).

These tables list recoverable and unrecoverable error codes. All error codes that are not explicitly considered recoverable are considered unrecoverable.

Table 6. Recoverable Error Codes

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

Table 7. 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 the SAP Mobile 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	SAP Mobile 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 SAP 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 8. 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

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

X