SYBASE®
An SAP® Company

Developer Guide: iOS Object API Applications

Sybase Unwired Platform 2.1 ESD #2

DOCUMENT ID: DC01217-01-0212-02
LAST REVISED: July 2012

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

Contents

Getting Started with iOS Development

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

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

Companion guides include:

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

  Complete the tutorials to gain a better understanding of Unwired Platform components and the development process.
- *Troubleshooting for Sybase Unwired Platform*.
- The iOS HeaderDoc provides a complete reference to the APIs:
  - The Framework Library HeaderDoc is installed to `<UnwiredPlatform_InstallDir>\MobileSDK\ObjectAPI\iOS\headerdoc`. For example, `C:\Sybase\UnwiredPlatform\MobileSDK\ObjectAPI\iOS\headerdoc`.
  - You can generate HeaderDoc from the generated Objective-C code. See *http://developer.apple.com/mac/library/navigation/index.html*.
- *Fundamentals* contains high-level mobile computing concepts, and a description of how Sybase Unwired Platform implements the concepts in your enterprise.

Object API Applications

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

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

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

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

- Server-side components address the interaction between the enterprise information system (EIS) data source and the data cache. EIS data subsets and business logic are encapsulated in artifacts, called mobile business object or DOE packages, that are deployed to Unwired Server.
- Client-side components are built into the mobile application and address the interaction between the data cache and the mobile device data store. This can include synchronizing data with the server, offline data access capabilities, and data change notification.

These applications:

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

Best Uses for Object API Applications

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

Cache Synchronization

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

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

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

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

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

Client Runtime Architecture

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

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

Mobile Channel Interfaces

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

- The messaging channel serves as the abstraction to all device-side notifications (BlackBerry Enterprise Service, Apple Push Notification Service, and others) so that when changes to back-end data occur, devices can be notified of changes relevant for their application and configuration. This channel also enables data synchronization on iOS. The messaging channel sends these types of notifications:
  - Change notifications - when Unwired Server detects changes in the back-end EIS, Unwired Server can send a notification to the device. By default, sending change notifications is disabled, but you can enable sending change notifications per synchronization group.
    To capture change notifications, you can register an onSynchronize callback. The synchronization context in the callback has a status you can retrieve.
  - When synchronizing, operation replay records are sent to the Unwired Server and the messaging channel sends a notification of `replayFinished`. The application must call another synchronize method to retrieve the result.
- The synchronization channel sends data to keep the Unwired Server and client synchronized. The synchronization is bi-directional.

Mobile Middleware Services

Mobile middleware services (MMS) arbitrate and manage communications between device requests from the mobile channel interfaces in the form that is suitable for transformation to a

common MBO service request and a canonical form of enterprise data supplied by the data services.

Data Services

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

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

Documentation Roadmap for Unwired Platform

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

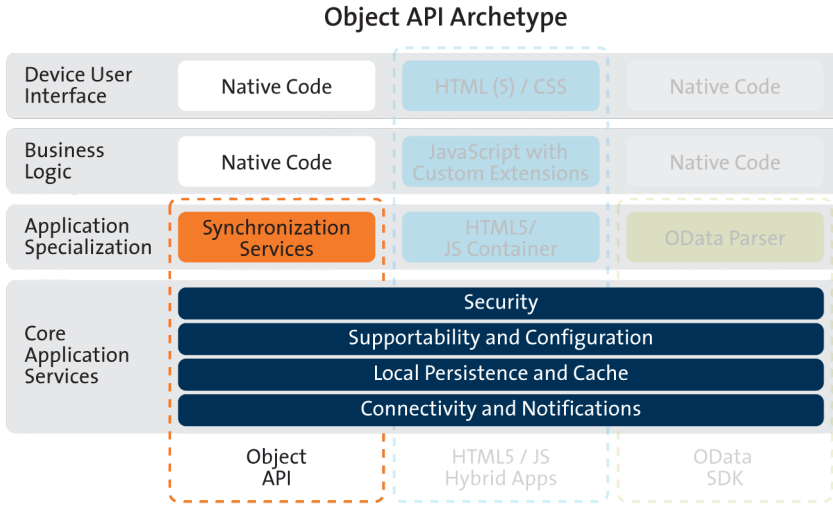
See *Documentation Roadmap* in *Fundamentals* for document descriptions by user role. *Fundamentals* is available on the Sybase Product Documentation Web site.

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

Development Task Flow for Object API Applications

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

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



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

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

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

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

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

Installing the iOS Development Environment

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

Downloading the Xcode IDE

Download and install Xcode.

1. Download Xcode from the Apple Web site: *http://developer.apple.com/xcode/*.
2. Close the iTunes program before beginning the installation.
3. Complete the Xcode installation following the instructions in the installer.

Downloading Older Versions of the Xcode IDE

If you do not have the supported version of Xcode and the iOS SDK, you need to download it from the Downloads for Apple Developers Web site.

See *Supported Hardware and Software* for the most current version information for mobile device platforms and third-party development environments. If necessary, you can download older versions.

1. Go to *http://developer.apple.com/downloads/*.

   You must be a paying member of the iOS Developer Program. Free members do not have access to the supported version.
2. Log in using your Apple Developer credentials.
3. (Optional) Deselect all Categories except Developer Tools to narrow the search scope.
4. Download the supported Xcode and SDK combination.

Installing X.509 Certificates on iOS Clients

Install generated X.509 certificates and test them in your iOS clients. A certificate provides an additional level of secure access to an application, and may be required by an organization's security policy.

Creating a Project

Build a device application project.

Generating HeaderDoc from Generated Code

Once you have generated Objective-C code for your mobile business objects, you can generate HeaderDoc (HTML reference information) on the Mac from the generated code. HeaderDoc

provides reference information for the MBOs you have designed. The HeaderDoc will help you to programmatically bind your device application to the generated code.

1. Navigate to the directory containing the generated code that was copied over from the Eclipse environment.
2. Run:

```
>headerdoc2html –o GeneratedDocDir GeneratedCodeDir
>gatherheaderdoc GeneratedDocDir
```

You can open the file `OutputDir/masterTOC.html` in a Web browser to see the interlinked sets of documentation.

Note: You can review complete details on HeaderDoc in the *HeaderDoc User Guide*, available from the Mac OS X Reference Library at *http://developer.apple.com/mac/library/navigation/index.html*.

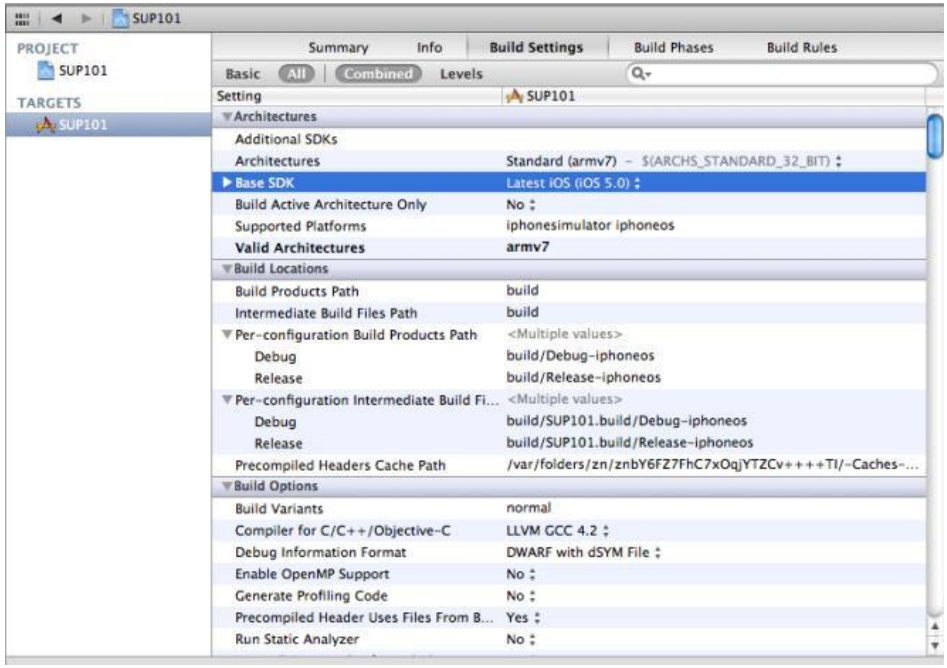
Importing Libraries and Code

Import the generated MBO code and associated libraries into the iOS development environment.

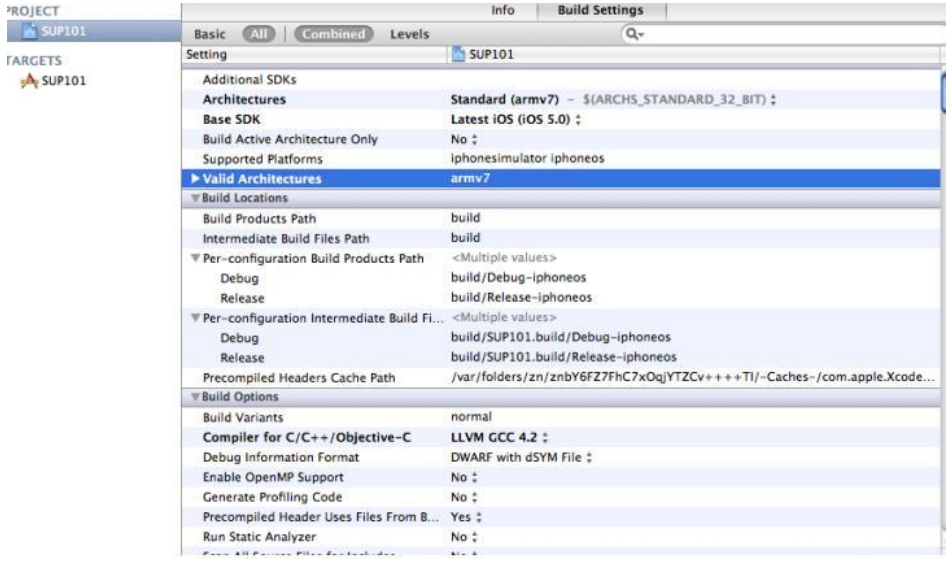
Note: For more information on Xcode, refer to the Apple Developer Connection: *http://developer.apple.com/tools/Xcode/*.

1. Start Xcode 4.2 and select **Create a new Xcode project**.
2. Select **iOS Application** and **Window-based Application** as the project template, and then click **Next**.
3. Enter `<ProjectName>` as the **Product Name**, `MyCorp` as the **Company Identifier**, select **Universal** as the **Device Family** product, and then click **Next**.

   Note: If you will deploy more than one Xcode project with the same application name, the applications will overwrite each other on the device. Ensure that projects do not share the same name even though they have different application IDs.
4. Select the **Architectures** tab, and set Base SDK for All Configurations to `iOS 5.0`.



5. Select the **Deployment** tab and set the iOS Deployment Target to `iOS 5.0` or `iOS 4.3`, as appropriate for the device version where you will deploy. Earlier SDKs and deployment targets are not supported.
6. Select the Architectures as `Standard (armv7)` and the Targeted device family as `iPhone/iPad`. This ensures that the build of the application can run on either iPhone or iPad.



7. Select a location to save the project and click **Create** to open it.

   Xcode creates a folder,`<ProjectName>`, to contain the project file, `<ProjectName>.xcodeproj` and another `<ProjectName>` folder, which contains a number of automatically generated files.

   Copy the files from your Windows machine in to the <ProjectName> folder that Xcode created to contain the generated source code.
8. Connect to the Microsoft Windows machine where Sybase Unwired Platform is installed:
   a) From the Apple Finder menu, select **Go** > **Connect to Server**.
   b) Enter the name or IP address of the machine, for example, `smb://<machine DNS name>` or `smb://<IP Address>`.

      You see the shared directory.
9. Navigate to the `\UnwiredPlatform\MobileSDK\ObjectAPI\iOS` directory in the Unwired Platform installation directory, and copy the `includes` and `Libraries` folders to the `<ProjectName>/<ProjectName>` directory on your Mac.
10. Navigate to the mobile application project (for example, `C:\Documents and Settings\administrator\workspace\<ProjectName>`), and copy the `Generated Code` folder to the `<ProjectName>/<ProjectName>` directory on your Mac.
11. In the Xcode Project Navigator, right-click the `<ProjectName>` folder under the project, select **Add Files to "<ProjectName>"**, select the `Generated Code` folder, unselect **Copy items into destination group's folder (if needed)**, and click **Add**.

    The `Generated Code` folder is added to the project in the Project Navigator.

12. Right-click the `<ProjectName>` folder under the project, select **Add Files to "<ProjectName>"**, navigate to the `<ProjectName/ProjectName>/Libraries/Debug-iphonesimulator` directory, select the `libclientrt.a`, `libSUPObj.a`, `libMO.a`, libsupcore.a and `libAfariaSLL.a` libraries, unselect **Copy items into destination group's folder (if needed)**, and click **Add**.

    The libraries are added to the project in the Project Navigator.

    Note: The library version corresponds to the configuration you are building. For example, if you are building for a debug version of the simulator, navigate to `libs/Debug-iphonesimulator/` to add the libraries.

13. Right-click the project root, select **New Group**, and then rename it to `Resources`.
14. Right-click the `Resources` folder, select **Add Files to "<ProjectName>"**, navigate to the `includes` directory, select the `Settings.bundle` file, unselect **Copy items into destination group's folder (if needed)**, and click **Add**.

    The bundle `Settings.bundle` is added to the project in the Project Navigator.

    This bundle adds resources that lets iOS device client users input information such as server name, server port, user name and activation code in the Settings application.
15. Click the project root and then, in the middle pane, click the **<ProjectName>** project.
    a) In the right pane click the **Build Settings** tab, then scroll down to the **Search Paths** section.
    b) Enter the location of your `includes` folder (`"$SRCROOT/<ProjectName>/includes/**"`) in the **Header Search Paths** field.

    `$SRCROOT` is a macro that expands to the directory where the Xcode project file resides.
    c) Set Automatic Reference Counting (SRC) to NO.
    d) Set the iOS Deployment Target to 5.0 or 4.3.
16. Add the following frameworks from the SDK to your project by clicking on the active target, and selecting **Build Phase** > **Link Binary With Libraries**. Click on the + button and select the following binaries from the list:
    - AddressBook.framework
    - CoreFoundation.framework
    - QuartzCore.framework
    - Security.framework
    - CFNetwork.framework
    - SystemConfiguration.framework
    - MobileCoreServices.framework
    - libicucore.A.dylib
    - libstdc++.dylib
    - libz.1.2.5.dylib

17. Select **Product > Clean** and then **Product > Build** to test the initial set up of the project. If you have correctly followed this procedure, then you should receive a **Build Succeeded** message.
18. Write your application code to reference the generated MBO code. See the *Developer Guide for iOS* for information about referencing the iOS Client Object API.

Importing Libraries and Code for Applications Enabled with ARC

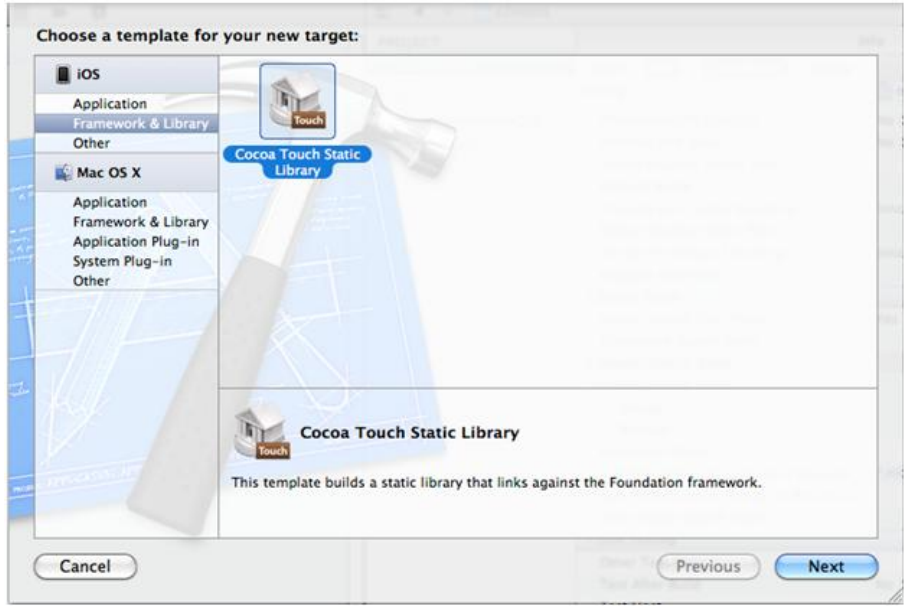
Import the generated MBO code and associated libraries into the iOS development environment, to support applications enabled with automatic reference counting (ARC).

Prerequisites

To use these steps, request a patch from CS&S.

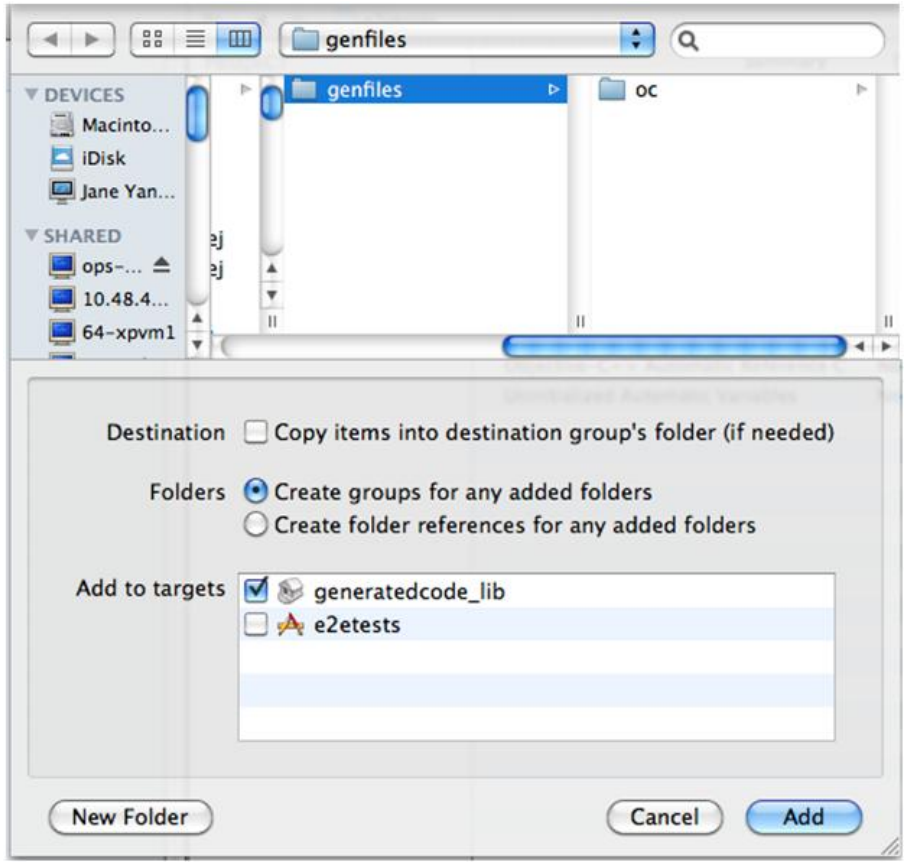
Task

1. Create a non-ARC static library target for the generated code.
   a) Select the application project file in Xcode, and click on **Add Target** at the bottom of the Project Settings screen. When prompted, select the "Cocoa Touch Static Library" template from the Framework & Library section and click **Next**.
   b) Enter the project name with the name you want for your library, for example, "generatedcode_lib". Make sure the "Use Automatic Reference Counting" option is not selected. Click on **Finish**. You have created a second target in your project.



c) Delete the sample class files the wizard created (`generatedcode_lib.h`, and `generatedcode_lib.m`).

