SYBASE®
An SAP® Company

Developer Guide: iOS Object API Applications

Sybase Unwired Platform 2.2 SP04

DOCUMENT ID: DC01217-01-0224-01
LAST REVISED: May 2013
Copyright © 2013 by Sybase, Inc. All rights reserved.
This publication pertains to Sybase software and to any subsequent release until otherwise indicated in new editions or technical notes. Information in this document is subject to change without notice. The software described herein is furnished under a license agreement, and it may be used or copied only in accordance with the terms of that agreement.
Upgrades are provided only at regularly scheduled software release dates. No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of Sybase, Inc.
Sybase trademarks can be viewed at the Sybase trademarks page at *http://www.sybase.com/detail?id=1011207*. Sybase and the marks listed are trademarks of Sybase, Inc. ® indicates registration in the United States of America.
SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world.
Java and all Java-based marks are trademarks or registered trademarks of Oracle and/or its affiliates in the U.S. and other countries.
Unicode and the Unicode Logo are registered trademarks of Unicode, Inc.
All other company and product names mentioned may be trademarks of the respective companies with which they are associated.
Use, duplication, or disclosure by the government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of DFARS 52.227-7013 for the DOD and as set forth in FAR 52.227-19(a)-(d) for civilian agencies.
Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

Contents

Getting Started with iOS Development 1
Object API Applications 1
Best Uses for Object API Applications 2
Cache Synchronization 2
Client Runtime Architecture 3
Documentation Roadmap for Sybase Unwired Platform 4
Development Task Flow for Object API Applications 5
Installing the iOS Development Environment 6
Downloading the Xcode IDE 6
Downloading Older Versions of the Xcode IDE 6
Installing X.509 Certificates on iOS Clients 6
Generating Objective-C Object API Code 6
Generating Objective-C Object API Code Using Sybase Unwired WorkSpace 7
Generating Object API Code Using the Code Generation Utility 11
Generated Code Location and Contents 12
Validating Generated Code 13
Creating a Project 13
Generating HeaderDoc from Generated Code 13
Downloading the Latest Afaria Libraries 14
Importing Libraries and Code 14
Importing Libraries and Code for Applications Enabled with ARC 18
Managing the Background State 21
Development Task Flow for DOE-based Object API Applications 23
Installing the iOS Development Environment 24
Downloading the Xcode IDE 24
Downloading Older Versions of the Xcode IDE 24

Installing X.509 Certificates on iOS Clients 24
Generating Objective-C Object API Code 24
Generated Code Location and Contents 25
Creating a Project 26
Generating HeaderDoc from Generated Code 26
Downloading the Latest Afaria Libraries 26
Importing Libraries and Code 26
Importing Libraries and Code for Applications Enabled with ARC 30
Managing the Background State 33
Developing the Application Using the Object API 35
Initializing an Application 35
Initially Starting an Application 35
Subsequently Starting an Application 43
Accessing MBO Data 43
Object Queries 43
Dynamic Queries 44
MBOs with Complex Types 44
Relationships 45
Manipulating Data 46
Creating, Updating, and Deleting MBO Records 46
Other Operations 47
Using SubmitPending and SubmitPendingOperations 47
Shutting Down the Application 48
Closing Connections 48
Debugging Runtime Errors and Performance Analysis 49
End to End Tracing 49
Tracking KPI 51
Uninstalling the Application 52
Deleting the Database and Unregistering the Application 52
Testing Applications 55

Testing an Application Using a Emulator 55
Client-Side Debugging 55
Server-Side Debugging 57
Localizing Applications 59
Localizing Menus and Interfaces 59
Localizing Embedded Strings 60
Validating Localization Changes 60
Packaging Applications 61
Signing 61
Apple Push Notification Service Configuration 61
Preparing an Application for Apple Push Notification Service 61
Provisioning an Application for Apple Push Notification Service 63
Preparing Applications for Deployment to the Enterprise 64
Client Object API Usage 67
Client Object API Reference 67
Application APIs 67
Application 67
ConnectionProperties 79
ApplicationSettings 83
ConnectionPropertyType 88
Afaria APIs 94
Using Afaria to Provision Configuration Data 94
Using Certificates from Afaria for Authentication 96
Connection APIs 104
SUPConnectionProfile 104
Set Database File Property 106
Synchronization Profile 106
Connect the Data Synchronization Channel Through a Relay Server 107
Authentication APIs 108
Logging In 108

Importing an X.509 Certificate to an iOS Client from the Unwired Server 108
Sample Code 109
Single Sign-On With X.509 Certificate Related Object API 112
Personalization APIs 114
Type of Personalization Keys 114
Getting and Setting Personalization Key Values 115
Synchronization APIs 115
Managing Synchronization Parameters 115
Performing Mobile Business Object Synchronization 116
Message-Based Synchronization APIs 116
Push Synchronization Applications 122
Retrieving Information about Synchronization Groups 123
Log Record APIs 123
SUPLogRecord API 123
Logger APIs 126
Log Level and Tracing APIs 126
Change Log API 127
entityType 127
operationType 128
rootEntityType 129
rootSurrogateKey 129
surrogateKey 130
Methods in the Generated Database Class 130
Code Samples 132
Security APIs 133
Encrypting the Client Database 133
Accessing a Previously Encrypted Database 134
SUPDataVault 134
Callback and Listener APIs 151
Callback Handler API 151

SUPApplicationCallback API 154
Apple Push Notification API 157
Query APIs 159
Retrieving Data from Mobile Business Objects . 159
Retrieving Relationship Data 167
Persistence APIs 168
Operations APIs 168
Object State APIs 172
Generated Package Database APIs 177
Large Attribute APIs 178
MetaData API 187
MetaData API 188
SUPDatabaseMetaData 188
SUPClassMetaData 188
EntityMetaData 189
SUPAttributeMetaData 189
Exceptions 189
Exception Handling 189
Exception Classes 195
Error Codes 196
Index 199

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*
- *Supported Hardware and Software*
- *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*.
- The iOS HeaderDoc provides a complete reference to the APIs:
  - The Framework Library HeaderDoc is installed to *SUP_HOME* `\MobileSDK22\ObjectAPI\iOS\headerdoc`. For example, `C:\Sybase\UnwiredPlatform\MobileSDK22\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.
- *Developer Guide: Migrating to Sybase Mobile SDK* contains information for developers who are migrating device applications to a newer software version, and changes to MBOs, projects, and the Unwired Server.

Object API Applications

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

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

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

Development involves both server-side and client-side components. 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 packages, that are deployed to the 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

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

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

  The messaging channel sends these types of communications:
  - Application registration - the messaging channel is used for application registration before establishing a connection to the Unwired Server.
  - Change notifications - when the Unwired Server detects changes in the back-end EIS, the 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.

  - Operation replay records - when synchronizing, these 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.
  - SAP Data Orchestration Engine (DOE) application synchronization - the messaging channel is used for synchronization for DOE applications.
- The synchronization channel sends data to keep the 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 EIS. Data that changes on the EIS as a result of device changes, or those originating elsewhere, are replicated to the device database.

Documentation Roadmap for Sybase 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.

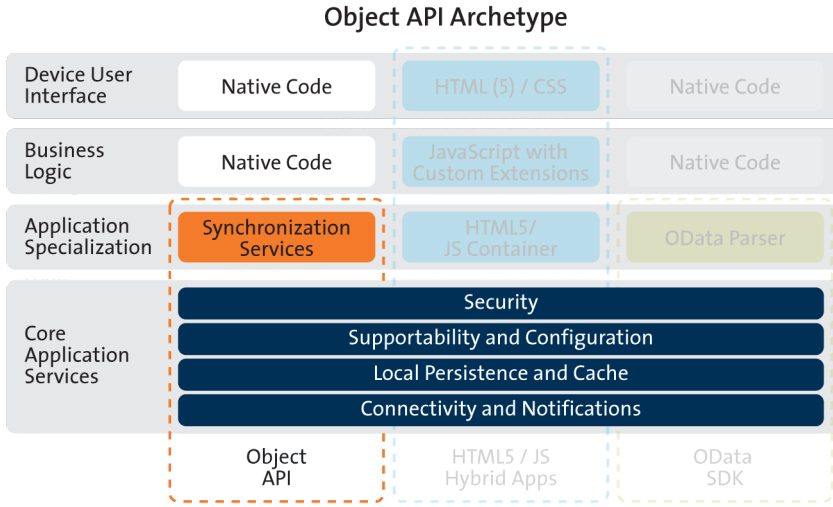
See *Documentation Roadmap* in *Fundamentals* for document descriptions by user role.

Check the Sybase® Product Documentation Web site regularly for updates: *http://sybooks.sybase.com/sybooks/sybooks.xhtml?id=1289&c=firsttab&a=0&p=categories* , then navigate to the most current version.

Development Task Flow for Object API Applications

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

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



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

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

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. 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.

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

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

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

   Sybase Unwired WorkSpace automatically computes the default page size after you choose the MBOs based on total attribute size. If an MBO's accumulated attribute size is larger than the page size setting, a warning displays.
5. Enter the information for these configuration options:

Option	Description
Language	Select **Objective C**.
Platform	Select the platform (target device) for which the device client code is intended. • Objective C • iOS
Unwired Server	Specify a default Unwired Server connection profile to which the generated code connects at runtime.
Server domain	Choose the domain to which the generated code will connect. If you specified an Unwired Server to which you previously connected successfully, the first domain in the list is chosen by default. You can enter a different domain manually. **Note:** This field is only enabled when an Unwired Server is selected.

Option	Description
Page size	(Optional) Select the page size for the generated client code. If the page size is not set, the default page size is 4KB at runtime. The default is a proposed page size based on the selected MBO's attributes. . The page size should be larger than the sum of all attribute lengths for any MBO that is included with all the MBOs selected, and must be valid for the database. If the page size is changed, but does not meet these guidelines, object queries that use string or binary attributes with a **WHERE** clause may fail. A binary length greater than 32767 is converted to a binary large object (BLOB), and is not included in the sum; a string greater than 8191 is converted to a character large object (CLOB), and is also not included). If an MBO attribute's length sum is greater than the page size, some attributes automatically convert to BLOB or CLOB, and therefore cannot be put into a **WHERE** clause. **Note:** This field is only enabled when an Unwired Server is selected.
Destination	Specify the destination of the generated device client files. Enter (or **Browse**) to either a **Project path** (Mobile Application project) location or **File system path** location. Select **Clean up destination before code generation** to clean up the destination folder before generating the device client files.

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

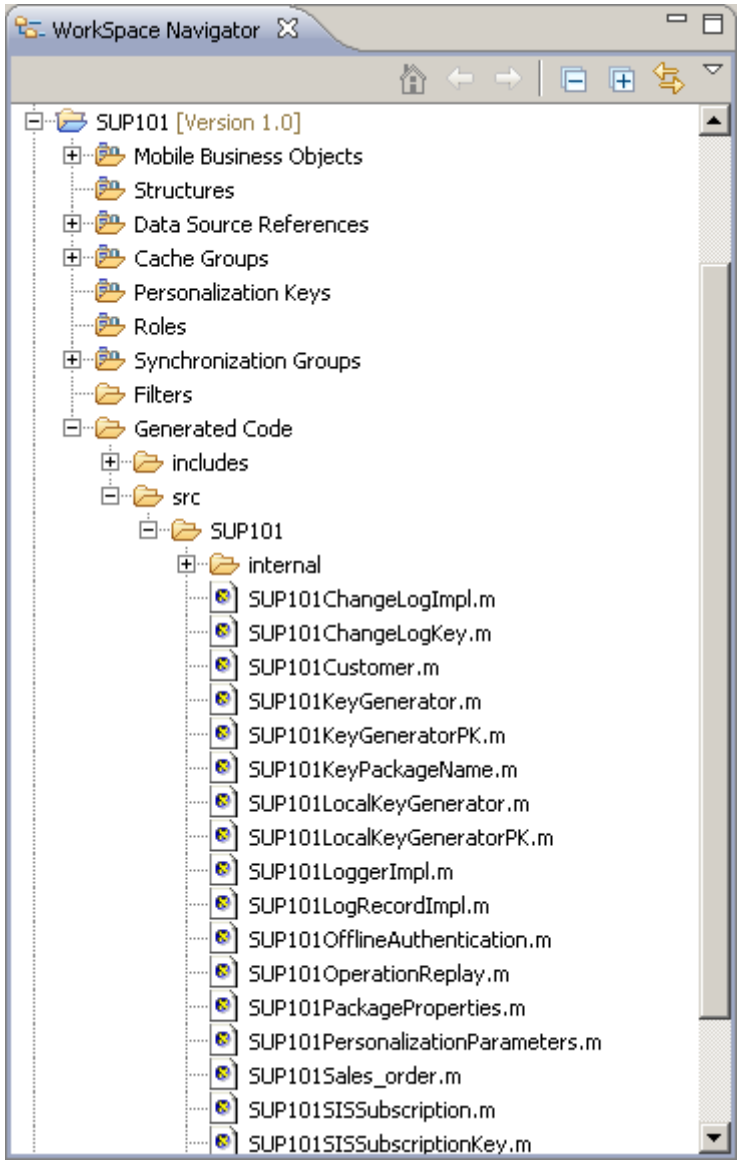
   The **Including object manager classes** option is enabled only for BlackBerry and C# if you select **Generate metadata classes**. The object manager allows you to retrieve the metadata of packages, MBOs, attributes, operations, and parameters during runtime using the name instead of the object instance.

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

7. 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, "SUP101".



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, `SUP101Customer.h`, `SUP101Customer.m`)	Include all the attributes, operations, object queries, and so on, defined in this MBO.
synchronization parameter class (for example, `SUP101CustomerSynchronizationParameter.h`, `SUP101CustomerSynchronizationParameter.m`)	Include any synchronization parameters defined in this MBO.
Key generator classes (for example, `SUP101KeyGenerator.h`, `SUP101KeyGenerator.m`)	Include generation of surrogate keys used to identify and track MBO instances and data.
Personalization parameter classes (for example, `SUP101PersonalizationParameters.h`, `SUP101PersonalizationParameters.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.