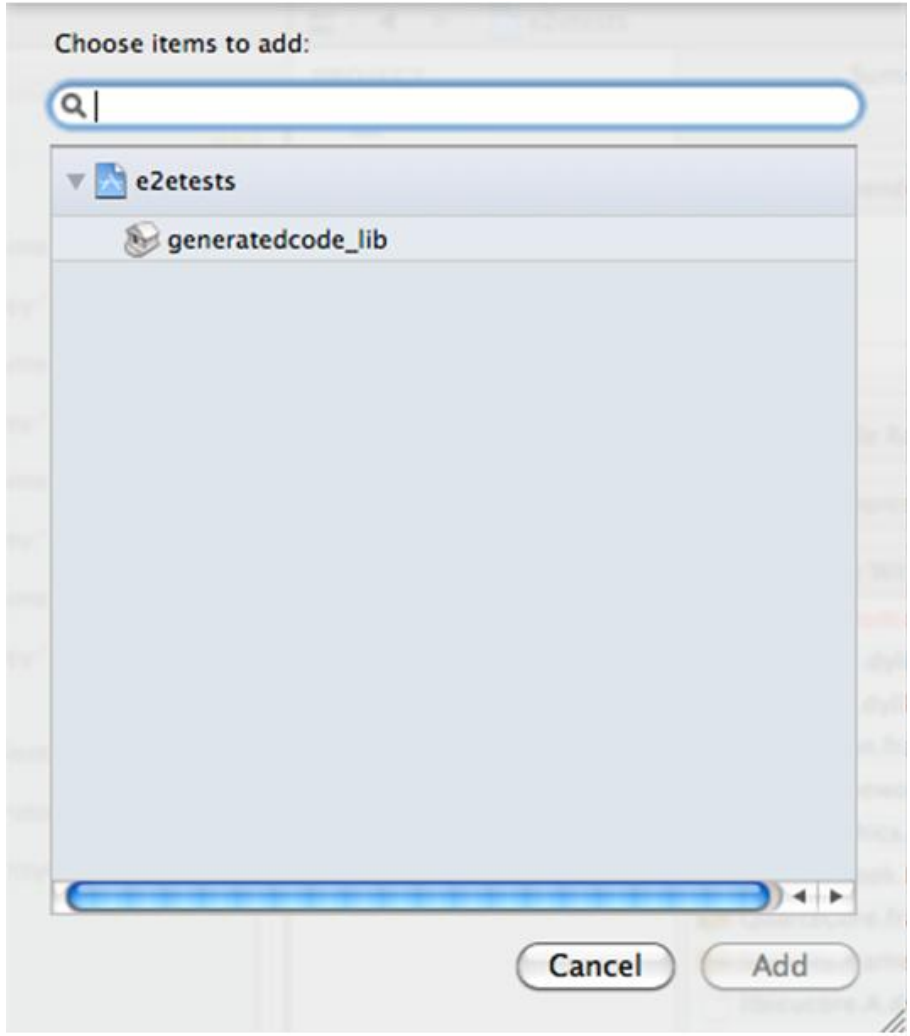
2. Make sure the static library is not using ARC by selecting the `generatedcode_lib` target, going to "Build Settings," and verifying "Automatic Reference Counting" is set to "NO".
3. Add generated code into the static library target.
   a) Right click on the `generatedcode_lib` folder from the Group & File view, and select **Add Files to ...**.
   b) Select your generated code location, and select the option "Add to targets" to "generatedcode_lib". Do not select *<your main target>*.
   c) Click **Add**.



4. Modify the build settings of the static library target.
   a) Select the generatedcode_lib target, and go to "Build Settings", and to "Header Search Paths".

b) Add the location of the SUP client stack `includes` folder. Make sure the "Recursive" checkbox is checked.

5. Link the main application target with the new static library.
   a) Select your main application target, then click on “Build Phase” and expand the “Link Binary With Libraries” section.
   b) Click on the plus (+) button and select the new static library from the list.
6. Add the static library as a dependency.
   a) Select your main application target, then click on “Build Phase” and expand the “Target Dependencies” section.
   b) Click on the plus (+) button and select the new static library from the list.



7. Make sure that ARC is enabled for your main application target.
   a) Select the main target, and go to “Build Settings”.
   b) Verify that Automatic Reference Counting” is set to “YES”.
8. Add your ARC enabled code into the main application target.
9. Import the Sybase Unwired Platform client stack libraries to the main target. Perform the steps in *Developer Guide: iOS Object API Applications > Development Task Flow for Object API Applications > Creating a Project > Importing Libraries and Code*, to import and add only the libraries to the main target. Do not add generated code to the main target, because you have created the secondary static library target with the generated code.
10. Build your ARC-enabled main application target with the Sybase Unwired Platform client stack and generated code.

    Ignore semantic issue warnings during compilation. For example:

```
"Semantic Issue
Type of property ‘databaseName’ does not match type of accessor
‘setDatabaseName:’"
```

Managing the Background State

To allow your application to continue to safely run when it goes into the background, you must implement code in its `AppDelegate` class to ensure that the `SUPApplication` instance's connection to the server shuts down gracefully when going into the background, and starts up when the application becomes active again.

This is important because in iOS, when an application goes into the background, it can have its network sockets invalidated, or the application may be shut down at any time. For correct behavior of the `SUPApplication` connection, the connection needs to be stopped when in background, and only started again when the application goes back to the foreground.

You must implement two `appDelegate` methods: `applicationDidEnterBackground` and `applicationWillEnterForeground`.

Note: The `applicationWillEnterForeground` method is also called when the application first starts up, where most applications would have code already to register the application and start the `SUPApplication` connection. This example code uses a boolean `wasPreviouslyInBackground` so that the `applicationWillEnterForeground` method can detect whether it is called on coming out of the background or is called on a first startup.

```
BOOL wasPreviouslyInBackground = NO;

- (void)applicationDidEnterBackground:(UIApplication *)application
{
    /*
     Use this method to release shared resources, save user data,
invalidate timers, and store enough application state information to
restore your application to its current state in case it is
```

```
terminated later.
     If your application supports background execution, this method
is called instead of applicationWillTerminate: when the user quits.
     */
       @try
       {
         wasPreviouslyInBackground = YES;
         [SUPApplication stopConnection:0];
       }
       @
       catch (NSException *ee)
       {
         // log an error or alert user via notification
       }
}



- (void)applicationWillEnterForeground:(UIApplication *)application
{
    /*
     Called as part of the transition from the background to the
inactive state; here you can undo many of the changes made on
entering the background.
     */
 if(wasPreviouslyInBackground)
   // Run these in the background since these are blocking calls and
   // this will be called from the UI thread.
   dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0);
   dispatch_async(queue, ^
     {
       @try
       {
         [SUPApplication startConnection:30];
       }
       @
       catch (NSException *ee)
       {
         // log an error or alert user via notification
       }
     });
}
```

Generating Objective-C Object API Code

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

Generating Objective-C Object API Code Using Sybase Unwired WorkSpace

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

Prerequisites

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

Task

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

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

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

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

Option	Description

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

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

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

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

4. Enter the information for these configuration options:

Option	Description
Language	Select **Objective-C**.
Platform	Select the platform (target device) for which the device client code is intended. • iOS
Unwired Server	Specify a default Unwired Server connection profile to which the generated code connects at runtime.

Option	Description
Server domain	Choose the domain to which the generated code connects. By default, if you specified an Unwired Server to which you previously connected successfully, the first domain in the list is chosen. Accept this domain, or enter a different one. **Note:** This field is enabled only when an Unwired Server is selected.
Page size	Not enabled for Objective-C.
Destination	Specify the destination of the generated device client files. Enter (or **Browse**) to a **Project path** or **File system path** (Mobile Application project) location, and select `Generated Code` as the destination. Select **Clean up destination before code generation** to clean up the destination folder before generating the device client files.
	If you select Java as the language, enter, or browse to the , which adds it to the project build path, and prevents errors after code generation.

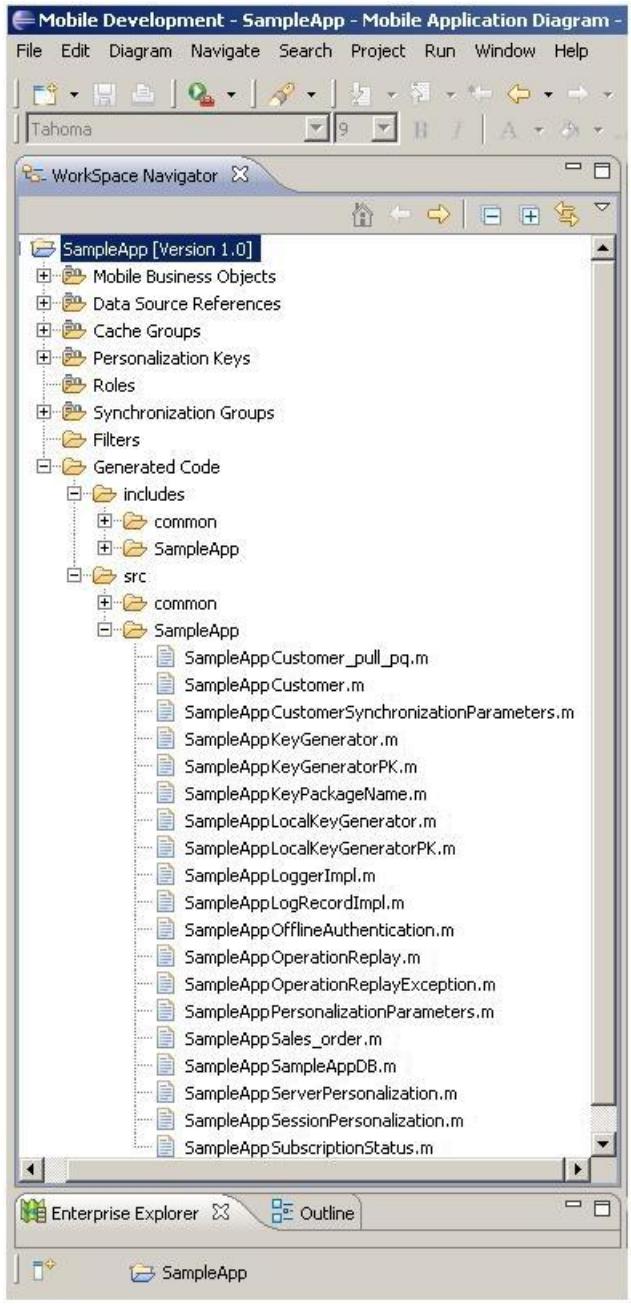
5. Select **Generate metadata classes** to generate metadata for the attributes and operations of each generated client object.

   The **Including object manager classes** option is only available if you select **Generate metadata classes**.

6. Click **Finish**.

   By default, the MBO source code and supporting documentation are generated in the project's `Generated Code` folder. The generated files are located in the `<MBO_project_name>` folder under the `includes` and `src` folders. The `includes` folder contains the header (*.h) files and the `src` folder contains the implementation (*.m) files.

   Because there is no namespace concept in Objective-C, all generated code is prefixed with packagename. For example, "SampleApp".



The frequently used Objective-C files in this project, described in code samples include:

Table 1. Source Code File Descriptions

Objective-C File	Description
MBO class (for example, `SampleAppCustomer.h`, `SampleAppCustomer.m`)	Include all the attributes, operations, object queries, and so on, defined in this MBO.
synchronization parameter class (for example, `SampleAppCustomerSynchronizationParameter.h`, `SampleAppCustomerSynchronizationParameter.m`)	Include any synchronization parameters defined in this MBO.
Key generator classes (for example, `SampleAppKeyGenerator.h`, `SampleAppKeyGenerator.m`)	Include generation of surrogate keys used to identify and track MBO instances and data.
Personalization parameter classes (for example, `SampleAppPersonalizationParameters.h`, `SampleAppPersonalizationParameters.m`)	Include any defined personalization keys.

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

7. Examine the generated code location and contents.
8. Validate the generated code.

Generating Object API Code Using the Code Generation Utility

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

Prerequisites

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

Task

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

   ```
   codegen.bat -oc -client -sqlite -mbs deployment_unit.xml [-output <output_dir>] [-doc]
   ```

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

   Ignore these warnings:

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

Generated Code Location and Contents

If you generated code in Unwired WorkSpace, generated object API code is stored by default in the "Destination" location you specified during code generation. If you generated code with the Code Generation Utility, generated object API code is stored in the `<UnwiredPlatform_InstallDir>\UnwiredPlatform\MobileSDK \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 Unwired WorkSpace, or specified in the Code Generation Utility. The contents include generated class (.h, .m) files that contain:

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

Validating Generated Code

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

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

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

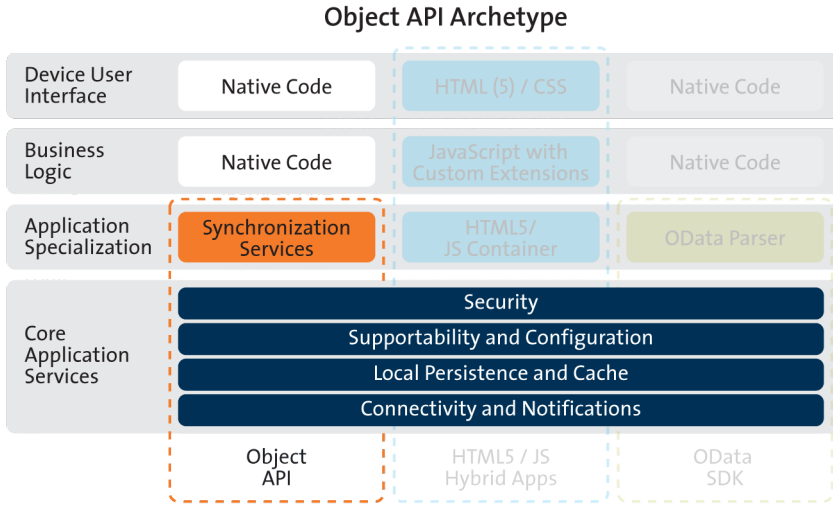
1. Select **Window > Preferences**.
2. Expand **Sybase, Inc > Mobile Development**.
3. Select **Mobile Business Object**.
4. Add or modify the **Naming Prefix** settings as needed.

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

Development Task Flow for DOE-based Object API Applications

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

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



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

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

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

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

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

Installing the iOS Development Environment

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

Downloading the Xcode IDE

Download and install Xcode.

1. Download Xcode from the Apple Web site: *http://developer.apple.com/xcode/*.
2. Close the iTunes program before beginning the installation.
3. Complete the Xcode installation following the instructions in the installer.

Installing X.509 Certificates on iOS Clients

Install generated X.509 certificates and test them in your iOS clients. A certificate provides an additional level of secure access to an application, and may be required by an organization's security policy.

Creating a Project

Build a device application project.

Generating HeaderDoc from Generated Code

Once you have generated Objective-C code for your mobile business objects, you can generate HeaderDoc (HTML reference information) on the Mac from the generated code. HeaderDoc provides reference information for the MBOs you have designed. The HeaderDoc will help you to programmatically bind your device application to the generated code.

1. Navigate to the directory containing the generated code that was copied over from the Eclipse environment.
2. Run:

```
>headerdoc2html –o GeneratedDocDir GeneratedCodeDir
>gatherheaderdoc GeneratedDocDir
```

You can open the file `OutputDir/masterTOC.html` in a Web browser to see the interlinked sets of documentation.

Note: You can review complete details on HeaderDoc in the *HeaderDoc User Guide*, available from the Mac OS X Reference Library at *http://developer.apple.com/mac/library/navigation/index.html*.

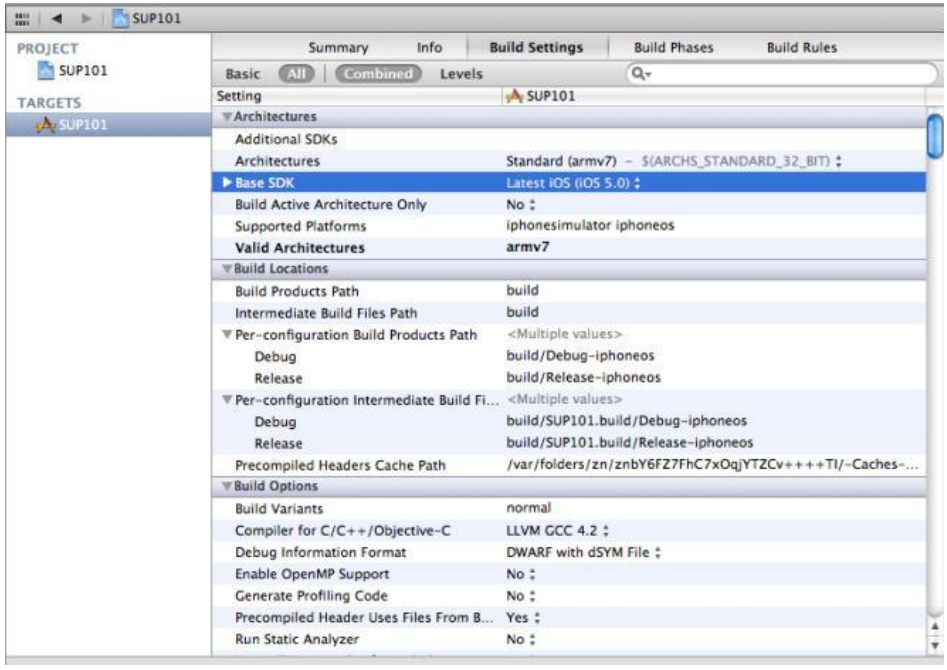
Importing Libraries and Code

Import the generated MBO code and associated libraries into the iOS development environment.

Note: For more information on Xcode, refer to the Apple Developer Connection: *http://developer.apple.com/tools/Xcode/*.

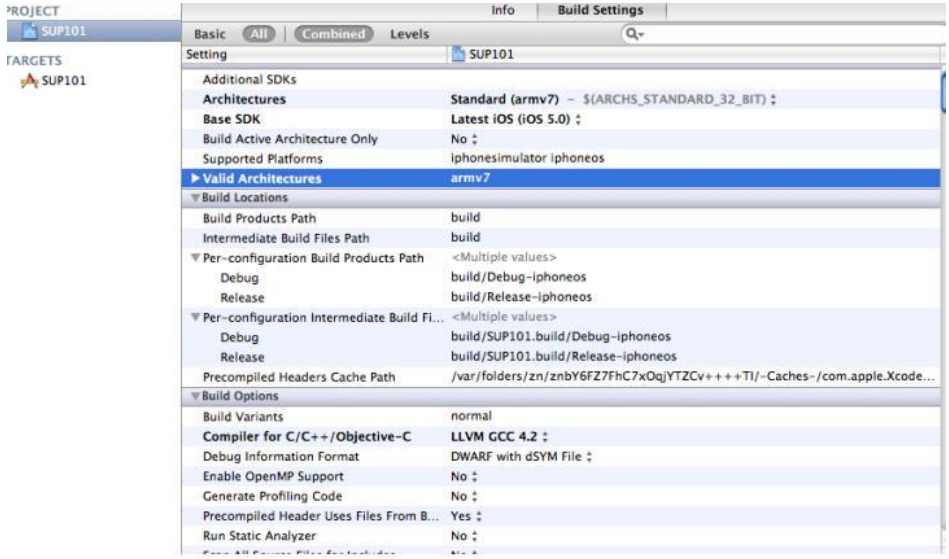
1. Start Xcode 4.2 and select **Create a new Xcode project**.
2. Select **iOS Application** and **Window-based Application** as the project template, and then click **Next**.
3. Enter `<ProjectName>` as the **Product Name**, `MyCorp` as the **Company Identifier**, select **Universal** as the **Device Family** product, and then click **Next**.

   Note: If you will deploy more than one Xcode project with the same application name, the applications will overwrite each other on the device. Ensure that projects do not share the same name even though they have different application IDs.
4. Select the **Architectures** tab, and set Base SDK for All Configurations to `iOS 5.0`.



5. Select the **Deployment** tab and set the iOS Deployment Target to `iOS 5.0` or `iOS 4.3`, as appropriate for the device version where you will deploy. Earlier SDKs and deployment targets are not supported.

6. Select the Valid architecture as `Standard (armv7)` and the Targeted device family as `iPhone/iPad`. This ensures that the build of the application can run on either iPhone or iPad.



7. Select a location to save the project and click **Create** to open it.

   Xcode creates a folder,`<ProjectName>`, to contain the project file, `<ProjectName>.xcodeproj` and another `<ProjectName>` folder, which contains a number of automatically generated files.

   Copy the files from your Windows machine in to the <ProjectName> folder that Xcode created to contain the generated source code.
8. Connect to the Microsoft Windows machine where Sybase Unwired Platform is installed:
   a) From the Apple Finder menu, select **Go** > **Connect to Server**.
   b) Enter the name or IP address of the machine, for example, `smb://<machine DNS name>` or `smb://<IP Address>`.
      You see the shared directory.
9. Navigate to the `\UnwiredPlatform\MobileSDK\ObjectAPI\DOE\iOS` directory in the Unwired Platform installation directory, and copy the `includes` and `libs` folders to the `<ProjectName>/<ProjectName>` directory on your Mac.
10. Navigate to the mobile application project (for example, `C:\Documents and Settings\administrator\workspace\<ProjectName>`), and copy the `Generated Code` folder to the `<ProjectName>/<ProjectName>` directory on your Mac.
11. In the Xcode Project Navigator, right-click the `<ProjectName>` folder under the project, select **Add Files to "<ProjectName>"**, select the `Generated Code` folder, unselect **Copy items into destination group's folder (if needed)**, and click **Add**.