8. Examine the generated code location and contents.
9. 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 Sybase 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 the 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 Sybase 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 *`SUP_HOME`*`\MobileSDK22\ObjectAPI\Utils\bin`, run the `codegen.bat` utility, specifying the following parameters:

   ```
   codegen.bat -oc -client -ul -mdp 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 *`SUP_HOME`*`\MobileSDK22\ObjectAPI \Utils\genfiles` directory, assuming `codegen.bat` is run from the *`SUP_HOME`*`\MobileSDK22\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 Sybase Unwired WorkSpace, generated object API code is stored by default in the "Destination" location you specified during code generation. If you generated code with the Code Generation Utility, generateed object API code is stored in the *`SUP_HOME`* `\MobileSDK22\ObjectAPI\Utils\genfiles` folder after you you generate code.

The contents of the folder is determined by the options you selected in the Generate Code wizard in Sybase 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.
- DatabaseClass – package level class that handles subscription, login, synchronization, and other operations for the package.
- Synchronization parameters – any synchronization parameters for the MBOs.
- Personalization parameters – personalization parameters used by the package.
- Metadata – Metadata class that allow you to query meta data including MBOs, their attributes, and operations, in a persistent table at runtime.

Validating Generated Code

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

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.

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*.

Downloading the Latest Afaria Libraries

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

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

Importing Libraries and Code

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

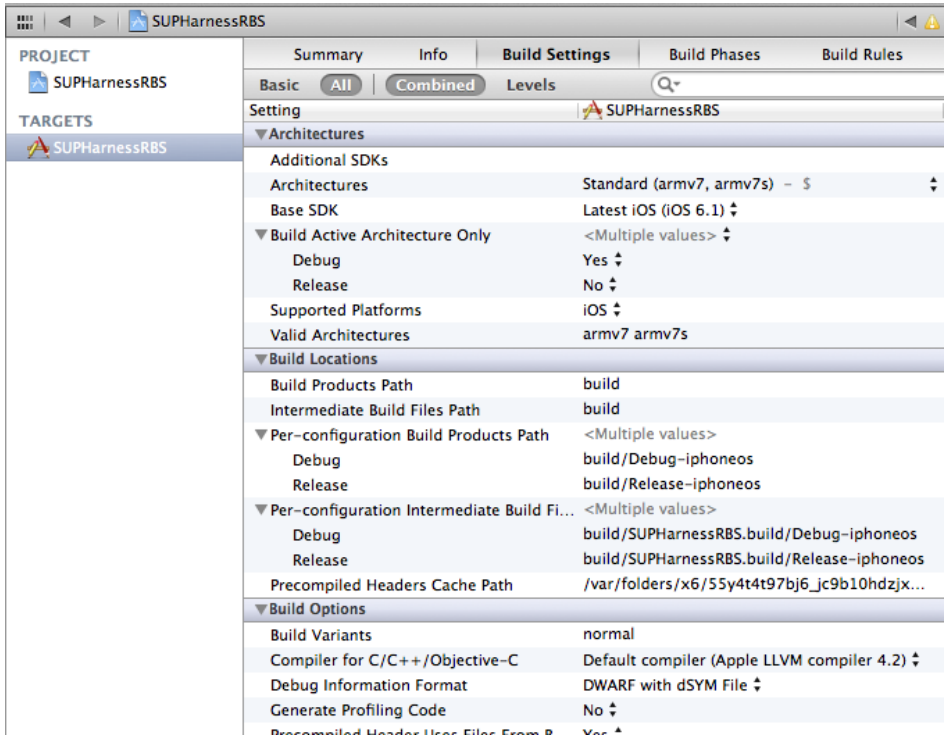
Note: For more information on Xcode, refer to the Apple Developer Connection: *http://developer.apple.com/tools/Xcode/*.

1. Start Xcode 4.6 and select **Create a new Xcode project**.

   At the time of this writing, Xcode 4.6 was the latest version. It is possible that you have a later version.
2. Select **iOS Application** and select an appropriate project template, and then click **Next**.
3. Enter *<ProjectName>* as the **Product Name**, *<Company Identifier>* 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 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.
5. Select the **Architectures** section under Build Settings, and set Base SDK for All Configurations to `iOS 6.1`.



6. Select the Valid Architectures with the default value `armv7 armv7s`, Supported Platforms as `iOS`, and the Targeted device family as `iPhone/iPad`. This ensures that the build of the application can run on either iPhone or iPad.
7. Scroll to the **Deployment** section and set the iOS Deployment Target as appropriate for the device version where you will deploy. The minimum version is `iOS 4.3` or later. Earlier SDKs and deployment targets are not supported.
8. Copy the files from your Windows machine to the <ProjectName> folder that Xcode created to contain the generated source code. 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 *SUP_HOME*`\MobileSDK<version>\ObjectAPI\iOS` 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. 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`, `libPerformanceLib.a`, `libsupClientUtil.a`, `libSUPSupportability.a`, `libAfariaSLL.a`, `libDatavault.a`, and `libsupUltralite.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 `Libraries/Debug-iphonesimulator/` to add the libraries.

    As an alternative to adding static libraries to the project, you can configure your project to specify the libraries in the project's build settings:

    - Select the project from the Project Navigator.
    - Click on the target under Targets and select **Build Settings**.
    - In the Linking section, expand **Other Linker Flags**.
    - Under Debug, add the following linker flags:

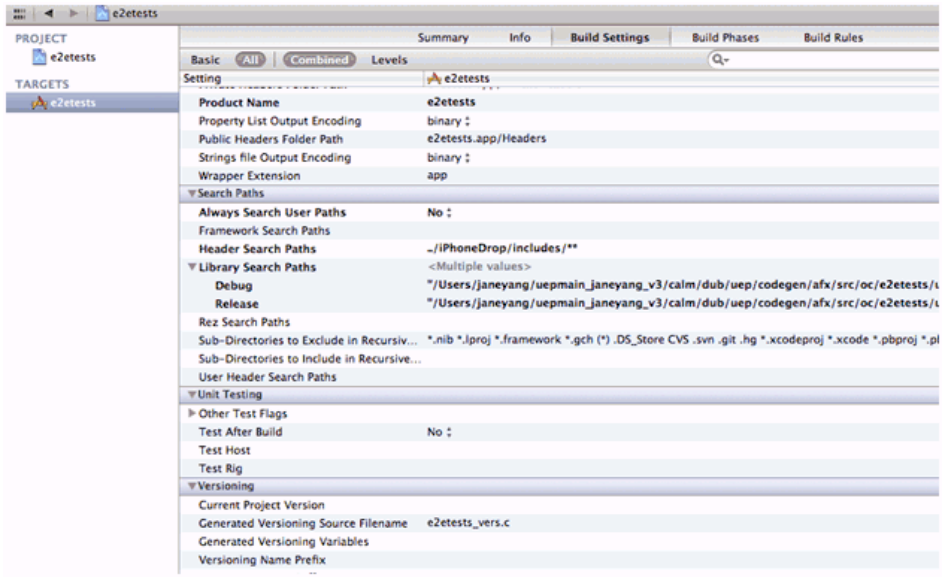
      ```
      $(SRCROOT)/$(PRODUCT_NAME)/Libraries/$(CONFIGURATION)$
      (EFFECTIVE_PLATFORM_NAME)/libMo.a
      $(SRCROOT)/$(PRODUCT_NAME)/Libraries/$(CONFIGURATION)$
      (EFFECTIVE_PLATFORM_NAME)/libSUPObj.a
      $(SRCROOT)/$(PRODUCT_NAME)/Libraries/$(CONFIGURATION)$
      (EFFECTIVE_PLATFORM_NAME)/libclientrt.a
      $(SRCROOT)/$(PRODUCT_NAME)/Libraries/$(CONFIGURATION)$
      (EFFECTIVE_PLATFORM_NAME)/libPerformanceLib.a
      $(SRCROOT)/$(PRODUCT_NAME)/Libraries/$(CONFIGURATION)$
      (EFFECTIVE_PLATFORM_NAME)/libsupClientUtil.a
      $(SRCROOT)/$(PRODUCT_NAME)/Libraries/$(CONFIGURATION)$
      (EFFECTIVE_PLATFORM_NAME)/libSUPSupportability.a
      $(SRCROOT)/$(PRODUCT_NAME)/Libraries/$(CONFIGURATION)$
      (EFFECTIVE_PLATFORM_NAME)/libAfariaSLL.a
      $(SRCROOT)/$(PRODUCT_NAME)/Libraries/$(CONFIGURATION)$
      (EFFECTIVE_PLATFORM_NAME)/libDatavault.a
      $(SRCROOT)/$(PRODUCT_NAME)/Libraries/$(CONFIGURATION)$
      (EFFECTIVE_PLATFORM_NAME)/libsupUltralite.a
      ```

      These linker flags resolve for all builds of the project.

12. Click the project root, in the middle pane click the **<ProjectName>** project, and set Objective-C Automatic Reference Counting in the Apple LLVM compiler 4.1 - Language section to No.
13. Click on the active target and modify the Library Search Path from the Building Settings. For example:

```
$(SRCROOT)/../iOS/Libraries/$(CONFIGURATION)$
(EFFECTIVE_PLATFORM_NAME)
```

Enter the path to the location where you copied the libraries. Specify separate profiles for debug and release, and specify "any iOS" and "any iOS simulator." Ensure that you escape the paths using double quotes.



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

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

15. 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:

    - CoreFoundation.framework
    - Security.framework
    - CFNetwork.framework
    - SystemConfiguration.framework
    - MobileCoreServices.framework
    - libicucore.A.dylib
    - libstdc++.dylib
    - libz.dylib

16. Hold the Option key, and select **Product > Clean Build Folder** 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.

17. Click on the active target, select the **Info** tab, change the "Application requires iPhone environment" setting to "Application does not run in background," and set to YES.

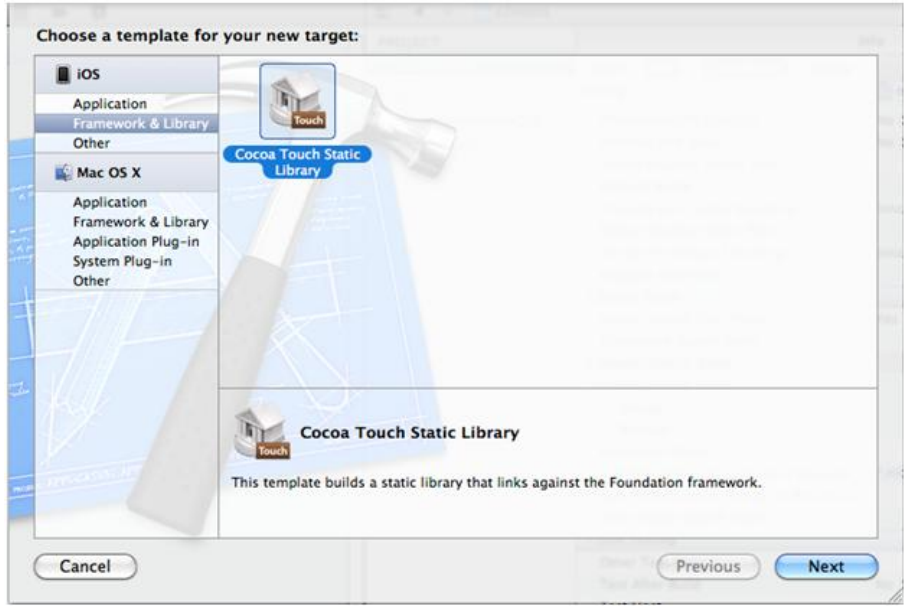
    Note: If you want to allow your application to continue to run safely in the background, do not perform this step. See *Developer Guide: iOS Object API Applications* > *Development Task Flow for Object API Applications* > *Creating a Project* > *Managing the Background State*.

18. Write your application code to reference the generated MBO code. See the *Developer Guide: iOS Object API Applications* for information about referencing the iOS Client Object API.

Importing Libraries and Code for Applications Enabled with ARC

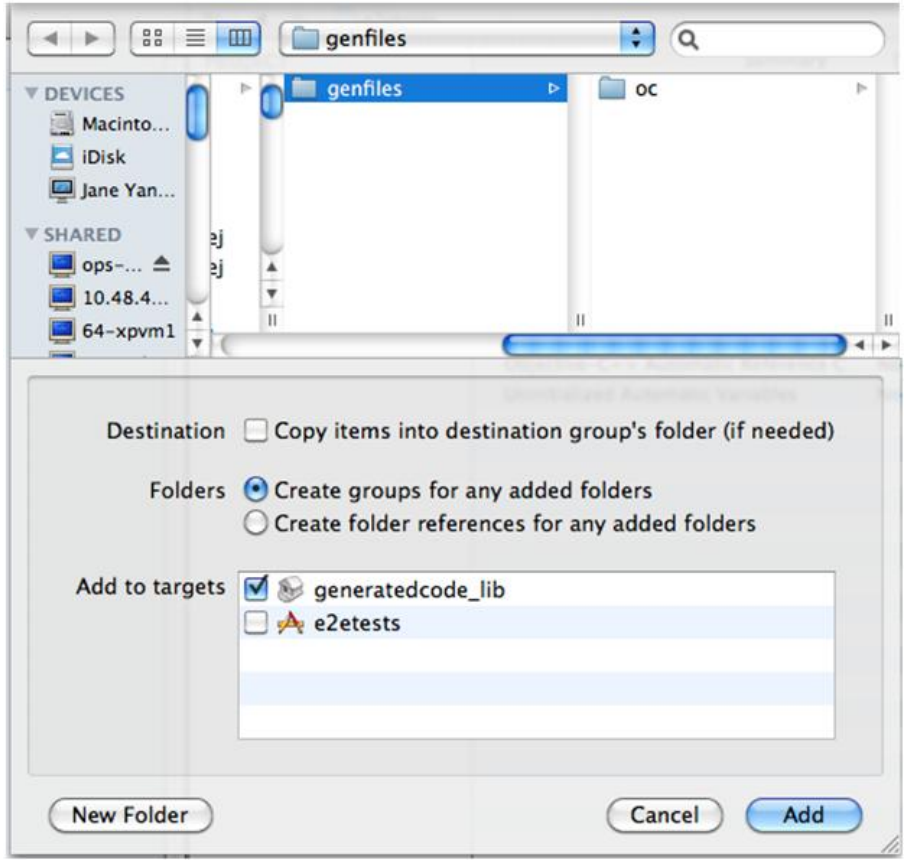
Import the generated MBO code and associated libraries into the iOS development environment, to support applications enabled with automatic reference counting (ARC).

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`).

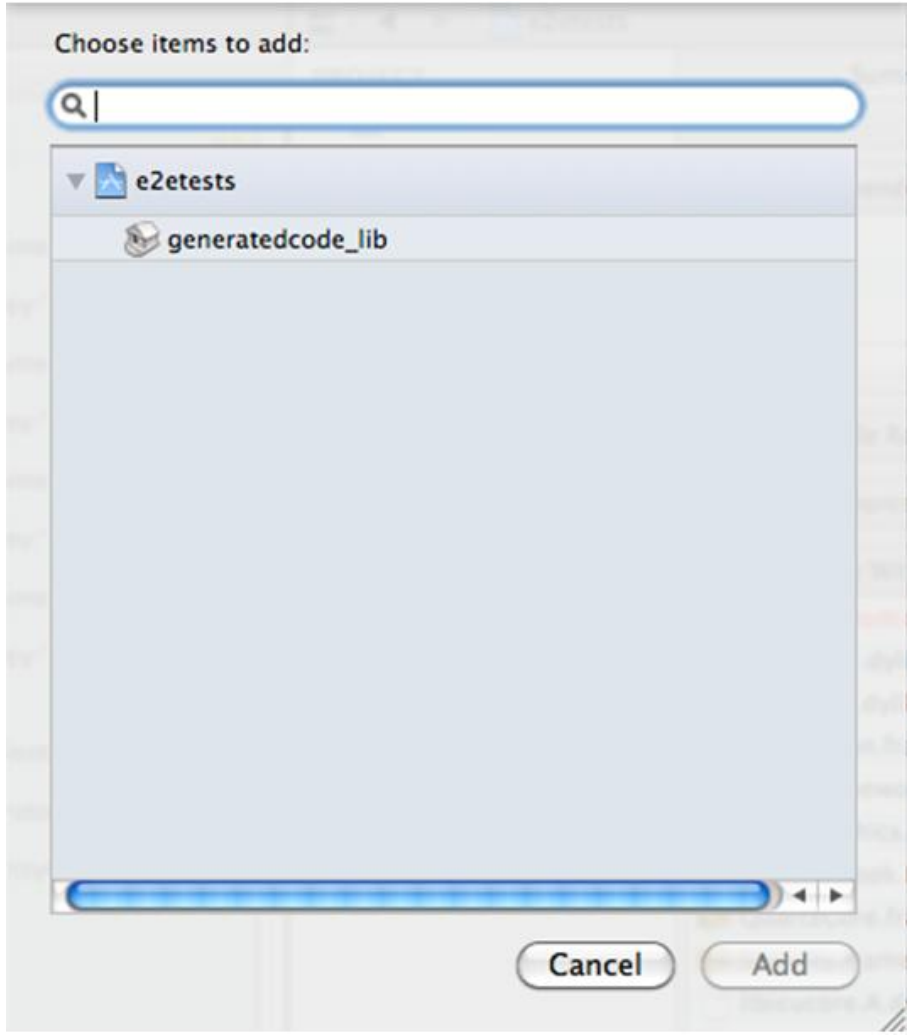
2. Make sure the static library is not using ARC by selecting the `generatedcode_lib` target, going to "Build Settings," and verifying "Automatic Reference Counting" is set to "NO".
3. Add generated code into the static library target.
   a) Right click on the `generatedcode_lib` folder from the Group & 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 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 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.

In addition, if your application is using replication based synchronization, and is synchronizing a large amount of data at the time the application goes into background, it may be necessary to interrupt the sync. To do this, the synchronization needs to be done using a sync status listener, and the `applicationDidEnterBackground` method must notify the listener to set the info.state flag to SYNC_STATUS_CANCEL (see *Developer Guide: iOS Object API Applications* > *Client Object API Usage* > *Callback and Listener APIs* > *SyncStatusListener API* for more details).

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.

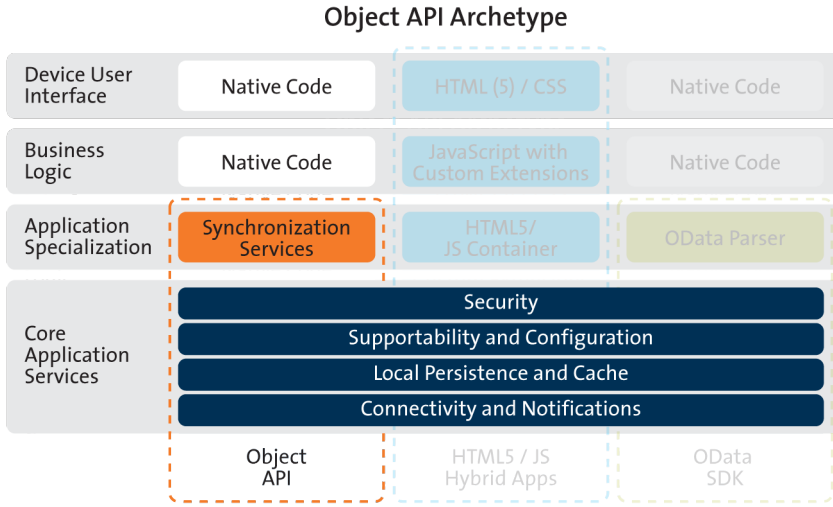
```
BOOL wasPreviouslyInBackground = NO;
```