The `Generated Code` folder is added to the project in the Project Navigator.

12. Right-click the `<ProjectName>` folder under the project, select **Add Files to "<ProjectName>"**, navigate to the `<ProjectName/ProjectName>/Libraries/Debug-iphonesimulator` directory, select the `libclientrt.a`, `libSUPObj.a`, `libMO.a`, libsupcore.a and `libAfariaSLL.a` libraries, unselect **Copy items into destination group's folder (if needed)**, and click **Add**.

    The libraries are added to the project in the Project Navigator.

    Note: The library version corresponds to the configuration you are building. For example, if you are building for a debug version of the simulator, navigate to `libs/Debug-iphonesimulator/` to add the libraries.

13. Right-click the project root, select **New Group**, and then rename it to `Resources`.
14. Right-click the `Resources` folder, select **Add Files to "<ProjectName>"**, navigate to the `includes` directory, select the `Settings.bundle` file, unselect **Copy items into destination group's folder (if needed)**, and click **Add**.

    The bundle `Settings.bundle` is added to the project in the Project Navigator.

    This bundle adds resources that lets iOS device client users input information such as server name, server port, user name and activation code in the Settings application.

15. Click the project root and then, in the middle pane, click the <**ProjectName**> project.
    a) In the right pane click the **Build Settings** tab, then scroll down to the **Search Paths** section.
    b) Enter the location of your `includes` folder (`"$SRCROOT/<ProjectName>/includes/**"`) in the **Header Search Paths** field.

       `$SRCROOT` is a macro that expands to the directory where the Xcode project file resides.
    c) Set Automatic Reference Counting (SRC) to NO.
    d) Set the iOS Deployment Target to 5.0 or 4.3.
16. Add the following frameworks from the SDK to your project by clicking on the active target, and selecting **Build Phase** > **Link Binary With Libraries**. Click on the + button and select the following binaries from the list:
    - AddressBook.framework
    - CoreFoundation.framework
    - QuartzCore.framework
    - Security.framework
    - CFNetwork.framework
    - SystemConfiguration.framework
    - MobileCoreServices.framework
    - libicucore.A.dylib
    - libstdc++.dylib

- libz.1.2.5.dylib

17. Select **Product > Clean** and then **Product > Build** to test the initial set up of the project. If you have correctly followed this procedure, then you should receive a **Build Succeeded** message.
18. Write your application code to reference the generated MBO code. See the *Developer Guide for iOS* for information about referencing the iOS Client Object API.

Importing Libraries and Code for Applications Enabled with ARC

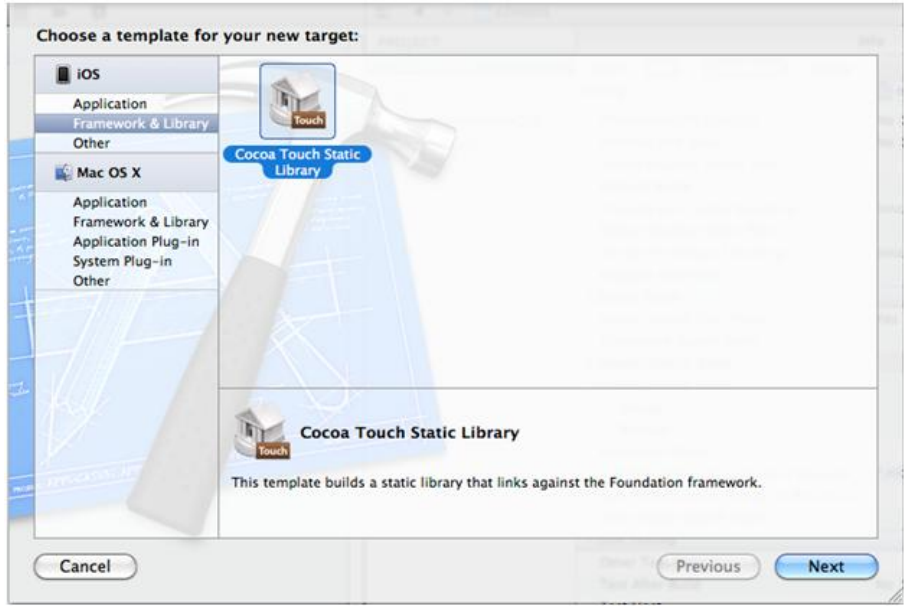
Import the generated MBO code and associated libraries into the iOS development environment, to support applications enabled with automatic reference counting (ARC).

Prerequisites

To use these steps, request a patch from CS&S.

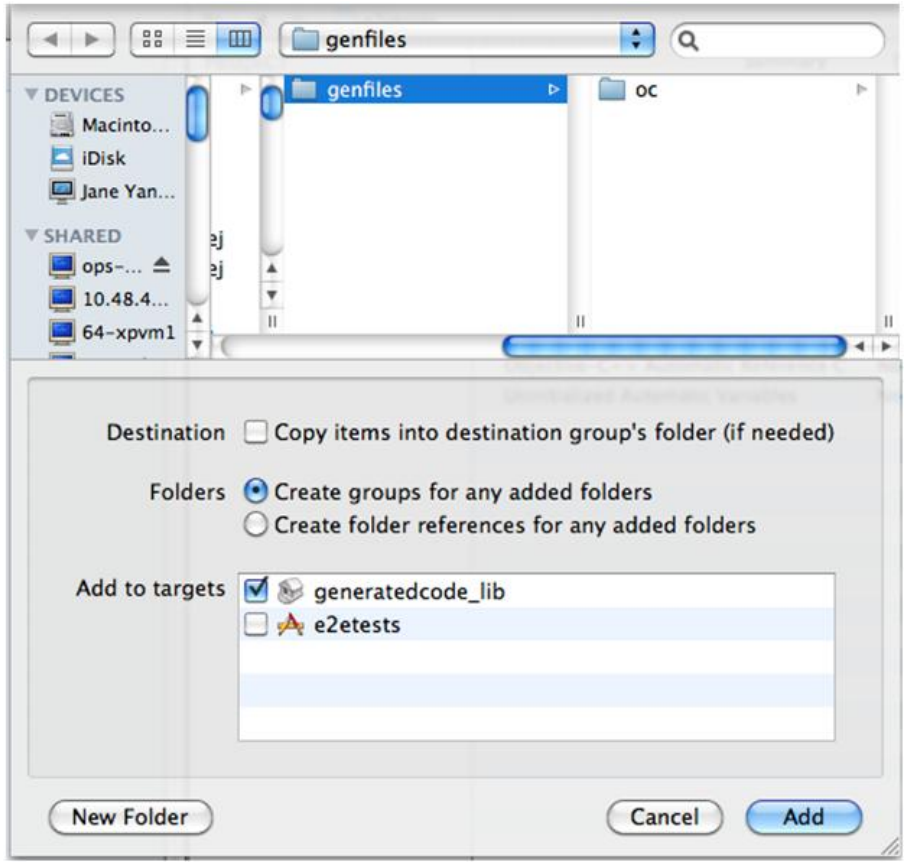
Task

1. Create a non-ARC static library target for the generated code.
   a) Select the application project file in Xcode, and click on **Add Target** at the bottom of the Project Settings screen. When prompted, select the "Cocoa Touch Static Library" template from the Framework & Library section and click **Next**.
   b) Enter the project name with the name you want for your library, for example, "generatedcode_lib". Make sure the "Use Automatic Reference Counting" option is not selected. Click on **Finish**. You have created a second target in your project.



c) Delete the sample class files the wizard created (`generatedcode_lib.h`, and `generatedcode_lib.m`).

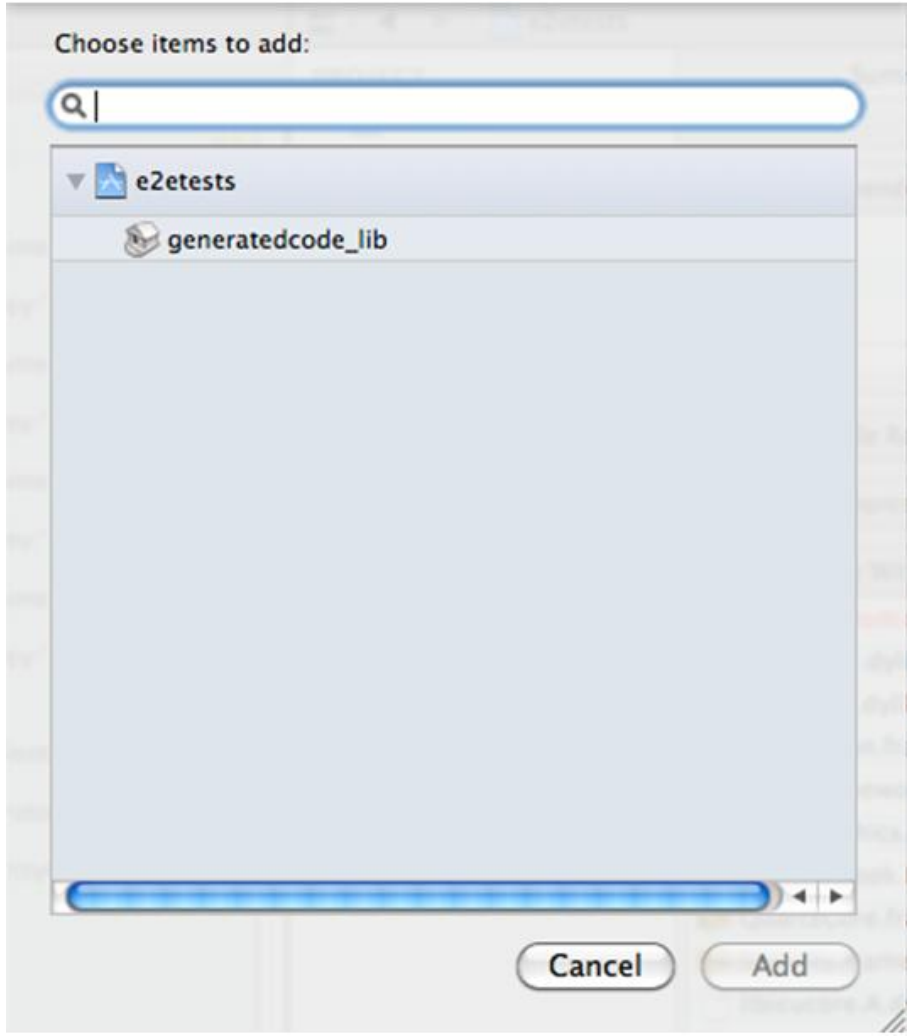
2. Make sure the static library is not using ARC by selecting the `generatedcode_lib` target, going to "Build Settings," and verifying "Automatic Reference Counting" is set to "NO".
3. Add generated code into the static library target.
   a) Right click on the `generatedcode_lib` folder from the Group & File view, and select **Add Files to ...**.
   b) Select your generated code location, and select the option "Add to targets" to "generatedcode_lib". Do not select *<your main target>*.
   c) Click **Add**.



4. Modify the build settings of the static library target.
   a) Select the generatedcode_lib target, and go to "Build Settings", and to "Header Search Paths".

b) Add the location of the SUP client stack `includes` folder. Make sure the "Recursive" checkbox is checked.

5. Link the main application target with the new static library.
   a) Select your main application target, then click on “Build Phase” and expand the “Link Binary With Libraries” section.
   b) Click on the plus (+) button and select the new static library from the list.
6. Add the static library as a dependency.
   a) Select your main application target, then click on “Build Phase” and expand the “Target Dependencies” section.
   b) Click on the plus (+) button and select the new static library from the list.



7. Make sure that ARC is enabled for your main application target.
   a) Select the main target, and go to “Build Settings”.
   b) Verify that Automatic Reference Counting” is set to “YES”.
8. Add your ARC enabled code into the main application target.
9. Import the Sybase Unwired Platform client stack libraries to the main target. Perform the steps in *Developer Guide: iOS Object API Applications* > *Development Task Flow for DOE-based Object API Applications* > *Creating a Project* > *Importing Libraries and Code*, to import and add only the libraries to the main target. Do not add generated code to the main target, because you have created the secondary static library target with the generated code.
10. Build your ARC-enabled main application target with the [[unresolved text-ref: product-short-name]] client stack and generated code.

    Ignore semantic issue warnings during compilation. For example:

```
"Semantic Issue
Type of property ‘databaseName’ does not match type of accessor
‘setDatabaseName:’"
```

Managing the Background State

To allow your application to continue to safely run when it goes into the background, you must implement code in its `AppDelegate` class to ensure that the `SUPApplication` instance's connection to the server shuts down gracefully when going into the background, and starts up when the application becomes active again.

This is important because in iOS, when an application goes into the background, it can have its network sockets invalidated, or the application may be shut down at any time. For correct behavior of the `SUPApplication` connection, the connection needs to be stopped when in background, and only started again when the application goes back to the foreground.

You must implement two `appDelegate` methods: `applicationDidEnterBackground` and `applicationWillEnterForeground`.

Note: The `applicationWillEnterForeground` method is also called when the application first starts up, where most applications would have code already to register the application and start the `SUPApplication` connection. This example code uses a boolean `wasPreviouslyInBackground` so that the `applicationWillEnterForeground` method can detect whether it is called on coming out of the background or is called on a first startup.

```
BOOL wasPreviouslyInBackground = NO;

- (void)applicationDidEnterBackground:(UIApplication *)application
{
    /*
     Use this method to release shared resources, save user data,
invalidate timers, and store enough application state information to
```

```
restore your application to its current state in case it is
terminated later.
     If your application supports background execution, this method
is called instead of applicationWillTerminate: when the user quits.
     */
       @try
       {
         wasPreviouslyInBackground = YES;
         [SUPApplication stopConnection:0];
       }
       @
       catch (NSException *ee)
       {
         // log an error or alert user via notification
       }
}



- (void)applicationWillEnterForeground:(UIApplication *)application
{
    /*
     Called as part of the transition from the background to the
inactive state; here you can undo many of the changes made on
entering the background.
     */
 if(wasPreviouslyInBackground)
   // Run these in the background since these are blocking calls and
   // this will be called from the UI thread.
   dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0);
   dispatch_async(queue, ^
     {
       @try
       {
         [SUPApplication startConnection:30];
       }
       @
       catch (NSException *ee)
       {
         // log an error or alert user via notification
       }
     });
}
```

Generating Objective-C Object API Code

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

Prerequisites

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

Task

1. Make sure that your <*ESDMA_dir*>\META-INF directory contains these three files:
   - afx-esdma.xml
   - ds-doe.xml
   - sup-db.xml
2. From <*UnwiredPlatform_InstallDir*>\UnwiredPlatform\MobileSDK\ObjectAPI\Utils\bin, run the codegen.bat utility, specifying the following parameters:

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

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

   Ignore these warnings:

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

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 (.h, .m) files that contain:

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

Customizing the Application Using the Object API

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

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

- Avoid making calls on the "main" thread on the device as this provides a poor response. Instead, use loading screens and activity spinners while doing the work in a background thread or operation queue. Do this while submitting and saving operations, and doing imports that update the tables displayed.
- Use an operation queue if you are trying to process imports and show them as they come in a `UITableViewController`. The operation callback will overwhelm the UI if you do one at a time. Instead, use an operation queue and process in groups.
- When testing for memory leaks, ignore the one-time startup leaks reported for the Messaging Server service.

Initializing an Application

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

Initially Starting an Application

Starting an application the first time.

Setting Up Application Properties

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

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

```
// Initialize Application settings
SUPApplication* app = [SUPApplication getInstance];

// The identifier has to match the application ID deployed to the SUP
server
app.applicationIdentifier = @"SUP101";

// ConnectionProperties has the infomation needed to register
// and connect to SUP server
SUPConnectionProperties* props = app.connectionProperties;
props.serverName = @"supserver.mycompany.com";
props.portNumber = 5001;
```

```
props.activationCode = @"activationcode";

// Other connection properties need to be set when connecting through
relay server

// provide user credentials
SUPLoginCredentials* login = [SUPLoginCredentials getInstance];
login.username = @"supAdmin";
login.password = @"supPwd";
props.loginCredentials = login;

// Initialize generated package database class with this Application
instance
[SUP101SUP101DB setApplication:app];
```

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 `SUPApplication` class. You do not need to use the `registerApplication` method for subsequent application startups.To start the connection to complete the registration process, use the `Application.startConnection` method.

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

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

```
SUPApplication* app = [SUPApplication getInstance];
@try {
  [app setApplicationIdentifier: @"appname"]; ( same as in SCC )
  [app setApplicationCallback:self];
  SUPConnectionProperties* props = app.connectionProperties;
  [props setServerName:@"servername"];
  [props setPortNumber:portnumber];
  [props setUrlSuffix:@""];
  [props setFarmId:@"1"];  ( same as in SCC )
  SUPLoginCredentials* login = [SUPLoginCredentials getInstance];
  login.username = @"username"; ( same as in SCC )
  login.password = nil;
  props.loginCredentials = login;
  props.activationCode = @"activationcode"; ( same as in SCC )
}
@catch (SUPPersistenceException * pe) {
  NSLog(@"%@: %@", [pe name],[pe message]);
}

// start database background synchronization here
@try {
  [app registerApplication:0]
}
@catch (SUPApplicationTimeoutException * pe) {
```

```
  NSLog(@"%@: %@", [pe name],[pe message]);
}
```

Setting Up the Connection Profile

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

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

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

Use the `SUPConnectionProfile` class to set up the locally generated database:

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

```
SUPConnectionProfile* cp = [SUP101SUP101DB getConnectionProfile];
[cp setEncryptionKey:@"Your key"];
[cp save];
```

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

Setting Up Connectivity

Store connection information to the Sybase Unwired Server data synchronization channel. Use Sybase Messaging settings (the iMO client) to connect to the messaging server.

Setting Up the Synchronization Profile

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

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

Typically, the application uses the settings as sent from the Unwired Server to connect to the Unwired Server for synchronization so that the administrator can set those at the application deployment time based on their deployment topology (for example, using relay server, using

e2ee security, or a certificate used for the intermediary, such as a Relay Server Web server). See the *Applications* and *Application Connection Templates* topics in *System Administration*.

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

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

   ```
   SUPConnectionProfile* cp = [SUP101_SUP101DB
   getSynchronizationProfile];
   ```

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

   ```
   SUPConnectionProfile* cp = [SUP101_SUP101DB
   getSynchronizationProfile];
   [cp setPortNumber:@"default"];
   [cp setNetworkStreamParams = "compression=zlib"];
   ```

Creating and Deleting a Device's Local Database

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

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

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

   ```
   if ([SUP101SUP101DB databaseExists])
   [SUP101SUP101DB createDatabase];
   ```

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

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

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

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

   ```
   [SUP101SUP101DB deleteDatabase];
   ```

Logging In

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

1. Normally, the user is authenticated through the `registerApplication` and `startConnection` methods in the `Application` class. Once this is done there is no need to authenticate again. However, the user can authenticate directly with the server at

any time during the application's execution by calling the generated database instance's or `beginOnlineLogin` method.

2. Authenticate using the last successful credentials on the device by calling the generated database instance's `offlineLogin` method.
3. Check the current login status by calling the generated database instance's `offlineLoginStatus` method, which returns one of the static values contained in the `com.sybase.persistence.SUPLoginStatus` class: `SUPLoginFailure`, `SUPLoginPending`, or `SUPLoginSuccess`.

Turn Off API Logger

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

```
[MBOLogger setLogLevel:LOG_OFF];
```

Setting Up Callbacks

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

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

- The `SUPApplicationCallback` class is used for monitoring changes to application settings, messaging connection status, and application registration status.
- The `SUPCallbackHandler` interface is used to monitor notifications and changes related to the database. Register callback handlers at the package level use the `registerCallbackHandler` method in the generated database class. To register for a particular MBO, use the `registerCallbackHandler` method in the generated MBO class.

Setting Up Callback Handlers

Use the callback handlers for event notifications.

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

1. The generated database class contains a method called `registerCallbackHandler`. Use this method to install your implementation of `SUPCallbackHandler`.
   For example:
   ```
   DBCallbackHandler* handler = [DBCallbackHandler newHandler];
   [SUP101SUP101DB registerCallbackHandler:handler];
   ```
2. Each generated MBO class also has the same method to register your implementation of the `SUPCallbackHandler` for that particular type. For example, if `Customer` is a generated MBO class, you can use the following code:

```
MyCustomerMBOCallbackHandler* handler =
[MyCustomerMBOCallbackHandler newHandler];
[Customer registerCallbackHandler:handler];
```

Synchronizing Applications

Synchronize package data between the device and the server.