```
- (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;
       [SUP101SUP101DB disableSync];
       [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
     {
       [SUP101SUP101DB enableSync];
       [SUPApplication startConnection:30];
     }
     @
     catch (NSException *ee)
     {
       // log an error or alert user via notification
     }
   });
}
```

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 DOE-based applications, connectivity 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. 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.

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 the Unwired Server.

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

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 *SUP_HOME*\MobileSDK22\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 *SUP_HOME*\MobileSDK22\ObjectAPI\Utils\genfiles directory, assuming codegen.bat is run from the *SUP_HOME*\MobileSDK22\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.

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*.

Downloading the Latest Afaria Libraries

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

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

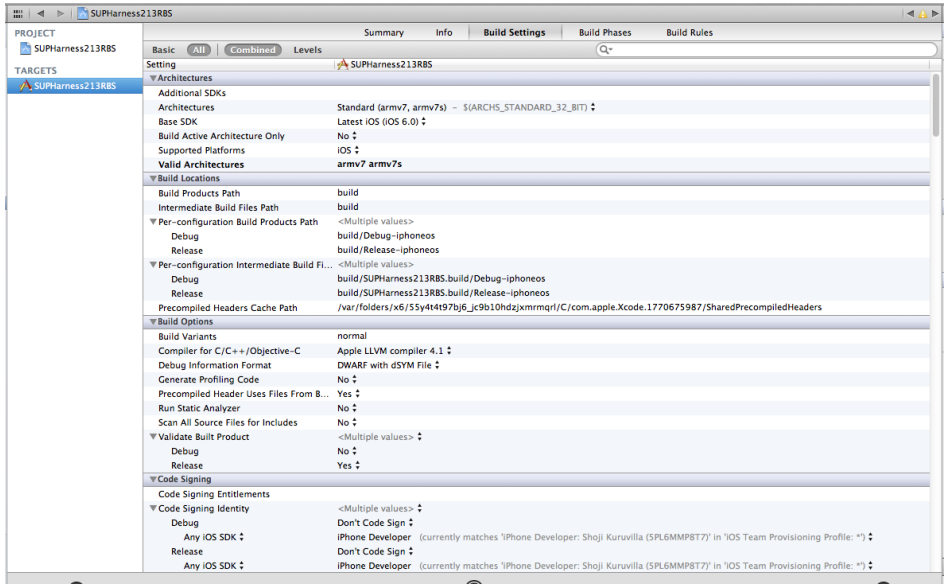
Importing Libraries and Code

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 4.5.1 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 6.0`.



5. Select the **Deployment** tab and set the iOS Deployment Target to `iOS 4.3` or later. Earlier SDKs and deployment targets are not supported.
6. Select the Valid Architectures with the default value `armv7 armv7s`, Supported Platforms as `iOS`, 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 *SUP_HOME*`\MobileSDK22\ObjectAPI\DOE\iOS` directory, and copy the `includes` and `Libraries` folders to the `<ProjectName>/<ProjectName>` directory on your Mac.
10. Navigate to the output directory that you specified when you generated objective-C code, and copy that 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 output folder with the generated Objective-C code that you just copied, unselect **Copy items into destination group's folder (if needed)**, and click **Add**.

    The output 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`, `libPerformanceLib.a`, `libsupClientUtil.a`, `libSUPSupportability.a`, `libsupSqlite.a`, `libAfariaSLL.a` and `libDatavault.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.

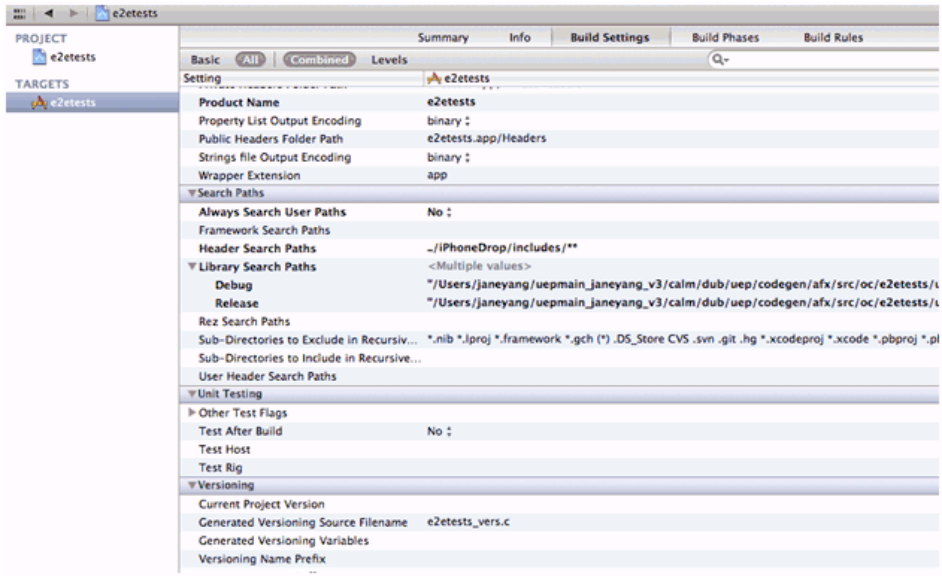
    Note: The library version corresponds to the configuration you are building. For example, if you are building for a debug version of the simulator, navigate to `libs/Debug-iphonesimulator/` to add the libraries.
13. Right-click the project root, select **New Group**, and then rename it to `Resources`.
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, in the middle pane click the **<ProjectName>** project, and set Automatic Reference Counting (ARC) to NO.
16. Click on the active target and modify the Library Search Path from the Building Settings. For example:

```
$(SRCROOT)/../iOS/Libraries/$(CONFIGURATION)$
(EFFECTIVE_PLATFORM_NAME)
```

Enter the path to the location where you copied the libraries. Specify separate profiles for debug and release, and specify "any iOS" and "any iOS simulator." Ensure that you escape the paths using double quotes.



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

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

18. 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:

- CoreFoundation.framework
- Security.framework
- CFNetwork.framework
- SystemConfiguration.framework
- MobileCoreServices.framework
- libicucore.A.dylib
- libstdc++.dylib
- libz.dylib

19. Hold the Option key, and select **Product > Clean Build Folder** 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.

20. In the `Info.plist` file, set the "Application does not run in background" setting to YES.

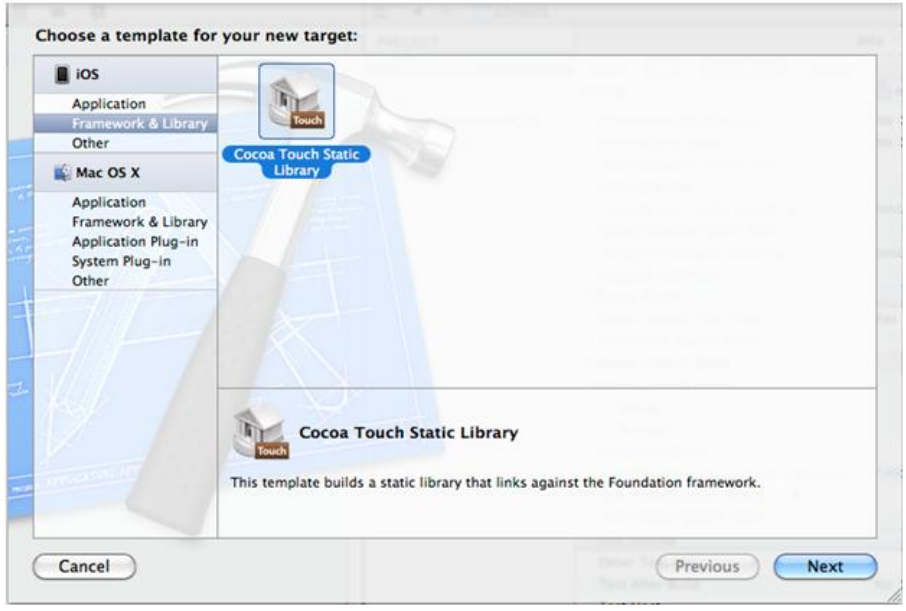
Note: If you want to allow your application to continue to run safely in the background, do not perform this step. See *Developer Guide: iOS Object API Applications* > *Development Task Flow for DOE-based Object API Applications* > *Creating a Project* > *Managing the Background State*.

21. Write your application code to reference the generated MBO code. See the *Developer Guide: iOS Object API Applications* for information about referencing the iOS Client Object API.

Importing Libraries and Code for Applications Enabled with ARC

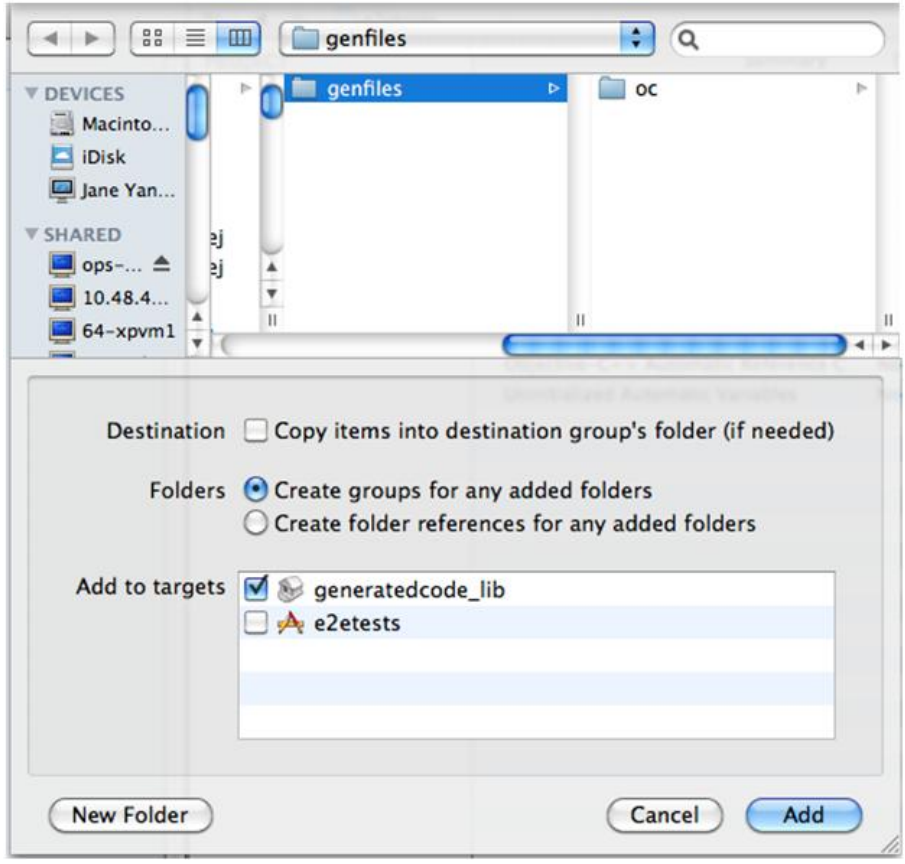
Import the generated MBO code and associated libraries into the iOS development environment, to support applications enabled with automatic reference counting (ARC).

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`).

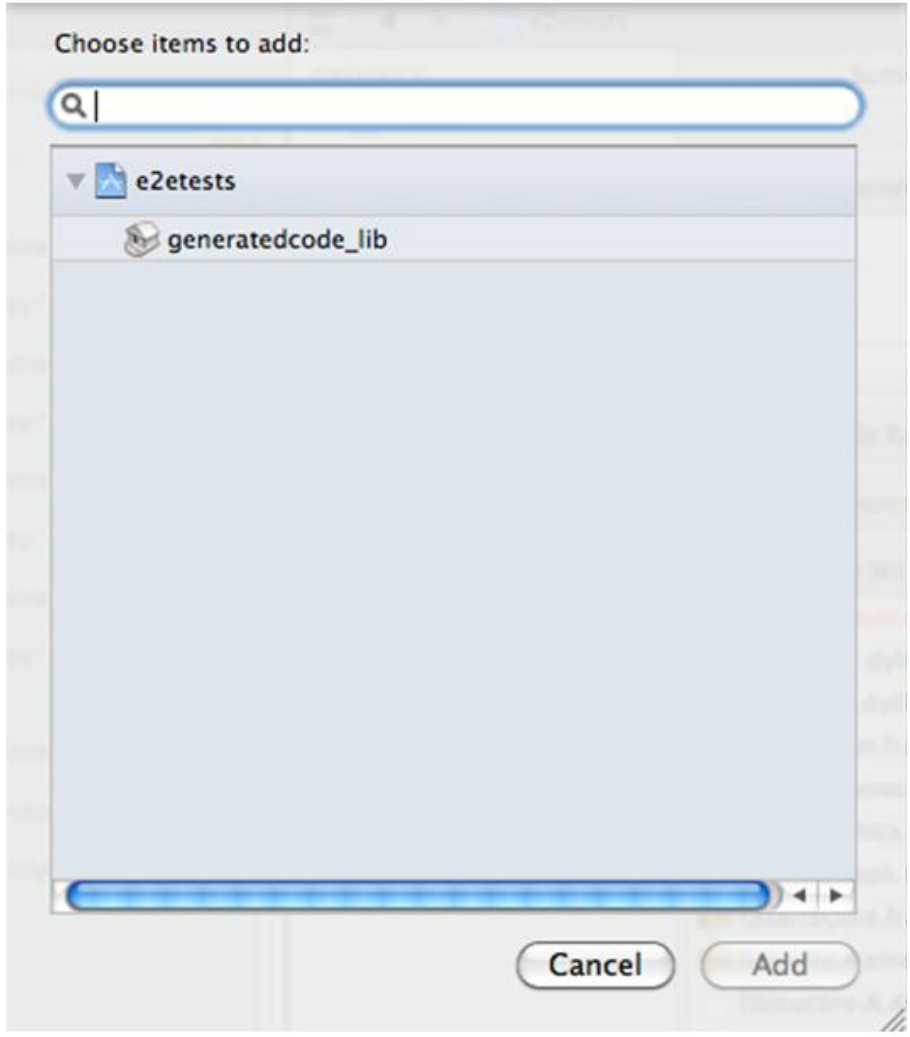
2. Make sure the static library is not using ARC by selecting the `generatedcode_lib` target, going to "Build Settings," and verifying "Automatic Reference Counting" is set to "NO".
3. Add generated code into the static library target.
   a) Right click on the `generatedcode_lib` folder from the Group & 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 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 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.

In addition, if your application is using replication based synchronization, and is synchronizing a large amount of data at the time the application goes into background, it may be necessary to interrupt the sync. To do this, the synchronization needs to be done using a sync status listener, and the `applicationDidEnterBackground` method must notify the listener to set the info.state flag to SYNC_STATUS_CANCEL (see *Developer Guide: iOS Object API Applications* > *Client Object API Usage* > *Callback and Listener APIs* > *SyncStatusListener API* for more details).

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;
       [SUP101SUP101DB disableSync];
       [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
     {
       [SUP101SUP101DB enableSync];
       [SUPApplication startConnection:30];
     }
     @
     catch (NSException *ee)
     {
       // log an error or alert user via notification
     }
   });
}
```