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

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

Nonblocking Synchronization

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

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

1. If you have not yet synchronized with Unwired Server, perform a synchronization.
2. Set the synchronization parameters if there are any.
3. Make a blocking synchronize call to Unwired Server to pull in all MBO data:
   ```
   [SUP101SUP101DB beginSynchronize];
   ```
4. List all customer MBO instances from the local database using an object query, such as `FindAll`, which is a predefined object query.
   ```
   SUPObjectList *objlist = [ClientObj findAll];
   ```
5. Find and update a particular MBO instance, and save it to the local database.
   ```
   SampleAppCustomer *customer = [ SampleAppCustomer find: 32]
   //Change some sttribute of the customer record
   customer.fname= @"New Name";
   [customer save];
   ```
6. Submit the pending changes. The changes are ready for upload, but have not yet been uploaded to the Unwired Server.
   ```
   [Customer submitPending];
   ```
7. Use non-blocking synchronize call to upload the pending changes to the Unwired Server. The previous replay results and new changes are downloaded to the client device in the download phase of the synchronization session.
   ```
   [SUP101SUP101DB beginSynchronize];
   ```

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:

   ```
   pp = [SUP101SUP101DB getPersonalizationParameters];
   ```

2. Assign values to the `PersonalizationParameters` object:

   ```
   pp.Pkcity = @"New York";
   ```

3. Save the `PersonalizationParameters` value to the local database:

   ```
   [pp save]
   ```

   Note: If you define a default value for a personalization key that value will not take effect, unless you call `pp.save()`.

4. Synchronize the `PersonalizationParameters` value to the Unwired Server:

   ```
   [SUP101SUP101DB synchronize];
   ```

Specifying Synchronization Parameters

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

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

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

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

   ```
   CustomerSynchronizationParameters *sp = [Customer
   getSynchronizationParameters];
   ```

2. Assign values to the synchronization parameter. For example, if the Customer MBO contains a parameter named `cityname`, assign the value to the `CustomerSynchronizationParameters` object's `cityname` field:

   ```
   sp.cityname = @"Kansas City";
   ```

3. Save your changes by calling the synchronization parameters object's `save` method:

```
[sp save];
```

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

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

```
[params delete];
<MBO>SynchronizationParameters *params = [<MBO>
getSynchronizationParameters]; //must re-get the sync parameter
instance
params.Param1 = @value1; //set new sync parameter value
params.Param2 = @value2; //set new sync parameter value
[params save];
```

Subsequently Starting an Application

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

Starting an application on subsequent occasions:

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

```
[application startConnection];
```

Accessing MBO Data

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

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 `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.

Dynamic Queries

Build queries based on user input.

Use the `SUPQuery` class to retrieve a list of MBOs.

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

   ```
   SUPQuery *myquery = [SUPQuery getInstance];
   myquery.testCriteria = [SUPAttributeTest
   match:@"fname" :@"Erin"];
   ```

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

   ```
   SUPObjectList* customers = [SampleAppCustomer
   findWithQuery:myquery]
   ```

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

   ```
   SUPQuery *query = [SUPQuery getInstance];
   [query select:@"c.fname,c.lname,s.order_date,s.id"];
   [query from:@"Customer":@"c"];
   [query join:@"SalesOrder":@"s":@"s.cust_id":@"c.id"];
   query.testCriteria = [SUPAttributeTest
   match:@"c.lname":@"Smith"];
   SUPQueryResultSet* resultSet = [SUP101_SUP101DB
   executeQuery:query];
   if(resultSet == nil)
   {
     MBOLog(@"executeQuery Failed !!");
     return;
   }
   for(SUPDataValueList* result in resultSet)
   {
     MBOLog(@"Firstname,lastname,order date,region = %@ %@ %@ %@",
     [SUPDataValue getNullableString:[result item:0]],
     [SUPDataValue getNullableString:[result item:1]],
     [[SUPDataValue getNullableDate:[result item:2]] description],
     [SUPDataValue getNullableString:[result item:3]]);
   }
   ```

MBOs with Complex Types

Mobile business objects are mapped to classes containing data and methods that support synchronization and data manipulation. You can develop complex types that support interactions with backend data sources such as SAP® and Web services. When you define an

MBO with complex types, Sybase Unwired Platform generates one class for each complex type.

Using a complex type to create an MBO instance.

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

   ```
   AuthenticationInfo* authinfo;
   authinfo = [AuthenticationInfo getInstance];
   authinfo.userName=@"Francie";
   ```

2. Instantiate the MBO object:

   ```
   SimpleCaseList *cr = [[SimpleCaseList alloc] init];
   cr.company = @"Calbro Services";
   ```

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

   ```
   [cr create:authinfo];
   [cr submitPending];
   ```

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:

   ```
   SampleAppCustomer *customer = [SampleAppCustomer find:101];
   SUPObjectList *orders = customer.salesOrders;
   ```

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

   ```
   SUPQuery *query = [SUPQuery getInstance];
   [query select:@"c.fname,c.lname,s.order_date,s.region"];
   [query from:@"Customer":@"c"];
   [query join:@"SalesOrder":@"s":@"s.cust_id":@"c.id"];
   query.testCriteria = [SUPAttributeTest
   match:@"c.lname":@"Devlin"];
   SUPQueryResultSet* resultSet = [SUP101SUP101DB
   executeQuery: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:

```
SampleAppCustomer *customer = [ SampleAppCustomer find: 32]
customer.city = @"Dublin";
SampleAppSales_order* order = [SampleAppSales_order find: 1220];
order.region = @"SA"; //update any field
[order update]; //call update on the child record
[order refresh];
[order.customer submitPending];
```

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.

Creating, Updating, and Deleting MBOs

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

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

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

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

   ```
   SampleAppCustomer *newcustomer = [[SampleAppCustomer alloc]
   init];
   newcustomer.fname = @"John";
   ... //Set the required fields for the customer
   [newcustomer create];
   [newcustomer submitPending];
   ```

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:

```
SampleAppCustomer *customer = [ SampleAppCustomer find: 32]
//find by the unique id
customer.city = @"Dublin"; //update any field to a new value
[customer update];
[customer submitPending];
```

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:

```
SampleAppCustomer *customer = [ SampleAppCustomer find: 32]
[customer delete];
[customer submitPending];
```

Other Operations

Use operations other than create, update, or delete.

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

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

```
SampleAppCustomerOtherOperation *other =
[[SampleAppCustomerOtherOperation alloc] init];
other.P1 = @"somevalue";
other.P2 = 2;
other.P3 = [NSDate date];
[other save];
```

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

```

[other submitPending];
```

Using SubmitPending and SubmitPendingOperations

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

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

Database Classes

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

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

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

Generated MBOs

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

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

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

Shutting Down the Application

Shut down an application and clean up connections.

Closing Connections

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

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

   ```
   [app stopConnection:<timeout_value>];
   ```
2. Close all connections to device database by calling the `closeConnection` method in the generated package database class. If one application has multiple packages, invoke the `closeConnection` API in all the packages.

Uninstalling the Application

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

Deleting the Database and Unregistering the Application

Delete the package database, and unregister the application.

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

   ```
   [SUP101DB deleteDatabase];
   ```

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

   ```
   @try {
     [app unregisterApplication:<time out value>]
   }
   @catch (SUPApplicationTimeoutException * pe) {
     NSLog(@"%@: %@", [pe name],[pe message]);
   }
   ```

Testing Applications

Test native applications on a device or simulator.

Testing an Application Using a Emulator

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

1. In Xcode, select **Product** > **Build** and then **Product** > **Run**.
   The project is built and the iPhone Simulator starts.
2. In the iPhone simulator, go to **Settings > <Application>** to enter the connection settings.
   - ServerNameSetting – the machine that hosts the server where the mobile application project is deployed.
   - ServerPortSetting – Unwired Server port number. The default is **5001**.
   - CompanyIDSetting – if connecting directly to Unwired Server company ID is not required. If connecting through a Relay Server, set the company ID to the Relay Server farm ID.
   - UserNameSetting – the user defined in the security configuration you selected at deployment time.
   - ActivationCodeSetting – the activation code for the user. This setting is not required for automatic registration; an activation code is only required if you are performing manual registration.
3. In the iPhone applications screen, open the application.
4. In Sybase Control Center verify that the application connection was created in **Applications** > **Application Connections**.
   When the application has successfully registered, the application connection displays a value of zero in the Pending Items column.
5. Test the functionality of the application. Use debug tools as necessary, setting breakpoints at appropriate places in the application.

Client-Side Debugging

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

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

- Unwired Server connection failed - use your device browser to check the connectivity of your device to the server.

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

To find out more information on the device client side:

- If you have implemented debugging in your generated or custom code (which Sybase recommends), turn on debugging and review the debugging information. See the API Reference information about using the `Logger` class to add logs to the client log record and synchronize them to the server (viewable in Sybase Control Center).
- Check the log record on the device. Use the **getLogRecords (SUPQQuery)** or **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` – Unwired Server administration codes.
    - Synchronization codes:
      - 200 – success.
      - 500 – failure.
  - `eisCode` – maps to HTTP error codes. If no mapping exists, defaults to error code 500 (an unexpected server failure).
  - `message` – the message content.
  - `component` – MBO name.
  - `entityKey` – MBO surrogate key, used to identify and track MBO instances and data.
  - `operation` – operation name.
  - `requestId` – operation replay request ID or messaging-based synchronization message request ID.
  - `timestamp` – message logged time, or operation execution time.

- If you have implemented `ApplicationCallback.`**`onConnectionStatusChanged`** for synchronization in the `CallbackHandler`, the connection status between Unwired Server and the device is reported on the device. See the `SUPCallbackHandler` 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:

```
SUPConnectionProfile *cp = [SUP101SUP101DB getConnectionProfile];

// To enable trace of client database operations (SQL statements,
etc.)
[cp enableTrace:YES];

// To enable trace of client database operations with values also
displayed
[cp enableTrace:YES withPayload:YES];

// To disable trace of client database operations
[cp enableTrace:NO];

// To enable trace of message headers sent to the server and
received from the server
// (this replaces the MBODebugLogger and MBODebugSettings used in
earlier versions of SUP)
[cp.syncProfile enableTrace:YES];

// To enable trace of both message headers and content, including
credentials
[cp.syncProfile enableTrace:YES withPayload:YES];

// To disable messaging trace
[cp.syncProfile enableTrace:NO];
```

Server-Side Debugging

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

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

- The domain or package does not exist. If you create a new domain, with a default status of disabled, it is unavailable until enabled.
- Authentication failed for the application user credentials.
- The operation role check failed for the synchronizing user.
- Back-end authentication failed.

- An operation failed on the remote, replication database back end, for example, a table or foreign key does not exist.
- An operation failed on the Web Service, REST, or SAP® back end.

To find out more information on the Unwired Server side:

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

  Note: Return to INFO mode as soon as possible, since DEBUG mode can affect system performance.
- Obtain DEBUG information for a specific device:
  - In the SCC administration console:
    1. Set the DEBUG level to a higher value for a specified device:
       a. In SCC, select **Application Connections**, then select **Properties...** > **Device Advanced**.
       b. Set the Debug Trace Level value.
    2. Set the TRACE file size to be greater than 50KB.
    3. View the trace file through SCC.
  - Check the `<server_install_folder>\UnwiredPlatform\Servers\MessagingServer\Data\ClientTrace` directory to see the mobile device client log files for information about a specific device.

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

Localizing Applications

In iOS, you use Interface Builder, which is part of Xcode, to define and layout controls in a view of the user interface. These descriptions are stored in Xcode Interface Builder (XIB) files. Once you have the English version of the layout defined you will need to create an XIB file for each language you want to support in your user interface.

Localizing Menus and Interfaces

Localize the menus and interfaces for an iOS application by selecting an XIB file to localize, and a language for localization.

1. Select the Xcode Interface Builder (XIB) file you want to localize in the Project Explorer.
2. Open the File Inspector by selecting **View** > **Utilities** > **File Inspector**. The File Inspector appears in a pane of the right of the Xcode window.
3. In the Localization section of the File Inspector pane, click the + button at the bottom of the section.
   This step makes the XIB file localizable by moving it into a folder named `en.lproj`.
4. Click the + button again.
   A menu appears with a list of languages.
5. Select the language you want to use in localizing the XIB file.

   The Localization section of the File Inspector displays the languages to which the file has been localized (in the example, French and English).

   The file's icon in the Project Explorer has a disclosure arrow next to it. Click the arrow to reveal the contents of the file. The Project Explorer displays one copy of the XIB file for each language you have chosen.
6. Double-click on each icon to open it in a new tab or new window.
7. Make the required changes to the interface elements in the language-specific XIB file, and then save the file.
8. Verify that the localized XIB files are added to the list of files copied into the application's bundle. If not:
   a) Click the project icon in the Project Explorer, and then click the Target icon.
   b) Select the Build Phases tab.
   c) Expand the Copy Bundle Resources section, and then click the + button.
   d) Select the additional XIB files from the `<language>.lproj` folders and click **Add**.

Localizing Embedded Strings

Localize embedded strings that are used in alert and dialog windows.

1. For each user interface string in your code, set the text property to a literal string using the `NSLocalizedString` macro.

   ```
   UserInterfaceLabel.text = NSLocalizedString(@"Display text",
   nil);
   ```

2. Generate the `.strings` files from all the `NSLocalizedString` references in your application. by using the `genstrings` command line program. See Apple documentation for command syntax and parameters.
   This command processes files in your directory hierarchy and creates `.strings` files for them in the `en.lproj` directory.
3. Provide your translator a copy of the `.strings` file. The translator should translate the right side of each of the `.strings` file entries.

Validating Localization Changes

Test that your changes appear in your application.

1. Launch the iOS simulator then launch Settings.app.
2. Select **General** > **International** > **Language**.
3. Select the language you want to test.
   The simulator restarts in the new language.
4. Launch your application and verify that it is localized.

Packaging Applications

Package applications according to your security or application distribution requirements.

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

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

Signing

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

Apple Push Notification Service Configuration

The Apple Push Notification Service (APNS) notifies users when information on a server is ready to be downloaded.

Apple Push Notification Service (APNS) allows users to receive notifications. APNS:

- Must be set up and configured by an administrator on the server.
- Must be enabled by the user on the device.
- Can be used with any device that supports APNS. Some older Apple devices may not support APNS.
- Cannot be used on a simulator.

Preparing an Application for Apple Push Notification Service

There are several development steps to perform before the administrator can configure the Apple Push Notification Service (APNS).

Note: Review complete details in the *iPhone OS Enterprise Deployment Guide* at *http://manuals.info.apple.com/en_US/Enterprise_Deployment_Guide.pdf*.

1. Sign up for the iOS Developer Program, which gives you access to the Developer Connection portal. Registering as an enterprise developer gets you the certificate you need to sign applications.
2. Configure your application to make use of Keychain as persistent storage for the database encryption key.
3. Create an App ID and ensure that it is configured to use Apple Push Notification Service (APNS).

Do not use wildcard characters in App IDs for iPhone applications that use APNS.

Verify that your `info.plist` file has the correct App ID and application name. Also, in Xcode, right-click **Targets** > **<your_app_target>** and select **Get Info** to verify the App ID and App name.

4. Create and download an enterprise APNS certificate that uses Keychain Access in the Mac OS. The information in the certificate request must use a different common name than the development certificate that may already exist. The reason for this naming requirement is that the enterprise certificate creates a private key, which must be distinct from the development key. Import the certificate as a login Keychain, not as a system Keychain. Validate that the certificate is associated with the key in the Keychain Access application. Get a copy of this certificate.
5. Create an enterprise provisioning profile and include the required device IDs with the enterprise certificate. The provisioning profile authorizes devices to use applications you have signed.
6. Create the Xcode project, ensuring the bundle identifier corresponds to the bundle identifier in the specified App ID.
7. To enable the APNS protocol, you must implement several methods in the application by adding the code below:

Note: The location of these methods in the code depends on the application; see the APNS documentation for the correct location.

```
//Enable APNS
[[UIApplication sharedApplication]
registerForRemoteNotificationTypes:
    (UIRemoteNotificationTypeBadge |
     UIRemoteNotificationTypeSound |
     UIRemoteNotificationTypeAlert)];

* Callback by the system where the token is provided to the client
application so that this
 can be passed on to the provider. In this case,
“deviceTokenForPush” and “setupForPush”
are APIs provided by SUP to enable APNS and pass the token to SUP
Server

- (void)application:(UIApplication *)app
didRegisterForRemoteNotificationsWithDeviceToken:
  (NSData *)devToken
{
    MBOLogInfo(@"In did register for Remote Notifications",
devToken);
    [SUPPushNotification setupForPush:app];
    [SUPPushNotification deviceTokenForPush:app
deviceToken:devToken];
}

* Callback by the system if registering for remote notification
failed.
```

```
- (void)application:(UIApplication *)app
didFailToRegisterForRemoteNotificationsWithError:
      (NSError *)err {
        MBOLogError(@"Error in registration. Error: %@", err);
       }

// You can alternately implement the pushRegistrationFailed API:

// +(void)pushRegistrationFailed:(UIApplication*)application
errorInfo: (NSError *)err


* Callback when notification is sent.


- (void)application:(UIApplication *)app
didReceiveRemoteNotification:(NSDictionary *)
   userInfo
{
    MBOLogInfo(@"In did receive  Remote Notifications", userInfo);
}

You can alternately implement the pushNotification API
+(void)pushNotification:(UIApplication*)application
notifyData:(NSDictionary *)userInfo
```

Provisioning an Application for Apple Push Notification Service

Use Apple Push Notification Service (APNS) to push notifications from Unwired Server to the iOS application. Notifications can include badges, sounds, or custom text alerts. Device users can customize which notifications to receive through Settings, or turn them off.

Each application that supports Apple Push Notifications must be listed in Sybase Control Center with its certificate and application name. You must perform this task for each application.

1. Confirm that the IT department has opened ports 2195 and 2196, by executing:

   ```
   telnet gateway.push.apple.com 2195
   telnet feedback.push.apple.com 2196
   ```

   If the ports are open, you can connect to the Apple push gateway and receive feedback from it.
2. Copy the enterprise certificate (`*.p12`) to the computer on which Sybase Control Center has been installed. Save the certificate in *`UnwiredPlatform_InstallDir`*`\Servers\MessagingServer\bin\`.
3. In Sybase Control Center, expand the **Servers** folder and click **Server Configuration** for the primary server in the cluster.
4. In the **Messaging** tab, select **Apple Push Configuration**, and:

a) Configure Application name with the same name used to configure the product name in Xcode. If the certificate does not automatically appear, browse to the directory.
b) Change the push gateway information to match that used in the production environment.
c) Restart Unwired Server.

5. Verify that the server environment is set up correctly:
   a) Open *UnwiredPlatform_InstallDir*`\Servers\UnwiredServer\logs\APNSProvider`.
   b) Open the log file that should now appear in this directory. The log file indicates whether the connection to the push gateway is successful or not.
6. Deploy the application and the enterprise distribution provisioning profile to your users' computers.
7. Instruct users to use iTunes to install the application and profile, and how to enable notifications. In particular, device users must:
   - Download the Sybase application from the App Store.
   - In the iPhone Settings app, slide the **Notifications** control to **On**.
8. Verify that the APNS-enabled iOS device is set up correctly:
   a) Click **Device Users**.
   b) Review the Device ID column. The application name should appear correctly at the end of the hexadecimal string.
   c) Select the Device ID and click **Properties**.
   d) Check that the APNS device token has been passed correctly from the application by verifying that a value is in the row. A device token appears only after the application runs.
9. Test the environment by initiating an action that results in a new message being sent to the client.

   If you have verified that both device and server can establish a connection to APNS gateway, the device will receive notifications and messages from the Unwired Server, including workflow messages, and any other messages that are meant to be delivered to that device. Allow a few minutes for the delivery or notification mechanism to take effect and monitor the pending items in the Device Users data to see that the value increases appropriately for the applications.
10. To troubleshoot APNS, use the *UnwiredPlatform_InstallDir*`\\Servers\UnwiredServer\log\APNSProvider` log file. You can increase the trace output by editing *SUP_Home*`\Servers\MessagingServer\Data\TraceConfig.xml` and configuring the tracing level for the APNSProvider module to debug for short periods.

Preparing Applications for Deployment to the Enterprise

After you have created your client application, you must sign your application with a certificate from Apple, and deploy it to your enterprise.

Note: Developers can review complete details in the *iPhone OS Enterprise Deployment Guide* at *http://manuals.info.apple.com/en_US/Enterprise_Deployment_Guide.pdf*.

1. Sign up for the iOS Developer Program, which gives you access to the Developer Connection portal. Registering as an enterprise developer gets you the certificate you need to sign applications.
2. Create a certificate request on your Mac through Keychain.
3. Log in to the Developer Connection portal.
4. Upload your certificate request.
5. Download the certificate to your Mac. Use this certificate to sign your application.
6. Create an AppID.

   Verify that your `info.plist` file has the correct AppID and application name. Also, in Xcode, right-click **Targets** > **<your_app_target>** and select **Get Info** to verify the AppID and App name.
7. Create an enterprise provisioning profile and include the required device IDs with the enterprise certificate. The provisioning profile authorizes devices to use applications you have signed.
8. Create an Xcode project ensuring the bundle identifier corresponds to the bundle identifier in the specified App ID. Ensure you are informed of the "Product Name" used in this project.

Client Object API Usage

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

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

Client Object API Reference

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

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

There is a subdirectory for `ObjectiveC`.

Application APIs

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

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

getInstance

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

Syntax

```
+ (SUPApplication*)getInstance;
```

Returns

`getInstance` returns a singleton `Application` object.

Examples

- **Get the Application Instance**

```
SUPApplication* app = [SUPApplication getInstance];
```

setApplicationIdentifier

Sets the identifier for the current application.

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

Syntax

```
+(void)setApplicationIdentifier:(NSString*)value;
```

Parameters

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

Examples

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

Note: The application identifier is case sensitive.

```
SUPApplication* app = [SUPApplication getInstance];
@try {
  [app setApplicationIdentifier: @"SUP101"]; ( same as in SCC )
  ...
}
@catch (SUPPersistenceException * pe) {
  NSLog(@"%@: %@", [pe name],[pe message]);
}
```

registrationStatus

Retrieves the current status of the mobile application registration.

Syntax

```
+(SUPInt)registrationStatus;
```

Returns

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

```
//The registration bas been successfully created.
#define SUPRegistrationStatus_REGISTERED 203

//The registration is currently being created.
```

```
#define SUPRegistrationStatus_REGISTERING 202

//The registration could not be created or deleted. Using
onRegistrationStatusChanged you can
//capture the associated errorCode and errorMessage. This is a
permanent condition that will
//not be automatically resolved,
//so registerApplication or unregisterApplication must be! called
again to retry.
#define SUPRegistrationStatus_REGISTRATION_ERROR 201

//The registration has been successfully deleted, or there was no
previous registration.
#define SUPRegistrationStatus_UNREGISTERED 205

//The registration is currently being deleted.
#define SUPRegistrationStatus_UNREGISTERING 204
```

registerApplication

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

If an application identifier has not already been set, a `SUPPersistanceException` is thrown. If connection properties are not available, a `SUPConnectionPropertyException` is thrown. If you use this method, do not call `startConnection`.

Syntax

```
- (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`.

```
SUPApplication* app = [SUPApplication getInstance];
[app setApplicationIdentifier:@"SUP101"];

MyApplicationCallbackHandler *ch = [MyApplicationCallbackHandler
getInstance];
```

```
[ch retain];
[app setApplicationCallback:ch];

SUPConnectionProperties* props = app.connectionProperties;
[props setServerName:@"supserver.mycompany.com"];
[props setPortNumber:5001];

SUPLoginCredentials* login = [SUPLoginCredentials getInstance];
login.username = @"supAdmin";
login.password = @"supPwd";
props.loginCredentials = login;

if ([app registrationStatus] != SUPRegistrationStatus_REGISTERED &&
[app registrationStatus] != SUPRegistrationStatus_REGISTERING )
{
[app registerApplication:120]; // 120 second timeout for
registration
}
```

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 an application identifier has not already been set, a `SUPPersistanceException` is thrown. If connection properties are not available, a `SUPConnectionPropertyException` is thrown. If the timeout is greater than 0 and the registration takes longer than the timeout, then a `SUPApplicationTimeoutException` is thrown, even though the process will continue in the background. If you use this method, do not call `startConnection`.

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

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

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

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

In such a case, the registration process has permanently failed and will not continue in the background. If a callback handler is registered and network connectivity is available for the

start of registration but becomes unavailable before the connection is established, the sequence of callbacks as a result of calling registerApplication is:

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

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

Syntax

```
- (void)registerApplication :(SUPInt)timeout;
```

Parameters

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

Examples

- **Register an Application –** Registers the application with a one minute waiting period.
  ```
  [app registerApplication:60];
  ```

Usage

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

```
SUPApplication* app = [SUPApplication getInstance];
[app setApplicationIdentifier:@"SUP101"];

MyApplicationCallbackHandler *ch = [MyApplicationCallbackHandler
getInstance];
[ch retain];
[app setApplicationCallback:ch];

SUPConnectionProperties* props = app.connectionProperties;
[props setServerName:@"supserver.mycompany.com"];
[props setPortNumber:5001];

SUPLoginCredentials* login = [SUPLoginCredentials getInstance];
login.username = @"supAdmin";
login.password = @"supPwd";
props.loginCredentials = login;

if ([app registrationStatus] != SUPRegistrationStatus_REGISTERED &&
```

```
[app registrationStatus] != SUPRegistrationStatus_REGISTERING )
{
[app registerApplication:120]; // 120 second timeout for
registration
}
```

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

```
+ (void)setApplicationCallback:(SUPApplicationCallback*)value;
```

Parameters

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

Examples

- **Set the Application Callback**

```
SUPApplication* app = [SUPApplication getInstance];
@try {
  [app setApplicationIdentifier: @"appname"]; ( same as in SCC )
  [app setApplicationCallback:self];
  ...
}
@catch (SUPPersistenceException * pe) {
  NSLog(@"%@: %@", [pe name],[pe message]);
}
```

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

```
+(void)startConnection:(int32_t)timeout;
```

Parameters

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

Returns

None.

Examples

- **Start the Application**

  ```
  startConnection(int timeout)
  ```

connectionStatus

Return current status of the mobile application connection.

Syntax

```
+ (int32_t)connectionStatus;
```

Returns

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

```
//The connection been successfully started.
#define SUPConnectionStatus_CONNECTED 103

//The connection is currently being started.
#define SUPConnectionStatus_CONNECTING 102
```

```
//The connection could not be started, or was previously started and
subsequently an error occurred. Using
//onConnectionStatusChanged you can capture the associated errorCode
and errorMessage. This is a temporary condition that
//can be automatically! resolved, if network connectivity can be
established or reestablished.
#define SUPConnectionStatus_CONNECTION_ERROR 101

//The connection been successfully stopped, or there was no previous
connection.
#define SUPConnectionStatus_DISCONNECTED 105

//The connection is currently being stopped.
#define SUPConnectionStatus_DISCONNECTING 104
```

Examples

- **Get the Application Connection Status**

  ```
  [SUPApplication connectionStatus];
  ```

stopConnection: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

```
+ (void)stopConnection:(int32_t)timeout
```

Parameters

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

Returns

None.

Examples

- **Stop the Application**

```
[SUPApplication stopConnection:<timeout>];
```

unregisterApplication

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

Syntax

```
+ (void)unregisterApplication;
```

Parameters

None.

Examples

- **Unregister an Application –** Unregisters the application.

```
[app unregisterApplication];
```

unregisterApplication: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

```
+ (void)unregisterApplication:(int32_t)timeout;
```

Parameters

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

Examples

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

  ```
  [app unregisterApplication:60];
  ```

Connection APIs

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

SUPConnectionProfile

The `SUPConnectionProfile` 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".

```
SUPConnectionProfile* cp = [SUP101SUP101DB getConnectionProfile];
[cp setEncryptionKey:@"Your key of more than 16 characters"];
// Immediately after the call to setEncryptionKey, call [cp
closeConnection]; to ensure that old connections with the wrong key
are no longer being used.
[cp closeConnection];
```

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.

Managing Device Database Connections

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

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

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

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

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

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

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

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

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

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

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

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

```
SUPConnectionProfile *cp = [SUP101SUP101DB 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.

```
cp.maxDbConnections = 6;
```

Set Database File Property

You can use `setProperty` to specify the database file name created in the `Documents` directory of the application, on the device or simulator.

```
SUPConnectionProfile *cp = [SUP101SUP101DB getConnectionProfile];
[cp setString:@"databaseFile":@"newDatabaseFileName"];
```

Usage

- Be sure to call this API before the database is created..
- The database is SQLite; use a database file name like `mydb.db`.
- 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.

Synchronization Profile

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

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

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

```
SUPConnectionProfile* sp = [SUP101SUP101DB
getSynchronizationProfile];
[sp setDomainName:@"default"];
```

Connect the Data Synchronization Channel Through a Relay Server

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

If a Relay Server is used, the 'companyID' in the SUPApplication property must correspond to the farm ID of the Relay Server.

```
SUPConnectionPropeties props = app.connectionProperties;
[props seFarmId:@"relayServer1"];
```

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

Authentication APIs

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

Logging In

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

Note: For DOE-based applications, do not use `beginOnlineLogin`. Instead, just set the user name and password in the synchronization profile and immediately call `subscribe`.

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

- **`beginOnlineLogin:(NSString *)user password:(NSString *)pass`** – sends a message to the Unwired Server with the user name and password. The Unwired Server responds with a message to the client with the login success or failure. This method checks the `SUPApplication connectionStatus` and immediately fails if the status is not `SUPConnectionStatus_CONNECTED`. Make sure the connection is active before calling `beginOnlineLogin`, or implement the `onLoginFailure` callback handler to catch cases where it may fail.

  ```
  [SUP101SUP101DB beginOnlineLogin:@"supUser" password:@"s3pUser"];
  ```

Importing an X.509 Certificate to an iOS Client from the Unwired Server

Log in to Unwired Server and authenticate a client using a generated X.509 certificate instead of a user name and password combination.

1. Copy the X.509 certificate used for authentication into a directory on the same host as Unwired Server. For example, `c:\certs`.
2. Create a registry string value on Unwired Server at `HKLM\Software\Sybase\Sybase Messaging Server\CertificateLocation` and populate it with the path. For example, `c:\certs`.
3. Name the X.509 certificate file as `domain_user.p12`, where *domain* is the Unwired Server domain and *user* is the certificate user. The user must have read permission for `.p12` file.
4. The system administrator must ensure the specified domain\user has "logon as batch job" permission on the Windows machine on which Unwired Server runs:
   a) Double-click **Control Panel** > **Administrative Tools** > **Local Security Policies**.
   b) Expand **Local Policies** and select **User Rights Assignment**.
   c) Right-click **Log on as a batch job** and select **Properties**.
   d) Select **Add User or Group** and add the domain\user.
5. The account under which Unwired Server runs must have adequate permissions to impersonate the domain\user. For example, the Administrator account for the domain.
6. Replace the `beginOnlineLogin` call, which passes a username and password, with code that imports the certificate from Unwired Server, sets up the login credentials for the package, then logs in with this `beginOnlineLogin` API that takes no parameters.

```
// Import certificate from server
SUPLoginCertificate *lc = [cs
getSignedCertificateFromServer:@"<ServerName>\\ssotest"
withServerPassword:@"s1s2o3T4" withCertPassword:@"password"];
[[LogInfo sharedInstance]
testPassed:@"SAPSSOCertTest" :@"GetCertificateFromServer"];
NSLog(@"Imported certificate from server: subjectCN =
%@",lc.subjectCN);

// Attach certificate to sync profile
sp.certificate = lc;
[lc release];

// If package requires login first, use beginOnlineLogin API
// which takes no parameters
while ([SUPApplication connectionStatus] !=
SUPConnectionStatus_CONNECTED) {
  NSLog(@"waiting to connect...");
  sleep(2);
```

```
}
[CrmDatabase beginOnlineLogin];
```

Sample Code

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

```
//// SSO certificate APIs
@try
{
SUPConnectionProfile *sp = [SUP101_SUP101DB
getSynchronizationProfile];
[sp setDomainName:@"ssocert"];
// Get handle to the certificate store
SUPCertificateStore *cs = [SUPCertificateStore getDefault];

// Getting certificate from a file bundled with the app
NSString *certPath = [[NSBundle mainBundle]
pathForResource:@"sybase101"
ofType:@"p12"];
SUPLoginCertificate *lc_resource = [cs
getSignedCertificateFromFile:certPath withPassword:@"password"];
NSLog(@"Got certificate from resource file, subjectCN =
%@",lc_resource.subjectCN);
[[LogInfo sharedInstance]
testPassed:@"SAPSSOCertTest" :@"GetCertificateFromResourceFile"];

// Getting certificate from file in Documents directory
NSArray *arrayPaths =
NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
NSUserDomainMask,
YES);
NSString *docDir = [arrayPaths objectAtIndex:0];
certPath = [NSString stringWithFormat:@"%@/sybase101.p12",docDir];
SUPLoginCertificate *lc_doc = [cs
getSignedCertificateFromFile:certPath withPassword:@"password"];
NSLog(@"Got certificate from documents directory file, subjectCN =
%@",lc_doc.subjectCN);
[[LogInfo sharedInstance]
testPassed:@"SAPSSOCertTest" :@"GetCertificateFromDocumentsFile"];

// Distinguished name property
NSLog(@"Test distinguished name property, should be null: DN =
%@",lc_doc.distinguishedName);

// Import certificate from server
SUPLoginCertificate *lc = [cs
getSignedCertificateFromServer:@"<ServerName>\\ssotest"
withServerPassword:@"s1s2o3T4" withCertPassword:@"password"];
[[LogInfo sharedInstance]
testPassed:@"SAPSSOCertTest" :@"GetCertificateFromServer"];
NSLog(@"Imported certificate from server: subjectCN =
%@",lc.subjectCN);
```

```
// Storage and retrieval of certificate
if(![SUPDataVault vaultExists:@"vaultTest"])
vault = [SUPDataVault createVault:@"vaultTest"
withPassword:@"vaultPassword" withSalt:@"vaultSalt"];
else
vault = [SUPDataVault getVault:@"vaultTest"];
[vault lock];
[vault unlock:@"vaultPassword" withSalt:@"vaultSalt"];
[lc save:@"test" withVault:vault];
[vault lock];
[vault unlock:@"vaultPassword" withSalt:@"vaultSalt"];
NSLog(@"Certificate stored. Now get the cert from the data
vault....");
SUPLoginCertificate *lc2 = [SUPLoginCertificate load:@"test"
withVault:vault];
[vault lock];
NSLog(@"Certificate retrieved successfully: subjectCN =
%@",lc2.subjectCN);
if([lc2.subjectCN isEqualToString:lc.subjectCN])
[[LogInfo sharedInstance]
testPassed:@"SAPSSOCertTest" :@"SaveAndLoadCertificate"];
else
[[LogInfo sharedInstance]
testFailed:@"SAPSSOCertTest" :@"SaveAndLoadCertificate"];
[lc2 release];
NSLog(@"Test getting a nonexistent certificate from the vault, see if
we get the right exception...");
BOOL noCertificatePass = NO;
@try
{
SUPLoginCertificate *lc_none = [SUPLoginCertificate load:@"bogus"
withVault:vault];
} @catch(SUPDataVaultException* e)
{
noCertificatePass = YES;
NSLog(@"Got exception when trying to get nonexistent cert, exception
is %@: %@",[e name],[e reason]);
}
if(noCertificatePass)
[[LogInfo sharedInstance]
testPassed:@"SAPSSOCertTest" :@"NonExistentCertificate"];
else
[[LogInfo sharedInstance]
testFailed:@"SAPSSOCertTest" :@"NonExistentCertificate"];

// Delete certificate
BOOL deletePass = YES;
// Try to get the deleted certificate, should get an exception:
SUPLoginCertificate *lc3 = nil;
[vault unlock:@"vaultPassword" withSalt:@"vaultSalt"];
@try
{
[SUPLoginCertificate delete:@"test" withVault:vault];
lc3 = [SUPLoginCertificate load:@"test" withVault:vault];
deletePass = NO;
} @catch(NSException* e)
```

```
{
NSLog(@"Exception getting deleted cert: %@: %@",[e name],[e
reason]);
deletePass = YES;
}
NSLog(@"Retrieve cert that was deleted, should be null: lc3 =
%@",lc3);
if(lc3 != nil) deletePass = NO;
if(deletePass)
[[LogInfo sharedInstance]
testPassed:@"SAPSSOCertTest" :@"DeleteCertificate"];
else
[[LogInfo sharedInstance]
testFailed:@"SAPSSOCertTest" :@"DeleteCertificate"];


// changeVaultPassword for LoginCertificate
[vault lock];
[vault unlock:@"vaultPassword" withSalt:@"vaultSalt"];
[vault changePassword:@"newPassword" withSalt:@"vaultSalt"];
[vault lock];
[vault unlock:@"newPassword" withSalt:@"vaultSalt"];
[lc save:@"test" withVault:vault];
[vault lock];
[vault unlock:@"newPassword" withSalt:@"vaultSalt"];
SUPLoginCertificate *lc4 = [SUPLoginCertificate load:@"test"
withVault:vault];
[vault lock];
[vault unlock:@"newPassword" withSalt:@"vaultSalt"];

// Change password back so we can rerun the test
[vault changePassword:@"vaultPassword" withSalt:@"vaultSalt"];
[vault lock];
if([lc4.subjectCN isEqualToString:lc.subjectCN])
[[LogInfo sharedInstance]
testPassed:@"SAPSSOCertTest" :@"ChangeVaultPassword"];
else
[[LogInfo sharedInstance]
testFailed:@"SAPSSOCertTest" :@"ChangeVaultPassword"];
[lc4 release];

// Attach certificate to sync profile
sp.certificate = lc;
[lc release];
}
@catch(NSException *e)
{
MBOLogError(@"Exception in getting certificate");
MBOLogError(@"%@: %@",[e name],[e reason]);
[pool drain];
return;
}

// If package requires login first, use beginOnlineLogin API
// which takes no parameters
while ([SUPApplication connectionStatus] !=
```

```
SUPConnectionStatus_CONNECTED) {
  NSLog(@"waiting to connect...");
  sleep(2);
}
[CrmDatabase beginOnlineLogin];
```

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.

- `SUPCertificateStore` class - wraps platform-specific key/certificate store class, or file directory
- `SUPLoginCertificate` class - wraps platform-specific X.509 distinguished name and signed certificate
- `SUPConnectionProfile` class - includes the certificate attribute used for Unwired Server synchronization.
- `SUPDataVault` class - provides secure persistent storage on the device for certificates.

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.

```
// Obtain a reference to the certificate store

SUPCertificateStore *certStore = [SUPCertificateStore getDefault];

// Import a certificate from iPhone keychain (into memory)

NSString *label = ...; // ask user to select a label
NSString *password = ...; // ask the user for a password
SUPLoginCertificate *cert = [certStore getSignedCertificate:label
withPassword:password];

// Alternate code: import a certificate blob from the server into
memory (server must be specially configured for this):

NSString *windows_username = .... // Windows username for fileshare
on server where the password is stored
NSString *windows_password = .... // Windows password
NSString *cert_password = .... // Password to unlock the certificate
SUPLoginCertificate *cert = [certStore
getSignedCertificateFromServer:windows_username
withServerPassword:windows_password
withCertPassword:cert_password];

// Lookup or create data vault
NSString *vaultPassword = ...; // ask user or from O/S protected
storage
NSString *vaultName = "..."; // e.g. "SAP.CRM.CertificateVault"
```

```
NSString *vaultSalt = "..."; // e.g. a hard-coded random GUID
SUPDataVault *vault;
@try
{
// Get vault, or create it if it doesn't exist
        if(![SUPDataVault vaultExists:vaultName])
            vault = [SUPDataVault createVault:vaultName
withPassword:vaultPassword withSalt:vaultSalt];
        else
            vault = [SUPDataVault getVault:vaultName];

// Save certificate into data vault

        [vault unlock:vaultPassword withSalt:vaultSalt];
        [cert save:label withVault:vault];

}
@catch (NSException *ex)
{
    // Handle any errors
}
@finally
{
    // Make sure vault is locked even if an error occurs
    [vault lock];
}
```

Selecting a Certificate for Unwired Server Connections

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

```
@try
{
    [vault unlock:vaultPassword withSalt:vaultSalt];
    SUPLoginCertificate *cert = [SUPLoginCertificate load:@"myCert"
withVault:vault];
    SUPConnectionProfile *syncProfile = [SUP101_SUP101DB
getSynchronizationProfile];
    syncProfile.certificate = cert;
    [cert release];
}
@catch(NSException *ex)
    // Handle any errors
}
@finally
{
    // Make sure vault is locked even if an error occurs
    [vault lock];
}
```

Connecting to Unwired Server with a Certificate

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

```
[SUP101SUP101DB beginOnlineLogin];

// Handle login response

[SUP101SUP101DB subscribe];
```

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.

Type of Personalization Keys

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

A personalization parameter can be a primitive or complex type.

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

Getting and Setting Personalization Key Values

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

Consider a personalization key "pkcity" that is associated with the synchronization parameter "cityname". The following example shows how to get and set personalization key values:

```
//get personalization key values
SampleApp_PersonalizationParameters *pp =  [SUP101SUP101DB
getPersonalizationparameters];
MBOLogInfo(@”Personalization Parameter for City = %@”, pp.PKCity);

//Set personalization key values
pp.PKCity = @”Hull”;
[pp.save];  //save the new pk value.
while ([SUP101SUP101DB hasPendingOperations])
        [NSThread sleepForTimeInterval:0.2];
```

An operation can have a parameter that is one of the Sybase Unwired Platform list types (such as `SUPIntList`, `SUPStringList`, or `SUPObjectList`). For example, consider a method for an entity `Customer` with signature `AnOperation`:

```
SUPIntList *intlist = [SUPIntList getInstance];
[intlist add:1];
[intlist add:2];

Customer *thecustomer = [Customer find:101];
[thecustomer AnOperation:intlist];
```

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 API is now deprecated. Call or before saving synchronization parameters. After saving the synchronization parameters, call or again to retrieve the new values filtered by those parameters.

Changing Synchronization Parameters

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

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

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

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

Consider the `Customer` MBO that has a `cityname` synchronization parameter. This example shows how to retrieve customer data corresponding to Kansas City.