Developing the Application Using the Object API

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

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 Unwired 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
Unwired Server
app.applicationIdentifier = @"SMP101";

// ConnectionProperties has the information needed to register
// and connect to Unwired Server
SUPConnectionProperties* props = app.connectionProperties;
props.serverName = @"server.mycompany.com";
// if you are using Relay Server, then use the correct port number
```

```
for the Relay Server.
// if connecting using http without a relay server, use the messaging
administration port, by default 5001.
// if connecting using https without a relay server, then use a new
port for https, for example 9001.
props.portNumber = 5001;

// if connecting using https, set the network protocol
props.NetworkProtocol = @"https";

props.activationCode = @"activationcode";
// if you are connecting through relay server, then use the MBS
farmId for that Relay Server
// otherwise use the farmId from the SCC application connection
props.farmId = @"farmId";
// if you are connecting through relay server and using auto
registration,
// then you must provide the correct urlSuffix from the relay server
// Obtain the url suffix value from the Relay Server configuration
for the specific farm.
// The client url suffix value should be used in the application.
// For example: props.urlSuffix = @"/ias_relay_server/client/
rs_client.dll";
props.urlSuffix = @"urlSuffix";

// 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
[SMP101SMP101DB setApplication:app];
```

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

```
// specify Relay Server Host
Props.serverName = @"relayserver.mycompany.com";
// specify Relay Server Port (port 80 by default)
Props.portNumber = 80;
// specify the Relay Server MBS Farm, for example MBS_Farm
Props.farmId = @"MBS_FARM";
```

Optionally, you can specify the Relay Server URL suffix.

Using a Reverse Proxy for Object API applications

The Object API application communicates with Unwired Server through two ports:

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

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

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

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

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

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

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

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

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

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

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

```
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]);
}

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

@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 UltraLite®J runtime tuning values.

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

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

Use the `SUPConnectionProfile` class to set up the locally generated database. Retrieve the connection profile object using the Sybase Unwired Platform database's `getConnectionProfile` method.

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

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

Note: Although multiple threads are allowed to write to the database at the same time, however, there will be errors when multiple threads write to the same MBO. Develop your application in such a way to avoid this situation.

Setting Up Connectivity

Store connection information to the Unwired Server data synchronization channel.

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 setServerName:@"xxxx"];
   [cp setPortNumber:2480];
   ```

Creating and Deleting a Device's Local Database

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

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

   ```
   [SUP101SUP101DB openConnection];
   ```

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

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

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

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

3. 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.

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

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

```
SUPConnectionProfile *syncProfile = [SUP101SUP101DB
getSynchronizationProfile];
[syncProfile setUser:@"user"];
[syncProfile setPassword:@"password"];
[SUP101SUP101DB onlineLogin];
```

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 register 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*.

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. Set the synchronization parameters if there are any.
2. Make a blocking synchronize call to Unwired Server to pull in all MBO data:
   ```
   [SUP101SUP101DB synchronize];
   ```
3. List all customer MBO instances from the local database using an object query, such as `findAll`, which is a predefined object query.
   ```
   SUPObjectList *objlist = [SUP101Customer findAll];
   ```
4. Find and update a particular MBO instance, and save it to the local database.
   ```
   SUP101Customer *customer = [SUP101Customer findByPrimaryKey:
   32838];
   //Change some sttribute of the customer record
   ```

```
customer.fname= @"New Name";
[customer save];
```

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

   ```
   [Customer submitPending];
   ```

6. 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.

Note: The `getSynchronizationParameters` method has been deprecated.

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

1. List all the synchronization parameters.

   ```
   SUPObjectList* r =  [SKPKCustomer getSubscriptions] ;
   ```

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

   ```
   SKPKCustomerSubscription *sp = [SKPKCustomerSubscription
   getInstance];
   ```

```
sp.name = @"example";
[SKPKCustomer addSubscription:sp];
```

3. Synchronize to download the data:

```
[SUP101SUP101DB synchronize];
```

Subsequently Starting an Application

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

Starting an application on subsequent occasions:

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

   Note: Once the application is registered, changes to any of the application connection properties do not take effect. To modify the connection properties, unregister the application, change the connection properties and then register again. Unregistering the application also removes the user from the server.
2. Set up the connection profile properties if needed for database location and tuning parameters.
3. Set up the synchronization profile properties if needed for SSL or a relay server.
4. Start the application connection to the server using the existing connection parameters and registration information.

```
[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 "key" as int, the Customer MBO would contain the `+ (Customer*) findByPrimaryKey:`

`(int)key` method, which performs the equivalent of `Select x.* from Customer x where x.key = :key`.

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:

   ```
   SUP101Customer *customer = [SUP101Customer findByPrimaryKey:
   32838];
   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:

```
SUP101Customer *customer = [SUP101Customer findByPrimaryKey:
32838];
customer.city = @"Dublin";
SUP101Sales_order* order = [SUP101Sales_order findByPrimaryKey:
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 MBO Records

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

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

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

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

   ```
   SUP101Customer *newcustomer = [[SUP101Customer 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:

```
SUP101Customer *customer = [SUP101Customer findByPrimaryKey:
32838]; //find by the primary key
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:

```
SUP101Customer *customer = [SUP101Customer findByPrimaryKey:
32838];
[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 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:

```
SUP101CustomerOtherOperation *other =
[[SUP101CustomerOtherOperation 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];
[SUP101SUP101DB synchronize];
```

Using SubmitPending and SubmitPendingOperations

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

Note that **submitPendingOperations** APIs are expensive. 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.
5. For a single MBO, you must call the `refresh()` method of the MBO instance before you use this instance again.
6. For related MBOs, you must call the `refresh()` method of the MBO instance before you use this instance again, even if the MBO's child or parent has called `submitPending`.

Shutting Down the Application

Shut down an application and clean up connections.

Closing Connections

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

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

   ```
   [app stopConnection:<timeout_value>];
   ```

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

Debugging Runtime Errors and Performance Analysis

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

End to End Tracing

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

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

On the client side, the client framework enables an application developer to switch on the trace for messages. The client traces the request at predefined points and all these transactions/ requests are recorded in a Business Transaction XML. Additionally, the client maintains a unique identifier in the HTTP header called the SAP Passport that is used to correlate traces across various components. This Business Transaction XML can later be uploaded to the SAP Solution Manager which is a central location to correlate all logging information.

Using Tracing APIs

Enable end-to-end tracing from mobile device to enterprise information system (EIS) for Object API applications using SAP® Solution Manager.

The API consists of the following interfaces or classes:

- **SUPE2ETraceService –** A public interface for use by the application's user interface developers.
- **SUPE2ETraceLevel –** Defines an enumeration of the trace levels that you can set to a passport. Trace levels control the amount of logging done on the server side.
- **SUPE2ETraceServiceImpl –** The implementation of the `SUPE2EtraceService` interface; the implementation is a singleton. There are additional methods for you to create a passport and business transaction.
- **SUPE2ETraceMessage –** An entity class which holds the request/response details and statistics and the passport. Object API internally makes use of this class to add request/

response details to the business transaction and to get a new passport for each new request. Object API sets the new passport to the HTTP header, 'SAP-PASSPORT' and sends it to the server side, so that the server can continue processing the E2E tracing.

Getting an Instance of the E2E Trace Service

Get an instance of the SUPE2ETraceService interface.

You can create a new instance in one of two ways.

Instantiate the object through its implementation class:

```
[SUPE2ETraceServiceImpl getInstance];
```

Instantiate the object through `SUPApplication`:

```
[SUPApplication getE2ETraceService];
```

Initializing the Trace

Set the trace level and start the trace.

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

Trace Level	Description
0 (NONE)	Specific to trace analysis on the client. No traces are triggered on the server.
1 (LOW)	Corresponds to response time- distribution analysis. This helps to analyse the time taken on each server component.
2 (MEDIUM)	Corresponds to performance analysis. Performance traces are triggered on the server side. Example: Introscope Transaction Trace, ABAP Trace, SQL Traces and so on.
3 (HIGH)	Corresponds to functional analysis.

```
SUPE2ETraceLevel level = SUPE2ETraceLevel_NONE;
switch (val)
{
  case 1:
    level = SUPE2ETraceLevel_LOW;
    break;
...
SUPE2ETraceServiceImpl *e2eTraceService = [SUPE2ETraceServiceImpl
getInstance];
[e2eTraceService setTraceLevel:level];
[e2eTraceService startTrace];
```

When you call the `startTrace` method, the `SUPE2ETraceService` initializes the trace and sets appropriate flags to indicate the trace has started. The method may perform other tasks as required by SAP's BTX API, such as getting a handle to the BTX writer from the BTX API.

Stopping the Trace

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

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

```
NSData *btx = [ [SUPE2ETraceServiceImpl getInstance]  stopTrace];
```

Uploading the BTX

Upload the business transaction to the server.

Upload the business transaction by calling `uploadTrace:(NSData *)btx` and passing the BTX byte array. The method returns true if the upload succeeds, otherwise it throws an `SUPE2ETraceUploadException`.

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

```
//ensure this blocking call gets executed in a separate thread
@try
{
  [traceService uploadTrace:btx];
}@catch (SUPE2ETraceUploadException *eue) {}
```

Tracking KPI

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

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

Specific interval types measure some additional KPIs:

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

After the interaction is stopped, a summary log in csv format and a detailed log in txt format is written to the device.The summary log contains sums of each of the KPI types. For example,

total Wallclock Time, total CPU Time, total number of roundTrips, total number of PersistenceRead, total CPU Time of PersistenceWrite, etc. The detailed log also contains a summary line, as well as KPI values for each interval.

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

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

```
- (void)startInteraction:(NSString *)interactionName;
```

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

```
- (void)stopInteraction;
```

Example of application interactions for collecting KPI:

```
// get the instance
id <SUPPerformanceAgentService> pa = [SUPPerfAgentServiceImpl
getInstance];
[pa startInteraction:@"Interaction 1";];
// application interaction
// ...
// ...
[pa stopInteraction];

[pa startInteraction:@"Interaction 2";];
// application interaction
// ...
// ...
[pa stopInteraction];
```

The following limitations apply:

- On iOS devices, there is no detailed log file written after the interaction is stopped.

Uninstalling the Application

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

Deleting the Database and Unregistering the Application

Delete the package database, and unregister the application.

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

   ```
   @try {
     [app unregisterApplication:<time out value>]
   }
   @catch (SUPApplicationTimeoutException * pe) {
   ```

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

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

```
[SUP101SUP101DB deleteDatabase];
```

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 the 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 applications screen, open the application.
3. 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.
4. 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,requestI
  d,timestamp
  ```

  This log format generates output similar to:

  ```
  level code eisCode message component entityKey operation requestId
  timestamp
   5,500,'','java.lang.SecurityException:Authorization failed:
  Domain = default Package = end2end.rdb:1.0 mboName =
  simpleCustomer action =
  delete','simpleCustomer','100001','delete','100014','2010-05-11
  14:45:59.710'
  ```

  - `level` – the log level currently set. Values include: 1 = TRACE, 2 = DEBUG, 3 = INFO, 4 = WARN, 5 = ERROR, 6 = FATAL, 7 = OFF.
  - `code` – Unwired Server administration codes.
    - Synchronization codes:
      - 200 – success.
      - 500 – failure.
  - `eisCode` – maps to HTTP error codes. If no mapping exists, defaults to error code 500 (an unexpected server failure).
  - `message` – the message content.
  - `component` – MBO name.
  - `entityKey` – MBO surrogate key, used to identify and track MBO instances and data.
  - `operation` – operation name.
  - `requestId` – operation replay request ID or messaging-based synchronization message request ID.
  - `timestamp` – message logged time, or operation execution time.
- If you have implemented `ApplicationCallback.`**onConnectionStatusChanged** for synchronization in the `CallbackHandler`, the connection status between the 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 Sybase Unwired Platform* > *Administer* > *Unwired Server* > *Server Log* > *Unwired Server Runtime Logging* > *Configuring Unwired Server Log Settings*.

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

- Obtain DEBUG information for a specific device:
  - In the SCC administration console:
    1. Set the DEBUG level to a higher value for a specified device:
       a. In SCC, select **Application Connections**, then select **Properties... > Device Advanced**.
       b. Set the Debug Trace Level value.
    2. Set the TRACE file size to be greater than 50KB.
    3. View the trace file through SCC.
  - Check the *SUP_HOME*\Servers\UnwiredServer\logs\ClientTrace directory to see the mobile device client log files for information about a specific device.

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

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).

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 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.
3. 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.

4. 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.
5. Create the Xcode project, ensuring the bundle identifier corresponds to the bundle identifier in the specified App ID.
6. 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 Sybase Unwired Platform to enable APNS and
pass the token to the Unwired 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
inside the didFailToRegisterForRemoteNotificationsWithError
```

```
method:

// +(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 inside
the didReceiveRemoteNotification method:

+(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 might include badges, sounds, or custom text alerts. Device users can use Settings to customize which notifications to receive or ignore.

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. Upload the APNS certificate to Sybase Control Center:
   a) In the navigation pane, click **Applications**.
   b) In the administration pane, click the **Applications** tab.
   c) Select the application for which you want to enable APNS, and click **Properties**.
   d) Click the **Push Configurations** tab and click on **Add**.
   e) Configure all required properties, including the corresponding password and upload the certificate. See *APNS Native Notification Properties* in *Sybase Control Center for Sybase Unwired Platform* online help.
3. In Sybase Control Center, configure Native Notifications for an application connection.

For details, see *Configuring Native Notifications* in *Sybase Control Center for Sybase Unwired Platform*.