```
CustomerSynchronizationParameters *sp = [Customer
getSynchronizationParameters];
sp.size = 3;
sp.user = @"testuser";
sp.cityname = @"Kansas City";
[sp save];
while ([SUP101_SUP101DB hasPendingOperations])
    [NSThread sleepForTimeInterval:0.2];
```

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

Performing Mobile Business Object Synchronization

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

Before you can synchronize MBO changes with the server, you must subscribe the mobile application package deployed on server by calling `SUP101DB.subscribe()`. This also downloads certain data to devices for those that have default values. You can use the `OnImportSuccess` method in the defined `CallbackHandler` to check if data download has been completed.

You can then call the **submitPendingOperations(string synchronizationGroup)** operation through the publication.

You can use a publication mechanism, which allows as many as 32 simultaneous synchronizations. However, performing simultaneous synchronizations on several very large Unwired Server applications can impact server performance, and possibly affect other remote users.

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

```
[SUP101SUP101DB submitPendingOperations:@"mySyncGroup"];
```

Or, you can synchronize all synchronization groups:

```
[SUP101SUP101DB submitPendingOperations];
```

Message-Based Synchronization APIs

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

beginOnlineLogin

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

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

beginOnlineLogin sends a message to the Unwired Server with the username and password. The Unwired Server responds with a message to the client with the login success or failure. This method checks the `SUPApplication connectionStatus` and immediately fails if the status is not `SUPConnectionStatus_CONNECTED`. Make sure the connection is active before calling `beginOnlineLogin`, or implement the `onLoginFailure` callback handler to catch cases where it may fail, 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

```
+ (void)beginOnlineLogin:(NSString *)user password:(NSString *)pass
```

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.

  ```
  [SUP101_SUP101DB beginOnlineLogin:@"supAdminID"
  password:@"supPwd"];
  ```

subscribe

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

Prerequisites for using **subscribe**:

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

Syntax

```
+(void) subscribe
```

Parameters

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

Returns

None.

Examples

- **Subscribe to a Sample Application –** Subscribe to SUP101SUP101DB.
  ```
  [SUP101SUP101DB subscribe];
  ```

unsubscribe

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

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

Syntax

```
+(void) unsubscribe
```

Parameters

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

Returns

None.

Examples

- **Unsubscribe from a Sample Application –** Unsubscribe from SUP101_SUP101DB.

```
[SUP101SUP101DB unsubscribe];
```

suspendSubscription

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

Syntax

```
+(void) suspendSubscription
```

Parameters

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

Returns

None.

Examples

- **Suspend a Subscription –** Suspend the subscription to SUP101_SUP101DB.
  ```
  [SUP101SUP101DB suspendSubscription];
  ```

beginSynchronize

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

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

Syntax

```
+(void) beginSynchronize[:(SUPObjectList*)synchronizationGroups]
[withContext:(NSString*)context]
```

Parameters

- **synchronizationGroups –** specifies a list of a list of `SUPSynchronizationGroup` 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 iOS*.

Returns

None.

Examples

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

```
// Sync all groups

[SUP101SUP101DB beginSynchronize];
```

- **Synchronize a Particular Group –** Synchronize data for SUP101DB for the SUP101 group.

```
// Sync all groups

[SUP101SUP101DB beginSynchronize];

// Sync just for particular groups. In this case, we just
synchronize one group,
// the group for the SUP101Customer MBO.

SUPObjectList *sgs = [SUPObjectList getInstance];
[sgs add:[SUP101Customer getSynchronizationGroup]];
[SUP101SUP101DB beginSynchronize:sgs
withContext:@"customergroupcontext"];
```

resumeSubscription

Sends a resume request to the Unwired Server.

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

Syntax

```
+(void) resumeSubscription
```

Parameters

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

Returns

None.

Examples

- **Resume a Subscription –** Resume the subscription to SUP101_SUP101DB.
  ```
  [SUP101SUP101DB resumeSubscription];
  ```

recover

Sends a recover request to the Unwired Server.

The recover message notifies the Unwired Server to send down all the data related to the package.

Note: Do not use `recover` with DOE-based applications.

Syntax

```
+(void) recover
```

Parameters

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

Returns

On success, **onRecoverSuccess** callback handler method is called. On failure, **onRecoverFailure** callback handler is called.

Examples

- – Send down all data for SUP101SUP101DB.
  ```
  [SUP101SUP101DB recover];
  ```

Retrieving Information about Synchronization Groups

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

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

SUPLogRecord API

Every package has a `LogRecordImpl` table in its own database. The Unwired Server can send import messages with `LogRecordImpl` records as part of its response to replay requests (success or failure). `LogRecord` stores two types of logs.

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

The Unwired Server can embed a "log" JSON array into the header of a server message; the array is written to the `LogRecordImpl` table by the client. The client application can also write its own records. Each entity has a method called `newLogRecord`, which allows the entity to write its own log record. The `LogRecordImpl` table has "component" and "entityKey" columns that associate the log record entry with a particular MBO and primary key value.

```
SUPObjectList *salesorders = [SampleAppSales_order findAll];
  if([salesorders size] > 0)
  {
    SampleAppSales_order * so = [salesorders item:0];
    SampleAppLogRecordImpl *lr = [so newLogRecord:
      [SUPLogLevel INFO] withMessage:@"testing
record"];
    MBOLogError(@"Log record is: %@",lr);

    // submitting log records
    [SUP101SUP101DB submitLogRecords];
    while ([SUP101SUP101DB hasPendingOperations])
      {
        [NSThread sleepForTimeInterval:0.2];
      }
  }
```

You can use the `getLogRecords` method to return log records from the table.

```
SUPQuery *query = [SUPQuery getInstance];
      SUPObjectList *loglist = [SUP101SUP101DB getLogRecords:query];
    for(id o in loglist)
    {
        LogRecordImpl *log = (LogRecordImpl*)o;
        MBOLogError(@"Log Record %llu: Operation = %@, Timestamp =
%@,

MBO = %@, key= %@,message=%@",log.messageId,log.operation,
  [SUPDateTimeUtil
```

```
toString:log.timestamp],log.component,log.entityKey,log.message);
}
```

Each mobile business object has a `getLogRecords` instance method that returns a list of all the log records that have been recorded for a particular entity row in a mobile business object:

```
SUPObjectList *salesorders = [SampleAppSales_order findAll];
if([salesorders size] > 0)
{
  SampleAppSales_order * so = [salesorders item:0];
  SUPObjectList *loglist = [so getLogRecords];
for(id o in loglist)
{
  LogRecordImpl *log = (LogRecordImpl*)o;
  MBOLogError(@"Log Record %llu: Operation = %@, Timestamp = %@,

MBO = %@, key= %@,message=%@",log.messageId,log.operation,
  [SUPDateTimeUtil
toString:log.timestamp],log.component,log.entityKey,log.message);
}
```

Mobile business objects that support dynamic queries can be queried using the synthetic attribute `hasLogRecords`. This attribute generates a subquery that returns true if an entity row has any log records in the database, otherwise it returns false. The following code example prints out a list of customers, including first name, last name, and whether the customer row has log records:

```
SUPQuery *query = [SUPQuery getInstance];
[query select:@"x.surrogateKey,x.fname,x.lname,x.hasLogRecords"];
[query from:@"Customer":@"x"];
SUPQueryResultSet *qrs = [SUP101SUP101DB executeQuery:query];
MBOLogError(@"%@",[qrs.columnNames toString]);
for(SUPDataValueList *row in qrs.array)
{
  MBOLogError(@"%@",[row toString]);
}
```

If there are a large number of rows in the MBO table, but only a few have log records associated with them, you may want to keep an in-memory object to track which rows have log records. You can define a class property as follows:

```
NSMutableArray* customerKeysWithLogRecords;
```

After data is downloaded from the server, initialize the array:

```
customerKeysWithLogRecords = [[NSMutableArray alloc]
initWithCapacity:20];
SUPObjectList *allLogRecords = [SUP101SUP101DB getLogRecords:nil];
for(id<SUPLogRecord> lr in allLogRecords)
{
  if(([lr entityKey] != nil) && ([[lr component] compare:@"Customer"]
== 0))
    [customerKeysWithLogRecords addObject:[lr entityKey]];
}
```

You do not need database access to determine if a row in the Customer MBO has a log record. The following expression returns true if a row has a log record:

```
BOOL hasALogRecord = [customerKeysWithLogRecords containsObject:
                                    [customerRow keyToString]];
```

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

```
- (void)processLogs
{
  SUPQuery *query = [SUPQuery getInstance];
  SUPObjectList *logRecords = [SUP101SUP101DB getLogRecords:query];

  for(id<SUPLogRecord> log in logRecords)
  {
    // Log warning message
    NSLog(@"log %@: %@ code:%d msg:%@",[log component],[log
entityKey],[log code],[log message]);
    if([[log component] isEqualToString:@"Customer"])
    {
      NSNumberFormatter *formatter = [[NSNumberFormatter alloc]
init];
      int64_t surrogateKey = [[formatter numberFromString:[log
entityKey]] longLongValue];
      [formatter release];
      SUP101Customer *c = [SUP101Customer find:surrogateKey];
      if(c.pending)
        [c cancelPending];
      [log delete];
      [log submitPending];
    }
  }

}
```

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.

Log Level and Tracing APIs

The `MBOLogger` class enables the client to add log levels to messages reported to the console.The application can set the log level using the `setLogLevel` method.

In ascending order of detail (or descending order of severity), the log levels defined are `LOG_OFF` (no logging), `LOG_FATAL`, `LOG_ERROR`, `LOG_WARN`, `LOG_INFO`, and `LOG_DEBUG`.

Macros such as `MBOLogError`, `MBOLogWarn`, and `MBOLogInfo` allow application code to write console messages at different log levels. You can use the method `setLogLevel` to

determine which messages get written to the console. For example, if the application sets the log level to `LOG_WARN`, calls to `MBOLogInfo` and `MBOLogDebug` do not write anything to the console.

```
[MBOLogger setLogLevel:LOG_INFO];
MBOLogInfo(@"This log message will print to the console”);
[MBOLogger setLogLevel:LOG_WARN];
MBOLogInfo(@"This log message will not print to the console");
MBOLogError(@"This log message will print to the console");
```

Tracing APIs

The SQL tracing API enables tracing of client database operations, and message headers sent to and received from the Unwired Server. The API is configured in the connection profile and synchronization profile.

```
SUPConnectionProfile *cp = [SUP101SUP101DB getConnectionProfile];

// To enable trace of client database operations (SQL statements,
etc.)
[cp enableTrace:YES];

// To enable trace of client database operations with values also
displayed
[cp enableTrace:YES withPayload:YES];

// To disable trace of client database operations
[cp enableTrace:NO];

// To enable trace of message headers sent to the server and received
from the server
// (this replaces the MBODebugLogger and MBODebugSettings used in
earlier versions of SUP)
[cp.syncProfile enableTrace:YES];

// To enable trace of both message headers and content, including
credentials
[cp.syncProfile enableTrace:YES withPayload:YES];

// To disable messaging trace
[cp.syncProfile enableTrace:NO];
```

Printing Log Messages

The following code example retrieves log messages resulting from login failures where the Unwired Server writes the failure record into the `LogRecordImpl` table. You can implement the `onLoginFailure` callback to print out the server message.

```
SUPQuery * query = [SUPQuery newInstancegetInstance];
SampleAppLogRecordImplList* loglist = (SUP101LogRecordImplList*)
[SUP101SUP101DB getLogRecords:query];
  for(SUP101LogRecordImpl* log in loglist)
  {
    MBOLogError(@"Log Record %llu: Operation = %@, Component = %@,
message = %@", log.messageId, log.operation,
```

```
log.component,log.message);
  }
```

Security APIs

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

Encryption of Client Data

The iOS Sybase Unwired Platform client libraries internally encrypt data before sending it over the wire, using its own encryption layer. Communication is performed over HTTP.

Encrypting the Client Database

There are two APIs that you can use to encrypt the client database.

`generateEncryptionKey()` causes a new random encryption key to be generated and used to encrypt the database. This key is immediately set in the connection profile.

```
NSString *newKey = nil;
[SUP101SUP101DB generateEncryptionKey];
newKey = [[SUP101SUP101DB getConnectionProfile] getEncryptionKey];
NSLog(@"generated encryption key = %@",newKey);
[SUP101SUP101DB closeConnection];
```

`changeEncryptionKey()` causes the database to be encrypted with the new key passed in.

```
[SUP101SUP101DB
changeEncryptionKey:@"longEncryptionKeyValueABCDEFG"];
[SUP101SUP101DB closeConnection];
```

Removing Encryption from the Database

To remove encryption from the database, use the `changeEncryptionKey` API and pass in a null value.

```
[SUP101SUP101DB changeEncryptionKey:nil];
[SUP101SUP101DB closeConnection];
```

Accessing a Previously Encrypted Database

If an application is starting up using a previously existing database that has been encrypted, the encryption key must be set in the connection profile before any database operations are done. This is done using the connection profile's `setEncryptionKey()` API.

```
[[SUP101SUP101DB getConnectionProfile] setEncryptionKey:newKey];
[SUP101SUP101DB closeConnection];
```

SUPDataVault

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

If you have installed the `SybaseDataProvider.apk` package, you can use the `SUPDataVault` 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
- Control access for a vault that is shared by multiple iOS applications

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

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

createVault

Creates a new secure store.

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

Syntax

```
+ (SUPDataVault*)createVault:(NSString*)name withPassword:
(NSString*)password withSalt:(NSString*)salt;
```

Parameters

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

Returns

createVault creates a `SUPDataVault` instance.

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

Examples

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

```
@try
{
    if(![SUPDataVault vaultExists:@"myVault"])
    {
        oVault = [SUPDataVault createVault:@"myVault"
                                withPassword:@"goodPassword"
                                withSalt:@"goodSalt"];
    }
}
@catch ( NSException *e )
{
    NSLog(@"SUPDataVaultException: %@",[e description]);
}
```

vaultExists

Tests whether the specified vault exists.

Syntax

```
+ (BOOL)vaultExists:(NSString*)name;
```

Parameters

- **name –** The vault name.

Returns

vaultExists can return the following values:

Returns	Indicates
YES	The vault exists.

Returns	Indicates
NO	The vault does not exist.

Examples

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

```
if ([SUPDataVault vaultExists:@"myVault"])
{
    [SUPDataVault deleteVault:@"myVault"];
}
```

getVault

Retrieves a vault.

Syntax

```
+ (SUPDataVault*)getVault:(NSString*)name;
```

Parameters

- **name –** The vault name.

Returns

getVault returns a `SUPDataVault` instance.

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

deleteVault

Deletes the specified vault from on-device storage.

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

Syntax

```
+ (void)deleteVault:(NSString*)name;
```

Parameters

- **name –** The vault name.

Examples

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

```
@try
{
    if([SUPDataVault vaultExists:@"myVault"])
    {
        [SUPDataVault deleteVault:@"myVault"];
    }
}
@catch ( NSException *e )
{
    NSLog(@"SUPDataVaultException: %@",[e description]);
}
```

lock

Locks the vault.

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

Syntax

```
- (void)lock;
```

Examples

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

```
[oVault lock];
```

isLocked

Tests whether the vault is locked.

Syntax

```
- (BOOL)isLocked;
```

Returns

isLocked can return the following values:

Returns	Indicates
YES	The vault is locked.
NO	The vault is unlocked.

unlock

Unlocks the vault.

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

Syntax

```
- (void)unlock:(NSString*)password withSalt:(NSString*)salt;
```

Parameters

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

Returns

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

Examples

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

```
@try
{
   [oVault unlock:@"password" withSalt:@"salt"];
}
@catch(SUPDataVaultException *e)
{
   NSLog(@"Exception will be thrown for bad password");
}
```

setLockTimeout

Determines how long a vault remains unlocked.

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

Syntax

```
- (void)setLockTimeout:(int32_t)timeout;
```

Parameters

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

Examples

- **Set the Lock Timeout –** Sets the lock timeout to 1 hour.
  ```
  [oVault setLockTimeout:3600];
  ```

getLockTimeout

Retrieves the configured lock timeout period.

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

Syntax

```
- (int32_t)getLockTimeout;
```

Returns

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

Examples

- **Set the Lock Timeout –** Retrieves the lock timeout in seconds.
  ```
  timeout = [oVault getLockTimeout];
  ```

setRetryLimit

Sets the retry limit value for the vault.

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

Syntax

```
- (void)setRetryLimit:(int32_t)limit;
```

Parameters

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

Examples

- **Set the Retry Limit –** Sets the retry limit to 5 attempts.
  ```
  [oVault setRetryLimit:5];
  ```

getRetryLimit

Retrieves the retry limit value for the vault.

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

Syntax

```
- (int32_t)getRetryLimit;
```

Returns

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

Examples

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

  ```
  int retrylimit = [oVault getRetryLimit];
  ```

setString

Stores a string object in the vault.

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

Syntax

```
- (void)setString:(NSString*)name withValue:(NSString*)value;
```

Parameters

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

Examples

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

  ```
  NSString *teststring = @"ABCDEFabcdef";
  @try {
     [oVault unlock:@"goodPassword" withSalt:@"goodSalt"];
     [oVault setString:@"testString" withValue:teststring];
  ```

```
}
@catch (NSException *e) {
   NSLog(@"Exception: %@",[e description]);
}
@finally {
   [oVault lock];
}
```

getString

Retrieves a string value from the vault.

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

Syntax

```
- (NSString*)getString:(NSString*)name;
```

Parameters

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

Returns

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

Examples

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

```
NSString *retrievedstring = nil;

@try {
   [oVault unlock:@"goodPassword" withSalt:@"goodSalt"];
   retrievedstring = [oVault getString:@"testString"];
}
@catch (NSException *e) {
   NSLog(@"Exception: %@",[e description]);
}
@finally {
   [oVault lock];
}
```

setValue

Stores a binary object in the vault.

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

Syntax

```
- (void)setValue:(NSString*)name withValue:(NSData*)value;
```

Parameters

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

Examples

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

```
@try {
   [oVault unlock:@"goodPassword" withSalt:@"goodSalt"];
   [oVault setValue:@"testValue" withValue:testvalue];
}
@catch (NSException *e) {
   NSLog(@"Exception: %@",[e description]);
}
@finally {
   [oVault lock];
}
```

getValue

Retrieves a binary object from the vault.

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

Syntax

```
- (NSData*)getValue:(NSString*)name;
```

Parameters

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

Returns

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

Examples

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

```
NSData *retrievedvalue = nil;

@try {
   [oVault unlock:@"goodPassword" withSalt:@"goodSalt"];
   retrievedvalue = [oVault getValue:@"testValue"];
}
@catch (NSException *e) {
   NSLog(@"Exception: %@",[e description]);
}
@finally {
   [oVault lock];
}
```

changePassword

Changes the password for the vault.

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

Syntax

```
- (void)changePassword:(NSString*)newPassword withSalt:
(NSString*)newSalt;
```

Parameters

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

Examples

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

  ```
  @try
  {
     [oVault unlock:@"goodPassword" withSalt:@"goodSalt"];
     [oVault changePassword:@"newPassword" withSalt:@"newSalt"];
  }
  @catch (NSException *e) {
     NSLog(@"Exception: %@",[e description]);
  }
  @finally
  {
     [oVault lock];
  }
  ```

setAccessGroup

Sets the access group if multiple application share a data vault.

This method is used only for iOS applications, and must be called before accessing any `DataVault` methods. The access group must be set only if a vault is shared by multiple iPhone applications. If the vault is used only by one application, do not set the access group. The access group is listed in the `keychain-access-groups` property of the `entitlements` plist file. The recommended format is `".com.yourcompany.DataVault"`.

Syntax

```
+ (void)setAccessGroup:(NSString *)accessGroup;
```

Parameters

- **accessGroup –** The access group name.

Examples

- **Sets the Access Group Name –** Sets the access group name so that multiple iOS applications can access the data vault.

  ```
  [oVault
  setAccessGroup:@"accessGroupName.com.yourcompany.DataVault"];
  ```

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.

Callback Handler 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. A callback handler provides message notifications and success or failure messages related to message-based synchronization. To receive callbacks, register your own handler with a database. You can use `SUPDefaultCallbackHandler` as the base class. In your handler, override the particular callback you want to use (for example, `onImport`).

Because both the database and entity handler can be registered, your handler may get called twice for a mobile business object import activity. The callback is executed in the thread that is performing the action. For example, `onImport` is always called from a thread other than the main application thread.

When you receive the callback, the particular activity is already complete.

The `SUPCallbackHandler` protocol consists of these callbacks:

- **onImport:(id)entityObject; –** invoked when an `import` is received. If Unwired Server accepts a requested change, it sends one or more `import` messages to the client, containing data for any created, updated, or deleted row that has changed on the Unwired Server as a result of the `replay` request.
- **onReplayFailure:(id)entityObject; –** invoked when a replay failure is received from the Unwired Server, whenever a particular device sends a create, update, or delete operation and the operation fails (Unwired Server rejects the requested operation).
- **onReplaySuccess:(id)entityObject; –** invoked when a replay success is received from the Unwired Server, whenever a particular device sends a create, update, or delete operation and the operation succeeds (Unwired Server accepts the requested operation).
- **onLoginFailure; –** invoked when a login failure message is received from the Unwired Server.
- **onLoginSuccess; –** called when a login result is received by the client.
- **onSubscribeFailure; –** invoked when a subscribe failure message is received from the Unwired Server, whenever an object in a subscribed entity changes.
- **onSubscribeSuccess; –** invoked when a subscribe success message is received from the Unwired Server, whenever an object in a subscribed entity changes.
- **- (int32_t)onSynchronize:(SUPObjectList*)syncGroupList withContext: (SUPSynchronizationContext*)context; –** invoked when the synchronization status changes. This method is called by the database class `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 `SUPSynchronizationContext` object passed into this method has a "status" attribute that contains the current synchronization status. The possible statuses are defined in the `SUPSynchronizationStatusType` enum, and include:

  - **SUPSynchronizationStatus_STARTING –** passed in when `beginSynchronize` is called.
  - **SUPSynchronizationStatus_UPLOADING –** synchronization status upload in progress.
  - **SUPSynchronizationStatus_DOWNLOADING –** synchronization status download in progress.
  - **SUPSynchronizationStatus_FINISHING –** synchronization completed successfully.
  - **SUPSynchronizationStatus_ERROR –** synchronization failed.

  For message-based synchronization, only the status values of STARTING, FINISHING, and ERROR are passed into this method.

  This callback handler returns `SUPSynchronizationActionCONTINUE`, unless the user cancels synchronization, in which case it returns

`SUPSynchronizationActionCANCEL`. This code example prints out the groups in a synchronization status change:

```
{
     MBOLogInfo(@"Synchronization response");

MBOLogInfo(@"================================================");

     for(id<SUPSynchronizationGroup> sg in syncGroupList)
     {
          MBOLogInfo(@"group = %@",sg.name);
     }

MBOLogInfo(@"================================================");

     if(context != nil)
     {
          MBOLogInfo(@"context: %ld,
%@",context.status,context.userContext);
     } else {
          MBOLogInfo(@"context is null");
     }

MBOLogInfo(@"================================================");

     return SUPSynchronizationActionCONTINUE;
}
```

- **onSuspendSubscriptionFailure; –** invoked when a call to suspend fails.
- **onSuspendSubscriptionSuccess; –** invoked when a suspend call is successful.
- **onResumeSubscriptionFailure; –** invoked when a resume call fails.
- **onResumeSubscriptionSuccess; –** invoked when a resume call is successful.
- **onUnsubscribeFailure; –** invoked when an unsubscribe call fails.
- **onUnsubscribeSuccess; –** invoked when an unsubscribe call is successful.
- **onImportSuccess; –** invoked when `onImport` succeeds.
- **onMessageException:(NSException*e); –** invoked when an exception occurs during message processing. Other callbacks in this interface (whose names begin with "on") are invoked inside a database transaction. If the transaction is rolled back due to an unexpected exception, this operation is called with the exception (before the rollback occurs).
- **onTransactionCommit; –** invoked on transaction commit.
- **onTransactionRollback; –** invoked on transaction rollback.
- **onResetSuccess; –** invoked when reset is successful.
- **onSubscriptionEnd; –** invoked on subscription end. `OnSubscriptionEnd` can occur when the device is registered, unlike `OnUnsubscribeSuccess`.
- **onStorageSpaceLow; –** invoked when storage space is low.
- **onStorageSpaceRecovered; –** invoked when storage space is recovered.
- **onConnectionStatusChange:(SUPDeviceConnectionStatus)connStatus: (SUPDeviceConnectionType)connType:(int32_t)errCode:(NSString*)errString; –** the application should call the register callback handler with a database class, and

implement the `onConnectionStatusChange` method in the callback handler. The API allows the device application to see what the error is in cases where the client cannot connect to the Unwired Server. `SUPDeviceConnectionStatus` and `SUPDeviceConnectionType` are defined in `SUPConnectionUtil.h`:

```
typedef enum {
  WRONG_STATUS_NUM = 0,
  // device connected
  CONNECTED_NUM = 1,
  // device not connected
  DISCONNECTED_NUM = 2,
  // device not connected because of flight mode
  DEVICEINFLIGHTMODE_NUM = 3,
  // device not connected because no network coverage
  DEVICEOUTOFNETWORKCOVERAGE_NUM = 4,
  // device not connected and waiting to retry a connection
  WAITINGTOCONNECT_NUM = 5,
  // device not connected becauseroaming was set to false
  // and device is roaming
  DEVICEROAMING_NUM = 6,
  // device not connected because of low space.
  DEVICELOWSTORAGE_NUM = 7
} SUPDeviceConnectionStatus;

typedef enum {
  WRONG_TYPE_NUM = 0,
  // iPhone has only one connection type
  ALWAYS_ON_NUM = 1
} SUPDeviceConnectionType;
```

This code example shows how to register a handler to receive a callback:

```
DBCallbackHandler* handler = [DBCallbackHandler newHandler];
[iPhoneSMTestDB registerCallbackHandler:handler];
[handler release];

MBOCallbackHandler* mboHandler = [MBOCallbackHandler newHandler];
[Product registerCallbackHandler:mboHandler];
[mboHandler release];
```

ApplicationCallback API

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

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

Table 2. Callbacks in the ApplicationCallback Interface

Callback	Description
`- (void)onApplicationSettingsChanged :(SUPString-List*)names`	Invoked when one or more application settings have been changed by the server administration.
`- (void)onConnectionStatusChanged :(SUPInt)connectionStatus :(SUPInt)errorCode : (SUPNullableString)errorMessage`	Invoked when the connection status changes. The possible connection status values are defined in the `ConnectionStatus` class.
`- (void)onDeviceConditionChanged :(SUPInt)deviceCondition`	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 :(SUPInt)registrationStatus :(SUPInt)errorCode :(SUPNullableString)errorMessage`	Invoked when the registration status changes. The possible registration status values are defined in the `RegistrationStatus` class.

Apple Push Notification API

The Apple Push Notification API allows applications to provide various types of push notifications to devices, such as sounds (audible alerts), alerts (displaying an alert on the screen), and badges (displaying an image or number on the application icon). Push notifications require network connectivity.

The client library `libclientrt` wraps the Apple Push Notification API in the file `SUPPushNotification.h`.

In addition to using the Apple Push Notification APIs in a client application, you must configure the push configuration on the server. This is performed under **Server Configuration** > **Messaging** > **Apple Push Configuration** in Sybase Control Center. You must configure the device application name (for push), the device certificate (for push), the Apple gateway, and the gateway port.

The following API methods abstract the Unwired Server, resolve the push-related settings, and register with an Apple Push server, if required. You can call these methods in the "applicationDidFinishLaunching" function of the client application:

```
@interface SUPPushNotification : NSObject
{

}
+(void)setupForPush:(UIApplication*)application;
+(void)deviceTokenForPush:(UIApplication*)application deviceToken:
(NSData
*)devToken;
+(void)pushRegistrationFailed:(UIApplication*)application
errorInfo:(NSError *)err;
+(void)pushNotification:(UIApplication*)application
notifyData:(NSDictionary *)userInfo;

+(void)setupForPush:(UIApplication*)application
```

After a device successfully registers for push notifications through Apple Push Notification Service, iOS calls the `didRegisterForRemoteNotificationWithDeviceToken` method in the client application. iOS passes the registered device token to this function, and the functions calls the `deviceTokenForPush` API to pass the device token to Unwired Server:

```
+(void)deviceTokenForPush:(UIApplication*)application deviceToken:
(NSData
*)devToken
```

If for any reason the registration with Apple Push Notification Service fails, iOS calls `didFailToRegisterForRemoteNotificationsWithError` in the client application which calls the following API:

```
+(void)pushRegistrationFailed:(UIApplication*)application
errorInfo: (NSError *)err
```

When iOS receives a notification from Apple Push Notification Service for an application, it calls `didReceiveRemoteNotification` in the client application. This calls the `pushNotification` API:

```
+(void)pushNotification:(UIApplication*)application
notifyData:(NSDictionary *)userInfo
```

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.

Pass an instance of the listener to the synchronize methods.

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

comparing it to the possible values in the `SyncStatusState` enumeration, the application can react accordingly to the state of the synchronization.

Possible uses of 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 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:

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.

Retrieving Data from Mobile Business Objects

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

Object Queries

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

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

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

This method retrieves all customers:

```
SUPObjectList *customers = [SampleAppCustomer findAll] ;
```

The preceding Object Query results in this generated method:

Consider an object query on a Customer MBO to find customers by last name. You can construct the query as follows:

```
Select  x.* from Customer x where x.lname =:param_lname
```

where `param_lname` is a string parameter that specifies the last name. Assume that the query above is named **findBylname**

This generates the following Client Object API:

```
(Customer *)findBylname : (NSString *)param_lname;
```

The above API can then be used just like any other read API. For example:

```
SampleApp_Customer * thecustomer = [ SampleApp_Customer findBylname:
@"Delvin"];
```

For each object query that returns a list, additional methods are generated that allow the caller to select and sort the results. For example, consider an object query, **findByCity**, which returns a list of customers from the same city. Since the return type is a list ,the following methods would be generated. The additional methods help the user with ways to specify how many results rows to skip, and how many subsequent result rows to return.

```
+ (SUPObjectList*) findByCity:(NSString*) city;
+ (SUPObjectList*) findByCity:(NSString*) city skip;
(int32_t) skip take:(int32_t)take;
```

SUPQuery and Related Classes

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

Table 3. SUPQuery and Related Classes

Class	Description
SUPQuery	Defines arbitrary search methods and can be composed of search conditions, object/row state filter conditions, and data ordering information.
SUPAttributeTest	Defines filter conditions for MBO attributes.
SUPCompositeTest	Contains a method to combine test criteria using the logical operators AND, OR, and NOT to create a compound filter.
SUPQueryResultSet	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 `SUPQuery` class is included in the client object API. The `SUPQuery` 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:

- **SUPTestCriteria –** criteria used to filter returned data.
- **SUPSortCriteria –** 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.

`SUPTestCriteria` can be an `SUPAttributeTest` or a `SUPCompositeTest`.

TestCriteria

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

```
SUPQuery *query = [SUPQuery getInstance];
[query select:@"c.fname,c.lname,s.order_date,s.region"];
[query  from:@"Customer":@"c"];
[query  join:@"SalesOrder":@"s":@"s.cust_id":@"c.id"];
query.testCriteria = [SUPAttributeTest match:@"c.lname":@"Devlin"];
SUPQueryResultSet* resultSet = [SUP101SUP101DB executeQuery:query];
if(resultSet == nil)
  {
    MBOLog(@"executeQuery Failed !!");
    return;
  }
for(SUPDataValueList* result in resultSet)
{
    MBOLog(@"Firstname,lastname,order date,region = %@ %@ %@ %@",
    [SUPDataValue  getNullableString:[result item:0]],
    [SUPDataValue getNullableString:[result item:1]],
    [[SUPDataValue getNullableDate:[result item:2]] description],
    [SUPDataValue getNullableString:[result item:3]]);
}
```

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

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

- IS_NULL
- NOT_NULL
- EQUAL
- NOT_EQUAL
- LIKE
- NOT_LIKE
- LESS_THAN
- LESS_EQUAL
- GREATER_THAN
- GREATER_EQUAL
- CONTAINS
- STARTS_WITH
- ENDS_WITH
- DOES_NOT_START_WITH
- DOES_NOT_END_WITH
- DOES_NOT_CONTAIN
- IN
- NOT_IN
- EXISTS
- NOT_EXISTS

For example, the Objective-C code shown below is equivalent to this SQL query:

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

```
SUPQuery *query = [SUPQuery getInstance];
SUPAttributeTest *test = [SUPAttributeTest getInstance];
test.attribute = @"id";
SUPObjectList *v = [SUPObjectList getInstance];
[v add:@"1"];
[v add:@"2"];
[v add:@"3"];
test.testValue = v;
test.operator = SUPAttributeTest_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 Objective-C 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)
```

SortCriteria

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

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 SUPQuery object to limit the result set:

```
SUPQuery *query = [SUPQuery newInstance];
[query setSkip:10];
[query setTake:2];
SUPObjectList *customerlist = [SampleAppCustomer
findWithQuery:query];
```

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.

For iOS, we need a code sample equivalent to this WM sample:

```
SUPQuery *query1 = [SUPQuery getInstance];
[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:

```
NSString *groupByItem = @"c.id, c.name";
SUPQuery *query1 = [SUPQuery getInstance];

//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:

```
SUPQuery *query2 = [SUPQuery getInstance];
[query2 select:@"c.id, SUP(c.id)"];
[query2 from:@"AllType":@"c"];
SUPAttributeTest *ts = [SUPAttributeTest getInstance];
ts.attribute = @"c.id";
ts.testValue = @"0";
ts.operator = SUPAttributeTest_GREATER_EQUAL;
[query2 where:ts];
[query2 groupBy:@"c.id"];

SUPAttributeTest *ts2 = [SUPAttributeTest getInstance];
ts2.attribute = @"c.id";
ts2.testValue = @"0";
ts2.operator = SUPAttributeTest_GREATER_EQUAL;
[query2 having:ts2];
```

Concatenating Queries

Concatenate two queries having the same selected items.

The `SUPQuery` 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:

```
SUPQuery *query1 = [SUPQuery getInstance];
//other code for query1

SUPQuery *query2 = [SUPQuery getInstance];
//other code for query 2

SUPQuery *query3 = (SUPQuery*)[query1 except:query2];
[SUP101SUP101DB 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 Objective-C 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 Objective-C code:

```
SUPQuery *query1 = [SUPQuery getInstance];
[query1 select:@"b.id"];
[query1 from:@"AllType":@"b"];
SUPQuery *query2 = [SUPQuery getInstance];
[query2 select:@"a.id"];
[query2 fromQuery:query1:@"a"];
SUPAttributeTest *ts = [SUPAttributeTest getInstance];
ts.attribute = @"a.id";
[ts setTestValue:@"1"];
[query2 where:ts];
SUPQueryResultSet *qs = [SUP101DB executeQuery:query2];
```

You can use a subquery as the selected item of a query. Use the `SelectItem` to set selected items directly. For example, the Objective-C 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 Objective-C code:

```
SUPQuery *selQuery = [SUPQuery getInstance];
[selQuery select:@"count(1)"];
[selQuery from:@"AllType":@"c"];
SUPAttributeTest *ttt = [SUPAttributeTest getInstance];
ttt.attribute = @"c.id";
ttt.operator = SUPAttributeTest_GREATER_EQUAL;
SUPColumn *cl = [SUPColumn getInstance];
cl.alias = @"d";
cl.attribute = @"id";
ttt.testValue = cl;
[selQuery where:ttt];

SUPObjectList *selectItems = [SUPObjectList getInstance];
SUPSelectItem *item = [SUPSelectItem getInstance];
item.query = selQuery;
item.asAlias = @"cn";
[selectItems add:item];
SUPQuery *subQuery2 = [SUPQuery getInstance];
subQuery2.selectItems = selectItems;
[subQuery2 from:@"AllType" :@"d"];
SUPQueryResultSet *qs = [SUP101DB executeQuery:subQuery2];
```

CompositeTest

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

Complex Example

This example shows the usage of `CompositeTest`, `SortCriteria`, and `Query` to locate all customer objects based on particular criteria.

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

```
SUPQuery *props = [SUPQuery getInstance];
// Define the attribute based conditions.
// Users can pass in a string if they know the attribute name. R1
column name = attribute name.
SUPCompositeTest *innerCompTest = [SUPCompositeTest getInstance];
[innerCompTest setOperator:SUPCompositeTest_OR];
[innerCompTest add:[SUPAttributeTest equal:@"state":@"CA"]];
[innerCompTest add:[SUPAttributeTest equal:@"state":@"NY"]];

SUPCompositeTest *outerCompTest = [SUPCompositeTest getInstance];
[outerCompTest setOperator:SUPCompositeTest_OR];
[outerCompTest add:[SUPAttributeTest equal:@"fname":@"Jane"]];
[outerCompTest add:[SUPAttributeTest equal:@"lname":@"Doe"]];

[outerCompTest add:innerCompTest];

// Define the ordering:
SUPSortCriteria *sort = [SUPSortCriteria getInstance];

[sort add:[SUPAttributeSort ascending:@"fname"]];
[sort add:[SUPAttributeSort ascending:@"lname"]];

// Set the Query object:
props.testCriteria = (SUPTestCriteria*)outerCompTest;
props.sortCriteria = sort;
props.skip = 10;
props.take = 5;

SUPObjectList * customers2 = [TestCRUDCustomer findWithQuery:props];
```

SUPQueryResultSet

The `SUPQueryResultSet` class provides for querying a result set from the dynamic query API. `SUPQueryResultSet` 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 `SUPQuery`, filling in the criteria for the query, and filtering the query results:

```
SUPQuery *query [SUPQuery getInstance];
[query select:@"c.fname,c.lname,s.order_date,s.region"];
[query from:@"Customer":@"c"];
[query join:@"SalesOrder":@"s":@"s.cust_id":@"c.id"];
SUPAttributeTest *at = [SUPAttributeTest getInstance];
at.attribute = @"lname";
at.testValue = @"Devlin";
at.operator = SUPAttributeTest_EQUAL;
query.testCriteria = at;
SUPQueryResultSet *qrs = [SUP101DB executeQuery:query];
while ([qrs next])
{
NSLog(@"%@,",[qrs getString:1 withName:@"c.fname"]);
NSLog(@"%@,",[qrs getString:2 withName:@"c.lname"]);
NSLog(@"%@,",[[qrs getDate:3 withName:@"s.order_date"]
description]);
NSLog(@"%@\n",[qrs getString:4 withName:@"s.region"]);
}



}
```

Retrieving Relationship Data

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

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

```
SampleApp_Customer *onecustomer = [SampleApp_Customer find:101];
SUPObjectList *orders = onecustomer.salesOrders;
```

Given an order, you can access its customer information.

```
SampleApp_Sales_order * order = [SampleApp_Sales_order *find: 2001];
SampleApp_Customer *thiscustomer = order.customer;
```

Persistence APIs

The persistence APIs include operations and object state APIs.

Operations APIs

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

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

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

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`.

(void)create

Example 1: Supports `create` operations on parent entities. The sequence of calls is:

```
SampleAppCustomer *newcustomer = [[SampleAppCustomer alloc] init];
newcustomer.fname = @”John”;
...  //Set the required fields for the customer
[newcustomer create];
[newcustomer submitPending];
while ([SUP101SUP101DB hasPendingOperations])
        [NSThread sleepForTimeInterval:0.2];
```

Example 2: Supports create operations on child entities.

```
SampleAppSales_Order *order = [[SampleAppSales_Order alloc] init];
order.region = @"Eastern";
...  //Set the other required fields for the order

SampleAppCustomer *customer = [SampleAppCustomer find:1008];
[order setCustomer:customer];
[order create];
[order.customer refresh]; //refresh the parent
[order.customer submitPending];  //call submitPending on the parent.
while ([SUP101SUP101DB hasPendingOperations])
        [NSThread sleepForTimeInterval:0.2];
```

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`.