4. Deploy the iOS application with an enterprise distribution provisioning profile to users' iOS devices.
5. 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**.
6. Verify that the APNS-enabled iOS device is set up correctly:
   a) In Sybase Control Center, ensure the user has already activated the application and is connected to the Unwired Server, by looking for the corresponding entry in **ApplicationsApplication Connections**.
   b) Validate that in the Application Connection ID, the application name appears correctly at the end of the string.
   c) Select the user and click **Properties**.
   d) Check that the *APNS Device Token* contains a value. This indicates that a token has passed successfully following a successful application activation
7. 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.
8. To troubleshoot APNS, use the *SUP_HOME*`\Servers\UnwiredServer\logs\server` log file.

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: Review complete details in the *iPhone OS Enterprise Deployment Guide* at *http://manuals.info.apple.com/en_US/Enterprise_Deployment_Guide.pdf*, and *About Your First App Store Submission* at *https://developer.apple.com/library/ios/#documentation/ToolsLanguages/Conceptual/YourFirstAppStoreSubmission/AboutYourFirstAppStoreSubmission/AboutYourFirstAppStoreSubmission.html#//apple_ref/doc/uid/TP40011375-CH1-SW1*.

Note:

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 Sybase 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* > *Developing the Application Using the Object API*.

Client Object API Reference

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

Review the reference details in the Client Object API documentation, located in *SUP_HOME*`\MobileSDK22\ObjectAPI\iOS\headerdoc`.

Application APIs

The `SUPApplication` class 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.

Application

Methods or properties in the `SUPApplication` class.

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` or `registerApplication`.

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`.

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];
[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;
[app registerApplication]; // method returns immediately
```

registerApplication: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). 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]);
}
```

ApplicationCallback Property

Callback for the current application. It is optional (but recommended) to set a callback, so that the application can respond to changes of connection status, registration status and application settings.

Syntax

```
public IApplicationCallback ApplicationCallback { get; set; }
```

Examples

- **Get the current ApplicationCallback handler**

```
application.ApplicationCallback = new MyApplicationCallback();
```

startConnection: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). If the timeout value is less than or equal to 0, then this method returns immediately without waiting for the registration to finish (a non-blocking call).

Returns

None.

Examples

- **Start the Application**

  ```
  [app startConnection: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
```

`ConnectionStatus` has the following possible values:

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

Examples

- **Get the Application Connection Status**

  ```
  [SUPApplication connectionStatus];
  ```

getConnectionProperties

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

Syntax

```
+  (SUPConnectionProperties*)connectionProperties;
```

Parameters

None.

Returns

Returns the connection properties instance.

ApplicationSettings Property

Return application settings that have been received from the Unwired Server after application registration and connection.

Syntax

```
Sybase.Mobile.ApplicationSettings ApplicationSettings { get; set; }
```

Returns

Application settings that have been received from the Unwired Server.

Examples

- **Get the Application Settings**

  ```
  ApplicationSettings applicationSettings =
  Application.GetInstance().ApplicationSettings
  ```

beginDownloadCustomizationBundle :(NSStream*)outputStream

Starts downloading the default resource bundle associated with the application, and saves it into the output stream that you provide.

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

Syntax

```
-(void) beginDownloadCustomizationBundle :(NSStream*)outputStream;
```

Parameters

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

Returns

None.

Examples

- **Download**

```
// Download the default bundle file and save it to the
defaultBundle.jar file
SUPApplication* app = [SUPApplication getInstance];
NSOutputStream* ostream = [self
openOutputStream:@"defaultBundle.jar"];
[app beginDownloadCustomizationBundle:ostream];
```

beginDownloadCustomizationBundle:(NSString*)customizationBundleID withOutputStream:(NSOutputStream*)outputStream

Start downloading the resource bundle named `customizationBundleID` and save it to an output stream.

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

Syntax

```
-(void) beginDownloadCustomizationBundle:
(NSString*)customizationBundleID withOutputStream:
(NSOutputStream*)outputStream;
```

Parameters

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

Returns

None.

Examples

- **Download**

```
// Download a specific (“Example”) resource bundle and save to
Example.jar
SUPApplication* app = [SUPApplication getInstance];
NSOutputStream* ostream = [self openOutputStream: @"
Example.jar"];
```

```
app beginDownloadCustomizationBundle:@"Example:2.0"
withOutputStream:ostream];
```

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. If the timeout value is less than or equal to 0, then this method returns immediately without waiting for the registration to finish (a non-blocking call).

Returns

None.

Examples

- **Stop the Application**

  ```
  [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.

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

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. If the timeout value is less than or equal to 0, then this method returns immediately without waiting for the registration to finish (a non-blocking call).

Examples

- **Unregister an Application –** Unregisters the application with a one minute waiting period.
```
[app unregisterApplication:60];
```

ConnectionProperties

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

activationCode

Retrieves or sets the activation code. If you register an application manually, you must set an activation code.

Syntax

```
@property(readwrite, retain, nonatomic) NSString* activationCode;
```

Parameters

None.

Returns

Returns the activation code.

networkProtocol

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

Syntax

```
@property(readwrite, retain, nonatomic) NSString* networkProtocol;
```

Parameters

None.

Returns

Returns the network protocol for the server connection URL.

loginCertificate

Retrieve the login certificate, or set this property to enable authentication by a digital certificate.

Syntax

```
@property(readwrite, retain, nonatomic) SUPLoginCertificate
*loginCertificate;
```

Parameters
None.

Returns

Returns the login certificate.

loginCredentials
Retrieve the login credentials, or set this property to enable authentication by username and password..

Syntax
```
 @property(readwrite, copy, nonatomic) SUPLoginCredentials
*loginCredentials;
```

Parameters
None.

Returns

Returns the login credentials.

portNumber
Retrieve or set the port number for the server connection URL.

Syntax
```
@property(readwrite) int32_t portNumber;
```

Parameters
None.

Returns

Returns the port number.

serverName
Retrieve or set the server name for the server connection URL.

Syntax
```
@property(readwrite, retain, nonatomic) NSString* serverName;
```

Parameters
None.

Returns

Returns the server name.

securityConfiguration

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

Syntax

```
@property(readwrite, retain, nonatomic) NSString*
securityConfiguration;
```

Parameters

None.

Returns

Returns the security configuration.

urlSuffix

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

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

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

Syntax

```
@property(readwrite, retain, nonatomic) NSString* urlSuffix;
```

Parameters

None.

Returns

Returns the URL suffix.

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

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

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

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

result in the following URL suffix:

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

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

Syntax
```
@property(readwrite, retain, nonatomic) NSString* farmId;
```

Parameters
None.

Returns

Returns the Farm ID.

httpHeaders
Retrieve or set any custom headers for HTTP network communications with a proxy server or Relay Server.

Syntax
```
@property(readwrite, retain, nonatomic) SUPStringProperties*
httpHeaders;
```

Parameters
None.

Returns

Returns the HTTP headers.

httpCookies

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

Syntax

```
@property(readwrite, retain, nonatomic) SUPStringProperties*
httpCookies;
```

Parameters

None.

Returns

Returns the HTTP cookies.

httpCredentials

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

Syntax

```
@property(readwrite, retain, nonatomic) SUPLoginCredentials
*httpCredentials;
```

Parameters

None.

Returns

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

ApplicationSettings

Methods or properties in the `SUPApplicationSettings` class.

isApplicationSettingsAvailable

Checks whether the application settings are available from the Unwired Server.

Syntax

```
- (BOOL) isApplicationSettingsAvailable;
```

Parameters
None.

Returns
Returns true if the application settings are available.

Examples

- **Check if application settings are available**

```
BOOL isSettingsAvailable = [[SUPApplication
getInstance].applicationSettings isApplicationSettingsAvailable];
```

getStringProperty
Retrieves a string property from the `applicationSettings`.

Syntax
```
+ (NSString*)getStringProperty:(SUPConnectionPropertyType)propId;
```

Parameters

- Type of ConnectionPropertyType.
- **propId –** The property ID of the SUPConnectionPropertyType.

Returns

Returns a string property value.

Examples

- **Get string property**

```
NSString *username = [[SUPApplication
getInstance].applicationSettings
getStringProperty:USERNAME_PROP_ID];
```

getIntegerProperty
Retrieves an integer property from the `applicationSettings`.

Syntax
```
+ (int) getIntProperty:(SUPConnectionPropertyType)propId;
```

Parameters

- Type of ConnectionPropertyType.
- **propId –** The property ID of the SUPConnectionPropertyType.

Returns
Returns an integer property value.

Examples

- **Get integer property**

```
int min_length = [[SUPApplication
getInstance].applicationSettings
getIntegerProperty:PWDPOLICY_MIN_LENGTH_PROP_ID];
```

getBooleanProperty
Retrieves a boolean property from the `applicationSettings`.

Syntax
```
+ (BOOL) getBooleanProperty:(SUPConnectionPropertyType)propId;
```

Parameters

- Type of ConnectionPropertyType.
- **propId –** The property ID of the SUPConnectionPropertyType.

Returns
Returns a boolean property value.

Examples

- **Get boolean property**

```
BOOL pwdpolicy_has_lower = [[SUPApplication
getInstance].applicationSettings getBooleanProperty:
PWDPOLICY_HAS_LOWER_PROP_ID];
```

custom1
A custom application setting for use by the application code.

Syntax
```
- (NSString*)custom1
```

Parameters
None.

Returns

Returns a custom application setting.

Examples

- **Custom application setting**

```
SUPApplicationSettings* applicationSettings = [[SUPApplication
getInstance] applicationSettings];
NSString* custom1 = [applicationSettings custom1];
```

custom2

A custom application setting for use by the application code.

Syntax

```
- (NSString*)custom2
```

Parameters

None.

Returns

Returns a custom application setting.

Examples

- **Custom application setting**

```
SUPApplicationSettings* applicationSettings = [[SUPApplication
getInstance] applicationSettings];
NSString* custom2 = [applicationSettings custom2];
```

custom3

A custom application setting for use by the application code.

Syntax

```
- (NSString*)custom3
```

Parameters

None.

Returns

Returns a custom application setting.

Examples

- **Custom application setting**

```
SUPApplicationSettings* applicationSettings = [[SUPApplication
getInstance] applicationSettings];
NSString* custom3 = [applicationSettings custom3];
```

custom4

A custom application setting for use by the application code.

Syntax

```
- (NSString*)custom4
```

Parameters

None.

Returns

Returns a custom application setting.

Examples

- **Custom application setting**

```
SUPApplicationSettings* applicationSettings = [[SUPApplication
getInstance] applicationSettings];
NSString* custom4 = [applicationSettings custom4];
```

domainName

Syntax

```
- (NSString*)domainName
```

Parameters

None.

Returns

Returns the domain name.

Examples

- **Domain name**

```
SUPApplicationSettings* applicationSettings = [[SUPApplication
getInstance] applicationSettings];
NSString* domainName = [applicationSettings domainName];
```

connectionId

Syntax

```
- (NSString*) connectionId
```

Parameters

None.

Returns

Returns a Connection ID for this application setting.

Examples

- **Connection ID**

```
SUPApplicationSettings* applicationSettings = [[SUPApplication
getInstance] applicationSettings];
NSString* connectionId= [applicationSettings connectionId];
```

ConnectionPropertyType

Methods or properties in the `SUPConnectionPropertyType` class.

PwdPolicy_Enabled

Indicates whether the password policy is enabled.

Syntax

```
ConnectionPropertyType PwdPolicy_Enabled
```

Parameters

None.

Returns

Examples

- **PwdPolicy_Enabled**

```
BOOL pwdpolicy_enabled = [[SUPApplication
getInstance].applicationSettings
getBooleanProperty:PWDPOLICY_ENABLED_PROP_ID];
```

PwdPolicy_Default_Password_Allowed

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

Syntax

```
ConnectionPropertyType PwdPolicy_Default_Password_Allowed
```

Parameters

None.

Returns

None.

Examples

- **PwdPolicy_Default_Password_Allowed**

```
BOOL pwdpolicy_default_pwd_allowed = [[SUPApplication
getInstance].applicationSettings
getBooleanProperty:PWDPOLICY_DEFAULT_PASSWORD_ALLOWED_PROP_ID];
```

PwdPolicy_Length

Defines the minimum length for a password.

Syntax

```
ConnectionPropertyType PwdPolicy_Length
```

Parameters

None.

Returns

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

Examples

- **PwdPolicy_Length**

```
int min_length = [[SUPApplication
getInstance].applicationSettings
getIntegerProperty:PWDPOLICY_MIN_LENGTH_PROP_ID];
```

PwdPolicy_Has_Digits

Indicates if the password must contain digits.

Syntax

```
ConnectionPropertyType PwdPolicy_Has_Digits
```

Parameters

None.

Returns

Returns true if the password must contain digits.

Examples

- **PwdPolicy_Has_Digits**

```
BOOL pwdpolicy_has_digits = [[SUPApplication
getInstance].applicationSettings getBooleanProperty:
PWDPOLICY_HAS_DIGITS_PROP_ID];
```

PwdPolicy_Has_Upper

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

Syntax

```
ConnectionPropertyType PwdPolicy_Has_Upper
```

Parameters

None.

Returns

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

Examples

- **PwdPolicy_Has_Upper**

```
BOOL pwdpolicy_has_upper = [[SUPApplication
getInstance].applicationSettings getBooleanProperty:
PWDPOLICY_HAS_UPPER_PROP_ID];
```

PwdPolicy_Has_Lower

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

Syntax

```
ConnectionPropertyType PwdPolicy_Has_Lower
```

Parameters

None.

Returns

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

Examples

- **PwdPolicy_Has_Lower**

```
BOOL pwdpolicy_has_lower = [[SUPApplication
getInstance].applicationSettings getBooleanProperty:
PWDPOLICY_HAS_LOWER_PROP_ID];
```

PwdPolicy_Has_Special

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

Syntax

```
ConnectionPropertyType PwdPolicy_Has_Special
```

Parameters

None.

Returns

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

Examples

- **PwdPolicy_Has_Special**

```
BOOL pwdpolicy_has_special = [[SUPApplication
getInstance].applicationSettings getBooleanProperty:
PWDPOLICY_HAS_SPECIAL_PROP_ID];
```

PwdPolicy_Expires_In_N_Days

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

Syntax

```
ConnectionPropertyType PwdPolicy_Expires_In_N_Days
```

Parameters

None.

Returns

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

Examples

- **PwdPolicy_Expires_In_N_Days**

```
int expires_in_n_days = [[SUPApplication
getInstance].applicationSettings
getIntegerProperty:PWDPOLICY_EXPIRES_IN_N_DAYS_PROP_ID];
```

PwdPolicy_Min_Unique_Chars

Specifies the minimum number of unique characters in the password.

Syntax

```
ConnectionPropertyType PwdPolicy_Min_Unique_Chars
```

Parameters

None.

Returns

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

Examples

- **PwdPolicy_Min_Unique_Characters**

```
int min_unique_characters = [[SUPApplication
getInstance].applicationSettings
getIntegerProperty:PWDPOLICY_MIN_UNIQUE_CHARS_PROP_ID];
```

PwdPolicy_Lock_Timeout

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

Syntax

```
ConnectionPropertyType PwdPolicy_Lock_Timeout
```

Parameters

None.

Returns

An integer specifying the timeout value.

Examples

- **PwdPolicy_Lock_Timeout**

```
int lock_timeout = [[SUPApplication
getInstance].applicationSettings
getIntegerProperty:PWDPOLICY_LOCK_TIMEOUT_PROP_ID];
```

PwdPolicy_Retry_Limit

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

Syntax

```
ConnectionPropertyType PwdPolicy_Retry_Limit
```

Parameters

None.

Returns

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

Examples

- **PwdPolicy_Retry_Limit**

```
int pwdpolicy_retry_limit = [[SUPApplication
getInstance].applicationSettings
getIntegerProperty:PWDPOLICY_RETRY_LIMIT_PROP_ID];
```

Afaria APIs

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

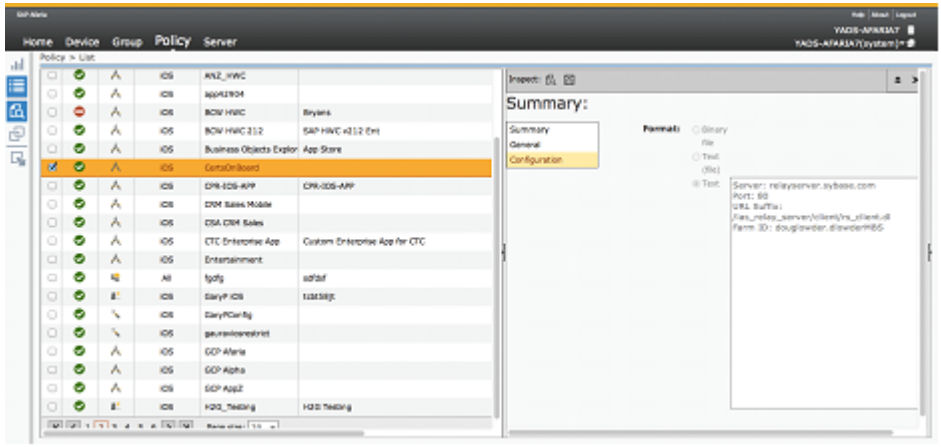
Using Afaria to Provision Configuration Data

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

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

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

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



You can obtain configuration data for your application using Afaria by calling the following API from the `SeedingAPISynchronous` class (in Afaria's `SeedingAPISynchronous.h` header file:

```
+ (NSInteger)retrieveSeedData:(NSString *)urlScheme InFile:
(NSMutableString *)seedFile withCredentials:(NSURLCredential
*)credentials;
```

Or, call this asynchronous API from the `SeedDataAPI` class (in `SeedDataAPI.h`):

```
- (void)retrieveSeedData;
```

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

This example code retrieves the configuration data using the Afaria API, parses it using the native iOS APIs, and applies the appropriate settings using the Sybase Unwired Platform APIs (the `SUPApplication` and `SUPConnectionProperties` classes).