In the following examples, the Customer and SalesOrder MBOs have a parent-child relationship.

Example 1: Supports `update` operations to parent entities. The sequence of calls is as follows:

```
SampleAppCustomer *customer = [ SampleAppCustomer find: 32]
//find by the unique id
customer.city = @"Dublin"; //update any field to a new value
[customer update];
[customer submitPending];
while ([SUP101SUP101DB hasPendingOperations])
[NSThread sleepForTimeInterval:0.2];
```

Example 2: Supports `update` operations to child entities. The sequence of calls is:

```
SampleAppSales_Order* order = [SampleAppSales_Order find: 1220];
order.region = @"SA"; //update any field
[order update]; //call update on the child record
[order refresh];
[order.customer submitPending]; //call submitPending on the parent
while ([SUP101SUP101DB hasPendingOperations])
[NSThread sleepForTimeInterval:0.2];
```

Example 3: Calling `save()` on a parent also saves any modifications made to its children:

```
SampleAppCustomer *customer = [ SampleAppCustomer find: 32]
SUPObjectList* orderlist = customer.orders;
SampleAppSales_Order* order = [orderlist item:0];
order.sales_rep = @"Ram";
customer.state = @"MA" ;
[customer save];
[customer submitPending];
while ([SUP101SUP101DB hasPendingOperations])
```

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`.

(void)delete

The following examples show how to perform deletes to parent entities and child entities.

Example 1: Supports `delete` operations to parent entities. The sequence of calls is:

```
SampleAppCustomer *customer = [ SampleAppCustomer find: 32]
[Customer delete];
[Customer submitPending];
while ([SUP101SUP101DB hasPendingOperations])
[NSThread sleepForTimeInterval:0.2];
```

Example 2: Supports `delete` operations child entities. The sequence of calls is:

```
SampleAppSales_order *order = [SampleAppSales_order find: 32]
[order delete];
[order.customer submitPending]; //Call submitPending on the parent.
while ([SUP101SUP101DB hasPendingOperations])
```

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.

(void)save

```
SampleAppCustomer *customer = [ SampleAppCustomer find: 32]
//Change some sttribute of the customer record
customer.fname= @"New Name";
[customer 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.

This is an example of an "other" operation:

```
SampleAppCustomerOtherOperation *other =
[[SampleAppCustomerOtherOperation alloc] init];
other.P1 = @"somevalue";
other.P2 = 2;
other.P3 = [NSDate date];
[other save];
[other submitPending];
```

Pending Operation

You can manage the pending state.

- **`(void)cancelPending`** – Cancels a pending record. A pending record is one that has been updated in the local client database, but not yet sent to the Unwired Server.

  ```
  [customer cancelPending];
  ```
- **`(void)cancelPendingOperations`** – Cancels the pending operations for an entire entity. This method internally invokes the `cancelPending` method.

  ```
  [Customer cancelPendingOperations];
  ```
- **`(void)submitPending`** – Submits a pending record to the Unwired Server. For MBS, a replay request is sent directly to the Unwired Server.

```
[customer submitPending];
```

- **+(void)submitPendingOperations** – Submits all data for all pending records to the Unwired Server. This method internally invokes the `submitPending` method.

```
[Customer submitPendingOperations];
```

- **+(void)submitPendingOperations: (NSString*)synchronizationGroup** – Submits all data for pending records from MBOs in this synchronization group to the Unwired Server. This method internally invokes the `submitPending` method.

```
[SampleAppSUP101DB submitPendingOperations:@"default"];
```

```
SampleAppCustomer *customer = [SUP101Customer find:101];
//Make some changes to the customer record.
//Save the changes

//If the user  wishes to cancel the changes, a call to cancel pending
will revert to the old values.

[customer cancelPending];

// The user can submit the changes to the server as follows:
[customer submitPending];
```

Date/Time

Classes that support managing date/time objects.

- **SUPDateValue.h** – manages an object of datatype `Date`.
- **SUPTimeValue.h** – manages an object of datatype `Time`.
- **SUPDateTimeValue.h** – manages an object of datatype `DateTime`.
- **SUPDateList.h** – manages a list of `Date` objects (the objects cannot be null).
- **SUPTimeList.h** – manages a list of `Time` objects (the objects cannot be null).
- **SUPDateTimeList.h** – manages a list of `DateTime` objects (the objects cannot be null).
- **SUPNullableDateList.h** – manages a list of `Date` objects (the objects can be null).
- **SUPNullableTimeList.h** – manages a list of `Time` objects (the objects can be null).
- **SUPNullableDateTimeList.h** – manages a list of `DateTime` objects (the objects can be null).

Example 1: To get a `Date` value from a query result set:

```
SUPQueryResultSet* resultSet = [SUP101SUP101DB executeQuery:query];
 for(SUPDataValueList* result in resultSet)
          [[SUPDataValue getNullableDate:[result item:2]]
description];
```

Example 2: A method takes `Date` as a parameter:

```
-(void)setModifiedOrderDate:(SUPDateValue*) thedate;
SUPDateValue *thedatevalue = [SUPDateValue newInstance];
[thedatevalue setValue:[NSDate date]];
[customer setModifiedOrderDate:thedatevalue];
```

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

Name	Type	Description
`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 derived from the time in seconds since an epoch, and increases each time a row is changed. `int64_t result = [customer replayCounter];`
`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`). `int64_t result = [customer 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. `int64_t result = [customer replayFailure];`

Entity State Example

Shows how the values of the entities that support pending state change at different stages during the MBO update process. The values that change between different states appear in bold.

Note these entity behaviors:

- The `isDirty` flag is set if the entity changes in memory but is not yet written to the database. Once you save the MBO, this flag clears.

- The `replayCounter` value that gets sent to the Unwired Server is the value in the database before you call `submitPending`. After a successful replay, that value is imported from the Unwired Server.
- The last two entries in the table are two possible results from the operation; only one of these results can occur for a replay request.

Description	Flags/Values
After reading from the database, before any changes are made.	isNew=false isCreated=false isDirty=false isDeleted=false isUpdated=false pending=false pendingChange='N' replayCounter=33422977 replayPending=0 replayFailure=0
One or more attributes are changed, but changes not saved.	isNew=false isCreated=false isDirty=**true** isDeleted=false isUpdated=false pending=false pendingChange='N' replayCounter=33422977 replayPending=0 replayFailure=0

Description	Flags/Values
After `[entity save]` or `[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]` is called to submit the MBO to the server.	isNew=false isCreated=false isDirty=false isDeleted=false isUpdated=true pending=true pendingChange='U' replayCounter=33424981 replayPending=**33424981** replayFailure=0

Description	Flags/Values
Possible result: the Unwired Server accepts the update, sends an import and a `replayResult` for the entity, and then refreshes the entity from the database.	isNew=false isCreated=false isDirty=false isDeleted=false isUpdated=**false** pending=**false** pendingChange=**'N'** replayCounter=**33422977** replayPending=**0** replayFailure=0
Possible result: The Unwired Server rejects the update, sends a `replayFailure` for the entity, and refreshes the entity from the database	isNew=false isCreated=false isDirty=false isDeleted=false isUpdated=true pending=true pendingChange='U' replayCounter=33424981 replayPending=**0** replayFailure=**33424981**

Refresh Operation

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

For example:

(void)refresh

```
[order refresh];
```

where `order` is an instance of the MBO entity.

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.

```
+ (void)createDatabase;
+ (void)deleteDatabase;
```

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.

```
SUP101Customer *customer1 = [SUP101Customer findByPrimaryKey:101];
SUP101Customer *customer2 = [SUP101Customer findByPrimaryKey:102];

// Use one transaction for better performance with multiple changes
SUPLocalTransaction *tx = [SUP101SUP101DB beginTransaction];
[customer1 save];
[customer2 save];
// Commit the transaction
[tx commit];
// Submit the changes to the server
[customer1 submitPending];
[customer2 submitPending];
```

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.

SUPBigBinary

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

```
- (void)close;
```

Examples

- – Writes a binary book cover image and closes the image file.

```
SUPBigBinary *image = book.cover;
NSData * data;

[image openForWrite:[data length]];
[image write:data];
[image close];
```

copyFromFile
Overwrites this `SUPBigBinary` object with data from the specified file.

Any previous contents of the file will be discarded. Throws an `ObjectNotSavedException` if this `SUPBigBinary` 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

```
- (void)copyFromFile :(SUPString)filepath;
```

Parameters

- **filepath –** The file containing the data to be copied.

copyToFile
Overwrites the specified file with the contents of this `SUPBigBinary` object.

Any previous contents of the file are discarded. Throws an `ObjectNotSavedException` if this `SUPBigBinary` 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

```
- (void)copyToFile :(SUPString)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

```
- (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 `SUPBigBinary` 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

```
- (void)openForRead;
```

Examples

- – Opens a binary book image for reading.

```
SUPBigBinary *image = book.cover;
[image openForRead];
```

openForWrite
Opens the value stream for writing.

Any previous contents of the value will be discarded. Throws an `ObjectNotSavedException` if this `SUPBigBinary` object is an attribute of an entity that has not yet been created in the database.

Syntax

```
- (void)openForWrite :(SUPLong)newLength;
```

Parameters

- **newLength –** The new value length in bytes. Some platforms may allow this parameter to be specified as 0, with the actual length to be determined later, depending on the amount of

data written to the stream. Other platforms require the total amount of data written to the stream to match the specified value.

Examples

- – Opens a binary book image for writing.

```
SUPBigBinary *image = book.cover;
[image openForWrite:[data length]];
```

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

```
- (SUPNullableBinary)read :(SUPLong)length;
```

Parameters

- **length –** The maximum number of bytes to be read into the chunk.

Returns

`read` returns a chunk of binary data read from the stream, or a null value if the end of the stream has been reached.

Examples

- – Reads in a binary book image.

```
SUPSampleBook *book = [SUPSampleBook findByPrimaryKey:bookID];
SUPBigBinary *image = book.cover;
int bufferLength2 = 1024;
[image openForRead];
NSData *data = [image read:bufferLength];
```

readByte

Reads a single byte from the stream.

Throws a `StreamNotOpenException` if the stream is not open for reading.

Syntax

```
- (SUPInt)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

```
- (void)seek :(SUPLong)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

```
- (void)write :(SUPBinary)data;
```

Parameters

- **data –** The data chunk to be written to the stream.

Examples

- – Opens a binary book image for writing.

```
SUPSampleBook *book = [SUPSampleBook findByPrimaryKey:bookID];

SUPBigBinary *image = book.cover;
NSData * data;

[image openForWrite:[data length]];
[image write:data];
```

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

```
- (void)writeByte :(SUPByte)data;
```

Parameters

- **data –** The byte value to be written to the stream.

SUPBigString

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

```
- (void)close;
```

Examples

- – Writes to the biography file, and closes the file.

```
SUPSampleAuthor * author = [SUPSampleAuthor
findByPrimaryKey:authorID];

SUPBigString *text = author.biography;

NSString *stringToWrite = @"something";

[text openForWrite:[stringToWrite length]];
[text write:stringToWrite];
[text close];
```

copyFromFile
Overwrites this `SUPBigString` object with data from the specified file.

Any previous contents of the value will be discarded. Throws an `ObjectNotSavedException` if this `SUPBigString` 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

```
- (void)copyFromFile :(SUPString)filepath;
```

Parameters

- **filepath –** The file containing the data to be copied.

copyToFile
Overwrites the specified file with the contents of this `SUPBigString` object.

Any previous contents of the file are discarded. Throws an `ObjectNotSavedException` if this `SUPBigString` 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

```
- (void)copyToFile :(SUPString)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

```
- (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 `SUPBigString` object is an attribute of an entity that has not yet been created in the database.

Syntax

```
- (void)openForRead;
```

Examples

- – Opens the biography file for reading.

```
SUPSampleAuthor * author = [SUPSampleAuthor
findByPrimaryKey:authorID];

SUPBigString *text = author.biography;
[text openForRead];
```

openForWrite

Opens the value stream for writing.

Any previous contents of the value will be discarded. Throws an `ObjectNotSavedException` if this `SUPBigString` object is an attribute of an entity that has not yet been created in the database.

Syntax

```
- (void)openForWrite :(SUPLong)newLength;
```

Parameters

- **newLength –** The new value length in bytes. Some platforms may allow this parameter to be specified as 0, with the actual length to be determined later, depending on the amount of data written to the stream. Other platforms require the total amount of data written to the stream to match the specified value.

Examples

- – Opens the biography file for writing.

```
SUPSampleAuthor * author = [SUPSampleAuthor
findByPrimaryKey:authorID];

SUPBigString *text = author.biography;

NSString *stringToWrite = @"something";

[text openForWrite:[stringToWrite length]];
```

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

```
- (SUPNullableBinary)read :(SUPLong)length;
```

Parameters

- **length –** The maximum number of characters to be read into the chunk.

Returns

`read` returns a chunk of string data read from the stream, or a null value if the end of the stream has been reached.

Examples

- – Reads in the biography file.

```
int64_t bufferLength = 1024;
NSString *something = [text read:bufferLength]; // null if EOF
while (something != nil)
{
   something = [text read:bufferLength];
}
```

readChar

Reads a single character from the stream.

Throws a `StreamNotOpenException` if the stream is not open for reading.

Syntax

```
- (SUPInt)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

```
- (void)seek :(SUPLong)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

```
- (void)write :(SUPString)data;
```

Parameters

- **data –** The data chunk to be written to the stream.

Examples

- – Writes to the biography file, and closes the file.

```
SUPSampleAuthor * author = [SUPSampleAuthor
findByPrimaryKey:authorID];

SUPBigString *text = author.biography;

NSString *stringToWrite = @"something";

[text openForWrite:[stringToWrite length]];
[text write:stringToWrite];
```

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

```
- (void)writeChar :(SUPChar)data;
```

Parameters

- **data –** The character value to be written to the stream.

MetaData and Object Manager API

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

MetaData and Object Manager API

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

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

You can generate metadata classes using the `-md` code generation option. You can use the `-rm` option to generate the object manager class. You can also generate metadata classes by selecting the option **Generate metadata classes** or **Generate metadata and object manager classes** option in the code generation wizard in the mobile application project.

SUPDatabaseMetaData

The `SUPDatabaseMetaData` class holds package-level metadata. You can use it to retrieve information about all the classes and entities for which metadata has been generated.

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.

SUPClassMetaData

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

```
NSLog(@"List classes that have metadata....");
SUPDatabaseMetaData *dmd = [SUP101SUP101DB metaData];
SUPObjectList *classes = dmd.classList;
for(SUPClassMetaData *cmd in classes)
{
  NSLog(@" Class name = %@:",cmd.name);
}
NSLog(@"List entities that have metadata, and their attributes
```

```
and operations....");
SUPObjectList *entities = dmd.entityList;
for(SUPEntityMetaData *emd in entities)
{
  NSLog(@" Entity name = %@, database table name =
      %@:",emd.name,emd.table);
  SUPObjectList *attributes = emd.attributes;
  for(SUPAttributeMetaData *amd in attributes)
      NSLog(@" Attribute: name = %@%@",amd.name,
          (amd.column ? [NSString stringWithFormat:@",
          database column = %@",amd.column] : @""));
  SUPObjectList *operations = emd.operations;
  for(SUPOperationMetaData *omd in operations)
  {
    NSLog(@" Operation: name = %@",omd.name);
    SUPObjectList *parameters = omd.parameters;
    for(SUPParameterMetaData *pmd in parameters)
       NSLog(@" Parameter: name = %@, type = %@",
           pmd.name, [pmd.dataType name]);
  }
}
```

SUPAttributeMetaData

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

Exceptions

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

Handling Exceptions

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

Server-Side Exceptions

A server-side exception occurs when a client tries to update or create a record and the Unwired Server throws an exception.

A server-side exception results in a stack trace in the server log, and a log record (`LogRecordImpl`) imported to the client with information on the problem. The client receives both the log record and a `replayFailed` message.

HTTP Error Codes

Unwired Server examines the EIS code received in a server response message and maps it to a logical HTTP error code, if a corresponding error code exists. If no corresponding code exists,

the 500 code is assigned to signify either a Sybase Unwired Platform internal error, or an unrecognized EIS error. The EIS code and HTTP error code values are stored in log records.

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

Table 4. Recoverable Error Codes

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

Table 5. Unrecoverable Error Codes

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

Error code 401 is not treated as a simple recoverable error. If the `SupThrowCredentialRequestOn401Error` context variable is set to true (the default), error code 401 throws a `CredentialRequestException`, which sends a credential request notification to the user's inbox. You can change this behavior by modifying the value of the `SupThrowCredentialRequestOn401Error` context variable in Sybase Control Center. If `SupThrowCredentialRequestOn401Error` is set to false, error code 401 is treated as a normal recoverable exception.

Mapping of EIS Codes to Logical HTTP Error Codes

A list of SAP® error codes mapped to HTTP error codes. By default, SAP error codes that are not listed map to HTTP error code 500.

Note: These JCO error codes are not applicable for DOE-based applications.

Table 6. Mapping of SAP Error Codes to HTTP Error Codes

Constant	Description	HTTP Error Code
JCO_ERROR_COMMUNICATION	Exception caused by network problems, such as connection breakdowns, gateway problems, or unavailability of the remote SAP system.	503
JCO_ERROR_LOGON_FAILURE	Authorization failures during login. Usually caused by unknown user name, wrong password, or invalid certificates.	401
JCO_ERROR_RESOURCE	Indicates that JCO has run out of resources such as connections in a connection pool.	503
JCO_ERROR_STATE_BUSY	The remote SAP system is busy. Try again later.	503

Client-Side Exceptions

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

Note: See *Callback Handlers*.

Attribute Datatype Conversion

When a non-nullable attribute's datatype is converted to a non-primitive datatype (such as class NSNumber, NSDate, and so on), you must verify that the the corresponding property for the MBO instance is assigned a non-nil value, otherwise the application may receive a runtime exception when creating a new MBO instance.

A typical scenario is when an attribute exists in ASE's identity column with a numeric datatype. For example, for a non-nullable attribute with a decimal datatype, the corresponding datatype in the generated Objective-C MBO code is `NSNumber`. When creating a new MBO instance, ensure that you assign this property a non-nil value.

Operation Name Conflicts

Operation names that conflict with special field types are not handled.

For example, if an MBO has attributes named id and description, those attributes are stored with the name `id_ description_`. If you create an operation called "description" and generate Object-C code, you see an error during compilation because of conflicting methods in the classes.

Exception Classes

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

- **SynchronizeException –** thrown when an error occurs during synchronization.
- **PersistenceException –** thrown when trying to access the local database.
- **ObjectNotFoundException –** thrown when trying to load an MBO that is not inside the local database.
- **NoSuchOperationException –** thrown when trying to call a method (using the Object Manager API) but the method is not defined for the MBO.
- **NoSuchAttributeException –** thrown when trying to access an attribute (using the Object Manager API) but the attribute is not defined for the MBO.
- **ApplicationRuntimeException –** thrown when a call to start the connection, register the application, or unregister the application cannot be completed due to an error.
- **ConnectionPropertyException –** thrown when a call to start the connection, register the application, or unregister the application cannot be completed due to an error in a connection property value or application identifier.

Query Exception Classes

Exceptions thrown by `SUPStatementBuilder` when building an `SUPQuery`, or by `SUPQueryResultSet` during processing of the results. These exceptions occur if the query called for an entity or attribute that does not exist, or tried to access results with the wrong datatype.

- **SUPAbstractClassException.h –** thrown when the query specifies an abstract class.
- **SUPInvalidDataTypeException.h –** thrown when the query tries to access results with an invalid datatype.
- **SUPNoSuchAttributeException.h –** thrown when the query calls for an atttribute that does not exist.
- **SUPNoSuchClassException.h –** thrown when the query calls for a class that does not exist.
- **SUPNoSuchParameterException.h –** thrown when the query calls for a parameter that does not exist.
- **SUPNoSuchOperationException.h –** thrown when the query calls for an operation that does not exist.

- **SUPWrongDataTypeException.h –** thrown when the query tries to access results with an incorrect datatype definition.

Messaging Client API Exception Classes

Exceptions in the messaging client (`clientrt`) library.

- **SUPObjectNotFoundException.h –** thrown by the `load:` method for entities if the passed-in primary key is not found in the entity table.
- **SUPPersistenceException.h –** may be thrown by methods that access the database. This may occur when application codes attempts to:
  - Insert a new row in an MBO table using a duplicate key value.
  - Execute a dynamic query that selects for attribute (column) names that do not exist in an MBO.

Index

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

X