```
NSMutableString *seedFile = [NSMutableString string];
retCode = [SeedingAPISynchronous retrieveSeedData:@"certsonboard-
seed" InFile:seedFile withCredentials:nil];
NSError *error = nil;
switch(retCode)
{
  case kSeedDataAvailable:  // Seed data is available, read the file
    NSLog(@"Seed file = %@",seedFile);
    NSLog(@"Seed data = %@",[NSString
stringWithContentsOfFile:seedFile encoding:NSUTF8StringEncoding
error:&error]);
    break;
  case kSeedDataUnavailable:
    NSLog(@"kSeedDataUnavailable"); // Error
    break;
  case kAfariaClientNotInstalled:
    NSLog(@"kAfariaClientNotInstalled");  // Error
    break;
  case kAfariaSettingsRequested:
    NSLog(@"kAfariaSettingsRequested");  // Error
    break;
}

// Read the text from the Afaria configuration file
NSString *configurationText = [NSString
stringWithContentsOfFile:seedFile encoding:NSUTF8StringEncoding
error:&error];

// Separate the text into lines
NSArray *configurationLines = [configurationText
componentsSeparatedByString:@"\n"];

// Create a dictionary, and go through the lines to find name value
pairs
NSMutableDictionary *settings = [NSMutableDictionary dictionary];
for(NSString *s in configurationLines)
{
  NSArray *nvpair = [s componentsSeparatedByString:@": "];
```

```
  if([nvpair count] == 2)
    [settings setValue:[nvpair objectAtIndex:1] forKey:[nvpair
objectAtIndex:0]];
}

// Use the name value pairs from the configuration file to set the
appropriate settings in the SUPApplication API
SUPApplication *app = [SUPApplication getInstance];
app.applicationIdentifier = @"myAppID";

SUPConnectionProperties *properties = app.connectionProperties;

properties.serverName = [settings valueForKey:@"Server"];
properties.portNumber = [[settings valueForKey:@"Port"] intValue];
properties.farmId = [settings valueForKey:@"Farm ID"];
properties.urlSuffix = [settings valueForKey:@"URL Suffix"];

NSLog(@"Server name is set to %@",properties.serverName);
NSLog(@"Port number is set to %d",properties.portNumber);
NSLog(@"Farm ID is set to %@",properties.farmId);
NSLog(@"URL suffix is set to %@",properties.urlSuffix);
```

Example output on the Xcode console:

```
2012-09-24 13:06:33.014 CertsOnboard[579:707] Seed file = /var/
mobile/Applications/21935FE8-843A-418D-A2BF-EE415B5D4DF0/Documents/
TEXT_FILE_
2012-09-24 13:06:33.016 CertsOnboard[579:707] Seed data = Server:
relayserver.sybase.com

Port: 80
URL Suffix: /ias_relay_server/client/rs_client.dl
Farm ID: example.exampleMBS
```

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

Using Certificates from Afaria for Authentication

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

Prerequisites:

- The iOS application has been built using the Sybase Unwired Platform generated code and framework headers and libraries.
- The iOS application includes the required Afaria headers SeedDataAPI.h and SeedingAPISynchronous.h.
- The iOS application has been registered with the Afaria server as an application policy and made available to the iOS client device.

In Sybase Unwired Platform, a certificate can be used for authentication by creating a LoginCertificate object (the SUPLoginCertificate class), and setting that as the certificate

property in the client's synchronization profile. The login certificate has two properties that are used in authentication; the subjectCN (the common name of the certificate) and the signedCertificate (the certificate data itself).

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

```
+ (NSInteger)retrieveCertificateWithPrivateKey:
(SecKeyRef)privateKey andPublicKey:(SecKeyRef)publicKey
andCommonName:(NSString *)commonName andChallenge:(NSString
*)challengeCode forUrlScheme:(NSString *)urlScheme inCertificate:
(SecCertificateRef *)certificate;


+ (NSInteger)retrieveCertificateWithUrl:(NSURL *)url andPrivateKey:
(SecKeyRef)privateKey andPublicKey:(SecKeyRef)publicKey
andCommonName:(NSString *)commonName andChallenge:(NSString
*)challengeCode inCertificate:(SecCertificateRef *)certificate;
```

After this, the application will have a SecCertificateRef with the certificate, and a SecKeyRef with the private key. The certificate data in the SecCertificateRef cannot be used as is in the signedCertificate property of an SUPLoginCertificate. The signedCertificate property value is expected to contain the certificate and a digest of the certificate in ASN.1 format. To create the signedCertificate property value:

This sample code shows how to get the Afaria certificate, create an `SUPLoginCertificate` object, and attach it to a Sybase Unwired Platform synchronization profile.

```
// At this point, an Afaria user should have a signed certificate and
a private key available after importing
// their certificate using either of the Afaria APIs
 /*

+ (NSInteger)retrieveCertificateWithPrivateKey:
(SecKeyRef)privateKey andPublicKey:(SecKeyRef)publicKey
andCommonName:(NSString *)commonName andChallenge:(NSString
*)challengeCode forUrlScheme:(NSString *)urlScheme inCertificate:
(SecCertificateRef *)certificate;

+ (NSInteger)retrieveCertificateWithUrl:(NSURL *)url andPrivateKey:
(SecKeyRef)privateKey andPublicKey:(SecKeyRef)publicKey
andCommonName:(NSString *)commonName andChallenge:(NSString
*)challengeCode inCertificate:(SecCertificateRef *)certificate;

SecCertificateRef certificate;
SecKeyRef privatekey;

*/

SUPLoginCertificate *loginCertificate = [SUPLoginCertificate
getInstance];

loginCertificate.subjectCN =
```

```
(NSString*)SecCertificateCopySubjectSummary(certificate);

loginCertificate.signedCertificate = [CertBlobUtility
makeCertBlob:certificate andPrivateKey:privatekey];

NSLog(@"Certificate created. Subject =
%@",loginCertificate.subjectCN);

NSLog(@"MD5 digest = %@",[CertBlobUtility
md5sum:loginCertificate.signedCertificate]);

NSLog(@"SHA1 digest = %@",[CertBlobUtility
sha1:loginCertificate.signedCertificate]);


// Attach certificate to sync profile

SUPConnectionProfile *syncProfile = [SAPSSOCertTestSAPSSOCertTestDB
getSynchronizationProfile];
syncProfile.certificate = loginCertificate;
[loginCertificate release];
```

CertBlobUtility Header

The CertBlob Utility header of the CertBlob class.

```
#import <Foundation/Foundation.h>
#import <Security/Security.h>

@interface CertBlobUtility : NSObject

// Returns the MD5 sum of the input data
+ (NSString*)md5sum:(NSData*)certData;

// Returns the SHA1 fingerprint of the input data
+ (NSString*)sha1:(NSData*)certData;

// Given a signed certificate and private key, return a certificate
blob suitable for use in an SUPLoginCertificate
+ (NSData *)makeCertBlob:(SecCertificateRef)certificate
andPrivateKey:(SecKeyRef)privateKey;
@end
```

CertBlobUtility Source

The CertBlob Utility source of the CertBlob class.

```
#import "CertBlobUtility.h"
#import <CommonCrypto/CommonDigest.h>


bool getAsn1LengthBytes(
  int iLengthVal,  // (IN) value to be encoded
  unsigned char* pbOut,     // (IN/OUT) buffer to be populated with
the encoding or NULL to get sizing information
  int *iOutLen    // (IN/OUT) if pbOut != NULL, size of pbOut buffer
in allocated bytes.  Is set to the number
```

```
  // of bytes required/written in the encoding on return.
);

bool makeCertBlob(
  unsigned char* pbCert, // Certificate to be encoded in the CertBlob
  int iCertLen, // Length in bytes of pbCert
  unsigned char* pbSig,  // Signature to be encoded in the CertBlob
  int iSigLen,  // Length in bytes of pbSig
  unsigned char byteAlgorithm, // Algorithm constant to be encoded in
the CertBlob
  unsigned char* pbOut, // (IN/OUT) buffer to be populated with the
encoding or NULL to get sizing information
  int *iOutLen // (IN/OUT) if pbOut != NULL, size of pbOut buffer in
allocated bytes.  Is set to the number
  // of bytes required/written in the encoding on return.
);


bool getAsn1LengthBytes(
  int iLengthVal,  // (IN) value to be encoded
  unsigned char* pbOut,     // (IN/OUT) buffer to be populated with
the encoding or NULL to get sizing information
  int *iOutLen    // (IN/OUT) if pbOut != NULL, size of pbOut buffer
in allocated bytes.  Is set to the number
  // of bytes required/written in the encoding on return.
)
{
  // simple short form length
  if ( iLengthVal < 0x80 )
  {
    if ( ( pbOut != NULL ) && ( *iOutLen < 1 ) )
      return false;

    *iOutLen = 1;
    if ( pbOut != NULL )
      *pbOut = (unsigned char) iLengthVal;
    return true;
  }

  // if we got here, we need long form, because the short form doesn't
fit in a single byte

  // count the number of bytes in iVal
  int iTmp = iLengthVal;
  int iCount = 0;
  iTmp = iLengthVal;
  unsigned char byteLast = 0;
  while ( iTmp != 0 )
  {
    iCount++;
    byteLast = (unsigned char) ( iTmp & 0xFF );
    iTmp >>= 8;
  }

  // case where caller wants to know how to size buffer
  if ( NULL == pbOut )
```

```
  {
    *iOutLen = iCount + 1; // +1 for the length byte
      return true;
  }

  if ( *iOutLen < iCount + 1 )
    return false;

  *iOutLen = iCount + 1; // +1 for the length byte

  // Create an array with the count of bytes, followed by the iVal
bytes
  // Setting the top bit of the count indicates that this is a count
with the value to follow, not the actual integer value
  pbOut[ 0 ] = (unsigned char) ( iCount | 0x80 );  // count
  iTmp = iLengthVal;
  while ( iTmp != 0 )
  {
    unsigned char b = (unsigned char) ( iTmp & 0xFF );
    iTmp >>= 8;
    pbOut[ iCount-- ] = b;
  }

  return true;
}

// makeCertBlob "C" function used by SSOCertManager makeCertBlob
method below
/*
 * Returns a buffer containing an ASN.1 encoding for a CertBlob.
 * Upon return, pbOut will be filled with the result and
 * iOutLen will contain the number of bytes written.  If this
 * function is called with NULL as the pbOut pointer, it will
 * populate iOutLen without writing anything.  The expected usage
 * is to call with pbOut==NULL to size the buffer, allocate the
buffer,
 * then call it again with the newly allocated buffer.
 *
 * Return value of false is if pbOut!=NULL and the passed in iOutLen
 * is less than the required number of bytes to write the result.
 *
 */
bool makeCertBlob(
  unsigned char* pbCert, // Certificate to be encoded in the CertBlob
  int iCertLen, // Length in bytes of pbCert
  unsigned char* pbSig,  // Signature to be encoded in the CertBlob
  int iSigLen,  // Length in bytes of pbSig
 unsigned char byteAlgorithm, // Algorithm constant to be encoded in
the CertBlob
  unsigned char* pbOut, // (IN/OUT) buffer to be populated with the
encoding or NULL to get sizing information
  int *iOutLen // (IN/OUT) if pbOut != NULL, size of pbOut buffer in
allocated bytes.  Is set to the number
  // of bytes required/written in the encoding on return.
  )
{
```

```
  int iCertLenLen, iSigLenLen;
  int iAlgorithmLen = 2;

  // get number of bytes in length descriptors
  if ( !getAsn1LengthBytes( iCertLen, NULL, &iCertLenLen ) )
    return false;

  if ( !getAsn1LengthBytes( iSigLen, NULL, &iSigLenLen ) )
    return false;

  // calculate size of content of sequence
  int iSeqLen = 1 +  // type code for OCTET STRING
  iCertLenLen +   // length bytes for Certificate
  iCertLen +      // data bytes for Certificate
  1 +             // type code for OCTET STRING
  iSigLenLen +    // length bytes for Signature
  iSigLen +       // data bytes for Signature
  1 +             // type code for INTEGER
  iAlgorithmLen;  // data bytes for algorithm (assumed to be an
integer that fits in a single byte)

  // now calculate size of outer sequence
  int iSeqLenLen;
  if ( !getAsn1LengthBytes( iSeqLen, NULL, &iSeqLenLen ) )
    return false;

  int iTotalLen = 1 +           // type code for SEQUENCE
  iSeqLenLen +  // length bytes for Sequence
  iSeqLen;      // data bytes for Sequence

  if ( NULL == pbOut )
  {
    // caller is just asking for required buffer size
    *iOutLen = iTotalLen;
      return true;
  }

  // test whether buffer is large enough
  if ( *iOutLen < iTotalLen )
    return false;

  // write everything to the buffer
  int iCurIdx = 0;

  // header bytes for wrapping sequence
  pbOut[ iCurIdx++ ] = (unsigned char) 0x30;  // type code for
SEQUENCE
  if ( !getAsn1LengthBytes( iSeqLen, pbOut + iCurIdx,
&iSeqLenLen ) )   // length bytes for Sequence
    return false;
  iCurIdx += iSeqLenLen;

  // first element of sequence -> certificate
  pbOut[ iCurIdx++ ] = (unsigned char) 0x04;  // type code for OCTET
STRING
  if ( !getAsn1LengthBytes( iCertLen, pbOut + iCurIdx,
```

```
&iCertLenLen ) )  // length bytes for Certificate
    return false;
  iCurIdx += iCertLenLen;
  memcpy( pbOut + iCurIdx, pbCert, iCertLen );  // bytes for
Certificate
  iCurIdx += iCertLen;

  // second element of sequence -> signature
  pbOut[ iCurIdx++ ] = (unsigned char) 0x04;  // type code for OCTET
STRING
  if ( !getAsn1LengthBytes( iSigLen, pbOut + iCurIdx,
&iSigLenLen ) )  // length bytes for Certificate
    return false;

  iCurIdx += iSigLenLen;
  memcpy( pbOut + iCurIdx, pbSig, iSigLen );  // bytes for
Certificate
  iCurIdx += iSigLen;

  // third element of sequence -> algorithm
  pbOut[ iCurIdx++ ] = (unsigned char) 0x02;  // type code for INTEGER
  pbOut[ iCurIdx++ ] = (unsigned char) 0x01;  // length bytes for
value (assume 1)
  pbOut[ iCurIdx++ ] = byteAlgorithm;  // algorithm constant

    return true;
}

@implementation CertBlobUtility

+ (NSString*)md5sum:(NSData*)certData
{

  CC_MD5_CTX md5;

  CC_MD5_Init(&md5);

  CC_MD5_Update(&md5, [certData bytes], [certData length]);

    unsigned char digest[CC_MD5_DIGEST_LENGTH];

    CC_MD5_Final(digest, &md5);

    NSString* s = [NSString stringWithFormat: @"%02x%02x%02x%02x%02x
%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x",
      digest[0], digest[1],
      digest[2], digest[3],
      digest[4], digest[5],
      digest[6], digest[7],
      digest[8], digest[9],
      digest[10], digest[11],
      digest[12], digest[13],
      digest[14], digest[15]];
    return s;
}
```

```
+ (NSString*)sha1:(NSData*)certData {
  unsigned char sha1Buffer[CC_SHA1_DIGEST_LENGTH];
  CC_SHA1(certData.bytes, certData.length, sha1Buffer);
  NSMutableString *fingerprint = [NSMutableString
stringWithCapacity:CC_SHA1_DIGEST_LENGTH * 3];
  for (int i = 0; i < CC_SHA1_DIGEST_LENGTH; ++i)
    [fingerprint appendFormat:@"%02x ",sha1Buffer[i]];
  return [fingerprint stringByTrimmingCharactersInSet:
[NSCharacterSet whitespaceCharacterSet]];
}

// SSOCertManager makeCertBlob: used by getCertBlob: API below
// Makes a certBlob from given certificate and private key and
returns it
+ (NSData *)makeCertBlob:(SecCertificateRef)certificate
andPrivateKey:(SecKeyRef)privateKey {
    NSData *sigData;
    NSData *certData;

    CFDataRef certCFData = SecCertificateCopyData(certificate);
    unsigned char certDigest[CC_SHA1_DIGEST_LENGTH];
    CC_SHA1( CFDataGetBytePtr(certCFData),
CFDataGetLength(certCFData), certDigest );

    certData = [NSData dataWithBytes:CFDataGetBytePtr(certCFData)
length:CFDataGetLength(certCFData)];

    size_t sigLen = 1024;
    uint8_t sigBuf[sigLen];

    // Encrypt the digest of the certificate with private key
    OSStatus err = SecKeyRawSign(privateKey, kSecPaddingPKCS1,
                                 certDigest,
CC_SHA1_DIGEST_LENGTH, //data.bytes, data.length,
                                 sigBuf, &sigLen);

    if (err == noErr) {
        sigData = [NSData dataWithBytes:sigBuf length:sigLen];
    }
    if ( certCFData != NULL )
        CFRelease(certCFData);

    if ( ( certData == nil ) || ( sigData == nil ) )
        return nil;

    int iLength = 0;
    if ( ( !makeCertBlob( (unsigned char *)[certData bytes],
[certData length], (unsigned char *)[sigData bytes], [sigData
length], 1, NULL, &iLength ) ) || ( iLength == 0 ) )
        return nil;

    unsigned char* pBuf = (unsigned char*)malloc(iLength);
   if ( !makeCertBlob( (unsigned char *)[certData bytes], [certData
length], (unsigned char *)[sigData bytes], [sigData length], 1, pBuf,
&iLength ) ) {
        free( pBuf );
```

```
        return nil;
    }

    NSData* certBlob = [NSData dataWithBytes:pBuf length:iLength];
    free( pBuf );

    return certBlob;
}


@end
```

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 setPageSize:4*1024];
[cp setEncryptionKey:@"Your key of more than 16 characters"];
// Immediately after the call to setEncryptionKey, call
[SUP101SUP101DB closeConnection]; to ensure that old connections
with the wrong key are no longer being used.
[SUP101SUP101DB closeConnection];
```

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

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

You can use the `cacheSize` API to control the size of the memory cache used by the database. The default size is 10MB.

```
SUPConnectionProfile *cp = [SUP101SUP101DB getConnectionProfile];
[cp setCacheSize:5000000];  // set to 5000000 bytes (~ 5 MB)
[SUP101SUP101DB closeConnection];  // close and open the connection
to use the new cache size
[SUP101SUP101DB openConnection];
```

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 can enter the related configuration in the application connection template through Sybase Control Center, and/or setup the configuration properties in the synchronization profile using the object API.

If a Relay Server is used, the 'companyID' in the `SUPApplication` property must correspond to the MBS farm ID that is used for the messaging client connection.

```
SUPConnectionProperties props = app.connectionProperties;
[props setFarmId:@"relayServer1"];
```

For data synchronization through a Relay Server, synchronization properties need to be set if the corresponding application connection template in Sybase Control Center does not have with the required values:

- Add the certificate file provided by the Relay Server to the `Resource` folder of your Xcode project.
- Add the following code before calling `[SUP101SUP101DB subscribe]`:

  ```
  SUPConnectionProfile *sp = [SUP101SUP101DB
  getSynchronizationProfile];
  [sp setUser:@"xxxx"]; //required
  [sp setPassword:@"xxxx"]; //required
  [sp setNetworkProtocol:@"https"]; // or http, optional
  [sp setPortNumber:443]; // if http then corresponding port,
  optional
  [sp
  setNetworkStreamParams:@"trusted_certificates=certificateName;com
  pression=zlib;url_suffix=urlsuffixProvidedByTheRelayServer"]; //
  optional
  ```

  - **NetworkProtocol –** http or https.
  - **PortNumber –** the correct port number for the selected NetworkProtocol.
  - **NetworkStreamParams –** certificateName: the name of the certificate you added in the Resource folder.

    urlsuffixProvidedByTheRelayServer: the RBS URL suffix provided by the Relay Server.

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

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 non-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:

- – set the user name and password in the connection profile and authenticate credentials against the Unwired Server.

```
SUPConnectionProfile *syncProfile = [SUP101SUP101DB
getSynchronizationProfile];
[syncProfile setUser:@"user"];
[syncProfile setPassword:@"password"];
[SUP101SUP101DB onlineLogin];
```

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 the Unwired Server. For example, `c:\certs`.
2. Create a registry string value on the 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 the 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 the Unwired Server runs must have adequate permissions to impersonate the domain\user. For example, the Administrator account for the domain.
6. Include code that imports the certificate from the Unwired Server, and sets up the login credentials for the package.

```
// 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];

while ([SUPApplication connectionStatus] !=
SUPConnectionStatus_CONNECTED) {
  NSLog(@"waiting to connect...");
  sleep(2);
}
```

7. Perform a database `subscribe` and `synchronize` as appropriate.

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, call the `subscribe` and `synchronize` methods..

```
[SUP101SUP101DB subscribe];
[SUP101SUP101DB synchronize];
```

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 the client and the 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 the 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 the 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
SUP101PersonalizationParameters *pp = [SUP101SUP101DB
getPersonalizationparameters];
MBOLogInfo(@"Personalization Parameter for City = %@", pp.PKCity);

//Set personalization key values
pp.PKCity = @"Hull";
[pp.save]; //save the new pk value.
[SUP101SUP101DB synchronize];
```

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.

Managing Synchronization Parameters

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

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

To add a synchronization parameter:

```
SKPKCustomerSubscription *sp = [SKPKCustomerSubscription
getInstance];
```

```
sp.name = @"example";
[SKPKCustomer addSubscription:sp];
```

To list all synchronization parameters:

```
SUPObjectList* r = [SKPKCustomer getSubscriptions];
```

To remove a synchronization parameter:

```
SUPObjectList* r = [SKPKCustomer getSubscriptions];
SKPKCustomerSubscription* sub = (SKPKCustomerSubscription*)[r item:
0];
[SKPKCustomer removeSubscription:sub];
```

Performing Mobile Business Object Synchronization

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

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:(NSString*)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.

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

beginOnlineLogin

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

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

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

In RBS, `beginSynchronize` creates a synchronize request, and puts it in the request queue; the synchronization thread processes the sync request, and does the synchronization automatically in the background. The synchronization thread can combine several synchronization requests and send them to the server. For each synchronization request, a SUPSynchronizationStatus_STARTING status is sent to the `onSynchronize` user callback function before the synchronization, and a SUPSynchronizationStatus_FINISHING status is sent to `onSynchronize` after the synchronization.

Syntax

```
+(void) beginSynchronize

+(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];
  ```

Push Synchronization Applications

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

Sybase Unwired Platform uses a messaging channel to send change notifications from the server to the client device. By default, change notification is disabled. You can enable the change notification of a synchronization group: If you see that setInterval is set to 0, then change detection is disabled, and notifications will not be delivered. Enable change detection and notification delivery by setting an appropriate value. For recommendations, see *Configuring Synchronization Groups* in *Sybase Control Center for Sybase Unwired Platform*.

```
id<SUPSynchronizationGroup> sg = [SUP101SUP101DB
getSynchronizationGroup:@"TCNEnabled"];
  if (![sg enableSIS]) {
      [sg setEnableSIS:YES];
      [sg setInterval:2];
      [sg save];
      [SUP101SUP101DB synchronize:@"PushEnabled"];
}
```

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

```
- (SUPSynchronizationActionType)onSynchronize:(SUPObjectList
*)syncGroupList withContext:(SUPSynchronizationContext *)context
{

  switch ([context status]) {
    case SUPSynchronizationStatus_STARTING_ON_NOTIFICATION:

      if(allowBackGroundSync)
      {
        return SUPSynchronizationAction_CONTINUE;
      }
      else
      {
        return SUPSynchronizationAction_CANCEL;
      }

      break;

    default:
       return SUPSynchronizationAction_CONTINUE; // return continue
for all other cases
      break;
  }
```

```
}
```

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.

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

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

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 = [SUP101Sales_order findAll];
  if([salesorders size] > 0)
  {
    SUP101Sales_order * so = [salesorders item:0];
    SUP101LogRecordImpl *lr = [SUP101LogRecordImpl getInstance];
```

```
    lr.message = :@"testing record"];
    lr.level = [SUPLogLevel INFO];
    [lr save];

    // submitting log records
    [SUP101SUP101DB submitLogRecords];
    // synchronize with server
    [SUP101SUP101DB synchronize:@"system"];
  }
}
```

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

```
 // Retrieve SUPLogger from the database class
SUPLogger logger = [SMP101DB getLogger];

// Set the log level for the logger
// Application can use getLogLevel to determine the current log level
setting for the Logger
[logger setLogLevel:[SMPLogLevel DEBUG]];

// create a log record at DEBUG level
[logger debug:@"Some debug message"];

// Prepare all outstanding client generated log records for upload
[logger submitLogRecords];
```

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 getInstance];
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);
  }
```

Change Log API

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

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

entityType

Returns the entity type.

Syntax

```
- (int)entityType
```

Parameters

None.

Returns

Returns the entity type.

Examples

- **Get the Entity Type**

```
SUPObjectList *changeLogs = [SUP101SUP101DB getChangeLogs:
[SUPQuery getInstance]];
if([changeLogs size] > 0)
{
    for (id<SUPChangeLog> cl in changeLogs)
    {
       MBOLogInfo(@"changelog:[entityType=%d]\n", [cl
entityType]);
    }
}
```

operationType

Returns the operation type of the MBO.

Syntax

```
- (unichar)operationType
```

Parameters

None.

Returns

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

Examples

- **Get the Operation Type**

```
SUPObjectList *changeLogs = [SUP101SUP101DB getChangeLogs:
[SUPQuery getInstance]];
if([changeLogs size] > 0)
{
    for (id<SUPChangeLog> cl in changeLogs)
    {
```

```
       MBOLogInfo(@"changelog:[operationType=%C]\n", [cl
operationType]);
    }
}
```

rootEntityType

Returns the name of the root parent entity type.

Syntax

```
- (int)rootEntityType
```

Parameters

None.

Returns

Returns the root entity type which is the root of the object graph.

Examples

- **Get the Root Entity Type**

```
SUPObjectList *changeLogs = [SUP101SUP101DB getChangeLogs:
[SUPQuery getInstance]];
if([changeLogs size] > 0)
{
    for (id<SUPChangeLog> cl in changeLogs)
    {
       MBOLogInfo(@"changelog:[rootEntityType=%d]\n", [cl
rootEntityType]);
    }
}
```

rootSurrogateKey

Returns the surrogate key of the root parent entity.

Syntax

```
- (long)rootSurrogateKey
```

Parameters

None.

Returns

The surrogateKey of the root entity.

Examples

- **Get the Root Surrogate Key**

```
SUPObjectList *changeLogs = [SUP101SUP101DB getChangeLogs:
[SUPQuery getInstance]];
if([changeLogs size] > 0)
{
    for (id<SUPChangeLog> cl in changeLogs)
    {
       MBOLogInfo(@"changelog:[rootSurrogateKey=%ld]\n", [cl
rootSurrogateKey]);
    }
}
```

surrogateKey

Returns the surrogate key of the entity.

Syntax

```
- (long)surrogateKey
```

Parameters

None.

Returns

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

Examples

- **Get the Surrogate Key**

```
SUPObjectList *changeLogs = [SUP101SUP101DB getChangeLogs:
[SUPQuery getInstance]];
if([changeLogs size] > 0)
{
    for (id<SUPChangeLog> cl in changeLogs)
    {
       MBOLogInfo(@"changelog:[surrogateKey=%ld]\n", [cl
surrogateKey]);
    }
}
```

Methods in the Generated Database Class

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

enableChangeLog

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

Syntax

```
+ (void) enableChangeLog
```

Returns

None.

Examples

- **Enable Change Log**

  ```
  [SUP101SUP101DB enableChangeLog];
  ```

getChangeLogs

Retrieve a list of change logs.

Syntax

```
+ (SUPObjectList*) getChangeLogs:(SUPQuery*)query
```

Returns

Returns a GenericList of type <Change Log>.

Examples

- **Get Change Logs**

  ```
  SUPObjectList *changeLogs = [SUP101SUP101DB getChangeLogs:
  [SUPQuery getInstance]];
  ```

deleteChangeLogs

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

Syntax

```
+ (void) deleteChangeLogs
```

Returns

None.

Examples

- **Delete Change Logs**

```
[SUP101SUP101DB deleteChangeLogs];
```

Usage

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

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

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

disableChangeLog

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

Syntax

```
+  (void) disableChangeLog
```

Returns

None.

Examples

- **Disable Change Log**

```
[SUP101SUP101DB.disableChangeLog];
```

Code Samples

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

```
// Retrieve all change logs
[SUP101SUP101DB enableChangeLog];
[SUP101SUP101DB synchronize];
SUPObjectList *changeLogs = [SUP101SUP101DB getChangeLogs:[SUPQuery
getInstance]];
if([changeLogs size] > 0)
```

```
{
    for (id<SUPChangeLog> cl in changeLogs)
    {
       MBOLogInfo(@"changelog:[entityType=%d]\n", [cl entityType]);
    }
}

// Retrieve only the change logs for Customer:
[SUP101SUP101DB enableChangeLog];
[SUP101SUP101DB synchronize];

SUPQuery *changelogQuery = [SUPQuery getInstance];
int entityInt = SUP101_ENTITY_TYPE_Customer;
changelogQuery.testCriteria = [SUPAttributeTest
equal:@"entityType" :[NSNumber numberWithInt:entityInt]];

SUPObjectList *changeLogs = [SUP101SUP101DB
getChangeLogs:changelogQuery ];
if([changeLogs size] > 0)
{
  for (id<SUPChangeLog> cl in changeLogs)
  {
    MBOLogInfo(@"changelog:[entityType=%d]\n", [cl entityType]);
  }
}
```

Security APIs

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

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];
```

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`.

By linking the `libDatavault.a` static library, 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 arbitrary length and can include any characters. The password and salt together generate the AES key. If the user enters the same password when unlocking, the contents are decrypted. If the user enters an incorrect password, exceptions occur. If the user enters an incorrect password a configurable number of times, the vault is deleted and any data stored within it becomes unrecoverable. The vault can also relock itself after a configurable amount of time.

Typical usage of the `SUPDataVault` is to implement an application login screen. Upon application start, the user is prompted for a password, which unlocks the vault. If the unlock attempt is successful, the user is allowed into the rest of the application. User credentials for synchronization can also be extracted from the vault so the user need not reenter passwords.

createVault

Creates a new secure store (a vault).

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

Syntax

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

Parameters

- **name –** an arbitrary name for a `DataVault` instance on this device. This name is effectively the primary key for looking up `DataVault` instances on the device, so it cannot use the same name as any existing instance. If it does, this method throws an exception with error code INVALID_ARG. The name also cannot be empty or null.
- **password –** the initial encryption password for this DataVault. This is the password needed for unlocking the vault. If null is passed, a default password is computed and used.
- **salt –** the encryption salt value for this DataVault. This value, combined with the password, creates the actual encryption key that protects the data in the vault. If null is passed, a default salt is computed and used.

Returns

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

If a vault already exists with the same name, a `SUPDataVaultException` is thrown 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

Returns true if the vault exists; otherwise returns false.

Examples

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

  ```
  if ([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.

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]);
}
```

getDataNames

Retrieves information about the data names stored in the vault.

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

Syntax

```
- (SUPObjectList *)getDataNames;
```

Parameters

None.

Returns

Returns a list of objects of type `SUPDVDataName`.

Examples

- **Get data names**

```
// Call getDataNames to retrieve all stored element names from our
data vault
NSArray *dataNames = [dataVault getDataNames];
if (dataNames != nil) {
  DVDataName *dataName;
  for (NSInteger iIdx = 0; iIdx < [dataNames count]; iIdx++) {
    dataName = [dataNames objectAtIndex:iIdx];
```

```
    if (dataName.type == kDVDataTypeString) {
      // Stored value is of string type
      NSString *thisStringValue = [dataVault
getString:dataName.name];
    }
    else if (dataName.type == kDVDataTypeBinary) {
      // Stored value is of binary type
      NSData *thisBinaryValue = [dataVault
getValue:dataName.name];
    }
    else {
      // Unknown type. Possibly stored using previous version of
dataVault
      // Try as string first and then as binary
      NSString *thisStringValue = [dataVault
getString:dataName.name];
      if (thisStringValue == nil) {
        NSData *thisBinaryValue = [dataVault
getValue:dataName.name];
      }
    }
  }
}
```

setPasswordPolicy

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

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

Syntax

```
- (void)setPasswordPolicy:SUPDVPasswordPolicy oPasswordPolicy;
```

Parameters

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

Examples

- **Set a password policy**

```
// setPasswordPolicy locks the vault to ensure the old password
conforms to the new password policy settings
[dataVault setPasswordPolicy:pwdPolicy];
```

Password Policy Structure

A structure defines the policy used to generate the password.

Table 2. Password Policy Structure

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

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

Settings for Password Policy

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

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

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

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

Password Errors

Password policy violations cause exceptions to be thrown.

Table 3. Password Errors

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

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

getPasswordPolicy

Retrieves the password policy set by `setPasswordPolicy`.

Use this method once the DataVault is unlocked.

Syntax

```
+ (SUPDataVault*)getPasswordPolicy:();
```

Parameters

None.

Returns

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

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

Examples

- **Get the current password policy**

```
// Use getPasswordPolicy to get the current policy set in the
vault
      pwdPolicy = [dataVault getPasswordPolicy];
```

lock

Locks the vault.

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

Syntax

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

Examples

- **Locks the data vault –** prevents changing the vaults properties or stored content.
```
[oVault lock];
```

isLocked
Checks whether the vault is locked.

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

Returns

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 attempts exceeds the retry limit, the vault is deleted.

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

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

Parameters

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

Returns

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

Examples

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

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

setString

Stores a string object in the vault.

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 ensures 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.

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

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

Examples

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

```
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.

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 ensures 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.

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

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

Examples

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

```
NSData *retrievedvalue = nil;

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

deleteValue

Deletes the specified value.

Syntax

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

Parameters

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

Examples

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

```
[SUPDataVault deleteValue:@"myValue"];
```

changePassword (two parameters)

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

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

Syntax

```
- (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 ensures 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];
}
```

changePassword (four parameters)

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

This overloaded method ensures the new password is compatible with the password policy, uses the current password to unlock the vault, and changes the password of the vault to a new

password. If the current password is not valid an `InvalidPassword` exception is thrown. If the new password is not compatible with the password policy set in `setPasswordPolicy` then an `IncompatiblePassword` exception is thrown.

Syntax

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

Parameters

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

Examples

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

```
// Call changePassword with four parameters, even if the vault is
locked.
// Pass null for oldSalt and oldPassword if the defaults were
used.
[dataVault changePassword:nil currentSalt:nil
newPassword:@"password!1A" newSalt:@"saltD#ddg#k05%gnd[!1A"];
```

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"];
```

Code Sample

Create a data vault for encrypted storage of application data.

```
SUPDataVault* dataVault = nil;
@try
{
  // If the dataVault already exists, call getVault and unlock it
  // If not, create the vault with necessary password
  // The password is chosen to make sure it satisfies password policy
criteria given below
  if ( [SUPDataVault vaultExists:@"SampleVault"] ) {
    dataVault = [SUPDataVault getVault:@"SampleVault"];
    [dataVault unlock:@"password!1A" withSalt:@"saltD#ddg#k05%gnd[!
1A"];
  }
  else {
    dataVault = [SUPDataVault createVault:@"SampleVault"
withPassword:@"password!1A"  withSalt:@"saltD#ddg#k05%gnd[!1A"];
  }

  // Supply various criteria for password policy
  SUPDVPasswordPolicy *pwdPolicy = [[[SUPDVPasswordPolicy alloc]
init] autorelease];
  pwdPolicy.defaultPasswordAllowed = YES;
  pwdPolicy.minLength = 4;
  pwdPolicy.hasDigits = YES;
  pwdPolicy.hasUpper = YES;
  pwdPolicy.hasLower = YES;
  pwdPolicy.hasSpecial = YES;
  pwdPolicy.expirationDays = 20;
  pwdPolicy.minUniqueChars = 3;
  pwdPolicy.lockTimeout = 1600;
  pwdPolicy.retryLimit = 20;

  // setPasswordPolicy will lock the vault to ensure old password
conforms to new password policy settings
  [dataVault setPasswordPolicy:pwdPolicy];

  // You must unlock the vault after setting the password policy
  [dataVault unlock:@"password!1A" withSalt:@"saltD#ddg#k05%gnd[!
1A"];

  // Use getPasswordPolicy to get the current policy set in the vault
```

```
  pwdPolicy = [dataVault getPasswordPolicy];
  NSLog(@" pwdPolicy %@ ",pwdPolicy.description);

  // Call setString by giving it a name:value pair to encrypt and
persist
  // a string data type within your dataVault.
  [dataVault setString:@"stringName" withValue:@"stringValue"];

  // Call getString to retrieve the string we just stored in our data
vault!
  NSString *storedStringValue = [dataVault getString:@"stringName"];
  NSLog(@" storedStringValue %@ ",storedStringValue.description);
  // Call setValue by giving it a name:value pair to encrypt and
persist
  // a binary data type within your dataVault unsigned char
acBinData[] = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07 };
  [dataVault setValue:@"binaryName" withValue:[NSData
dataWithBytes:acBinData length:7]];

  // Call getValue to retrieve the binary we just stored in our data
vault!
  NSData *storedBinaryValue = [dataVault getValue:@"binaryName"];

  NSLog(@" storedBinaryValue %@ ",storedBinaryValue );

  // Call getDataNames to retrieve all stored element names from our
data vault
  //        NSArray * dataNames = [dataVault getDataNames];

  SUPObjectList * dataNames = [dataVault getDataNames];

  if ( dataNames != nil ) {
    SUPDVDataName *dataName;
    //           for ( NSInteger iIdx = 0; iIdx < [dataNames count];
iIdx++ ) {
    for ( NSInteger iIdx = 0; iIdx < [dataNames size]; iIdx ++) {
      dataName = [dataNames objectAtIndex:iIdx];
      if ( dataName.type == SUPDVDataTypeString ) {
        // Stored value is of string type
        NSString *thisStringValue = [dataVault
getString:dataName.name];
        NSLog(@" thisStringValue %@ ",thisStringValue );
      }
      else if ( dataName.type == SUPDVDataTypeBinary ) {
        // Stored value is of binary type
       NSData *thisBinaryValue = [dataVault getValue:dataName.name];
        NSLog(@" thisBinaryValue %@ ",thisBinaryValue );
      }
      else {
        // Unknown type. Possibly stored using previous version of
dataVault
        // Try as string first and then as binary
        NSString *thisStringValue = [dataVault
getString:dataName.name];
        if ( thisStringValue == nil ) {
          NSData *thisBinaryValue = [dataVault
```

```
getValue:dataName.name];
          NSLog(@" thisBinaryValue %@ ",thisBinaryValue );
        }
      }
    }
  }

  [dataVault changePassword:@"password!2A"
withSalt:@"saltD#ddg#k05%gnd[!2A"];


  // Because this is a test example, we will delete our vault at the
end.
  // This means we will forever lose all data we persisted in our data
vault.
  [SUPDataVault deleteVault: @"SampleVault"];
}
@catch (DataVaultException *exception)
{
  NSLog(@"Datavault exception. Reason: %@", [exception reason]);
}
```

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 `SUPCallbackHandler` protocol 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, `onReplaySuccess`).

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, `onReplaySuccess` 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:

- **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). The `onReplaySuccess:(id)entityObject` is an MBO object instance that contains the data prior to the synchronization. You can use the Change Log API to find records that occur after the synchronization.
- **onImport:(id)entityObject; –** invoked when an `import` is received. If the 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. This method is for DOE-based applications only.
- **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 `synchronize` or `beginSynchronize` methods when the client initiates a synchronization, and is called again when the server responds to the client that synchronization has finished, or that synchronization failed.

  The `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 `synchronize` or `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.
  - **SUPSynchronizationStatus_ASYNC_REPLAY_UPLOADED –** asynchronous replay has been uploaded.

- **SUPSynchronizationStatus_ASYNC_REPLAY_COMPLETED –** asynchronous replay has been completed.
- **SUPSynchronizationStatus_STARTING_ON_NOTIFICATION –** change notification has been received from the server.

For DOE-based applications, 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. This method is for DOE-based applications only.
- **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`.
- **- (void)onMessageStart:(int)size withMethod:(NSString*)method withMbo: (NSString*)mbo; –** This method is for DOE-based applications only.

  This method is called at the beginning of processing a message from the server, before the message transaction starts. Only the callback handler registered with the package database class is invoked. Parameters:

  - **size** – The size of the incoming message content in bytes.
  - **method** – The method string from the message header.
  - **mbo** – If this message is for a specific MBO, the name of the MBO; otherwise null.

This 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];
```

SUPApplicationCallback API

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

You must register an `SUPApplicationCallback` implementation to your SUPApplication instance to receive these callbacks.

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

Table 4. Callbacks in the SUPApplicationCallback Interface

Callback	Description
`- (void)onApplicationSetting-sChanged :(SUPString-List*)names`	Invoked when one or more application settings have been changed by the server administration.

Callback	Description
`- (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. **Note:** Some of the connection status codes are not returned on certain client platforms due to platform operating system limitations.
`- (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 `SUPDeviceCondition` class.
`- (void)onRegistrationStatusChanged :(SUPInt)registrationStatus :(SUPInt)errorCode :(SUPNullableString)errorMessage`	Invoked when the registration status changes. The possible registration status values are defined in the `SUPRegistrationStatus` class.
`- (void)onHttpCommunicationError :(int32_t)errorCode :(NSString*) errorMessage :(SUPStringProperties*)responseHeaders;`	Invoked when an HTTP communication server/MobiLink rejects HTTP/MobiLink communication with an error code. • **errorCode –** Error code returned by the HTTP server or MobiLink. For example: code 401 for authentication failure, code 403 for authorization failure, and code 63 for MobiLink synchronization communication error. • **errorMessage –** Error message returned by the HTTP server or MobiLink. • **responseHeaders –** Response headers returned by the HTTP server or MobiLink.

Callback	Description

`- (void)onCustomizationBundleDownloadComplete : (NSString*) customizationBundleID: (int32_t) errorCode : (NSString*) errorMessage;`	Invoked when the download of a resource bundle is complete.
	• **errorCode –** If download succeeds, returns 0. If download fails, returns an error code.
	• **errorMessage –** If download succeeds, returns "". If download fails, returns an error message.
	  • RESOURCE_BUNDLE_NOTFOUND = 14881
	  • DOWNLOAD_RESOURCE_BUNDLE_STREAM_IS_NULL = 14882
	  • DOWNLOAD_RESOURCE_BUNDLE_FAILURE = 14883
	• **customizationBundleID –** The name of the resource bundle. If null, the default application resource bundle is downloaded.

Callback	Description

`(int)onPushNotification : (NSDictionary*)notification`	Invoked if a push notification arrives. You can add logic here to handle the notification. This callback is not called when a notification arrives when the application is not online.
	• **returns –** an integer to indicate if the notification has been handled. The return value is for future use. You are recommended to return SUP_NOTIFICATION_CONTINUE.
	  • 0: SUP_NOTIFICATION_CONTINUE if the notification was not handled by the callback method.
	  • 1: SUP_NOTIFICATION_CANCEL if the notification has already been handled by the callback method.
	When iOS receives a notification from the Apple Push Notification Service for an application, it calls `didReceiveRemoteNotification` in the client application. Call the following API inside `didReceiveRemoteNotification`:
	`+(void)pushNotification:(UIApplication*)application notifyData:(NSDictionary *)userInfo`
	If
	`+(void)pushNotification:(UIApplication*)application notifyData:(NSDictionary *)userInfo`
	is added inside of `didReceiveRemoteNotification`, then only the callback method
	`(int)onPushNotification :(NSDictionary*)notification`
	is triggered.

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 of the `SUPPushNotification` interface abstract the Unwired Server, resolve the push-related settings, and register with an Apple Push server, if required.

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. Call the `deviceTokenForPush` and `setupForPush` methods inside the `didRegisterForRemoteNotificationWithDeviceToken` method, or after the method. For example, you can store the device token and application parameters in variables and use them later to call `deviceTokenForPush` and `setupForPush`.

```
+(void)setupForPush:(UIApplication*)application
+(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. Call the following API inside `didFailToRegisterForRemoteNotificationsWithError`:

```
+(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. Call the following API inside `didReceiveRemoteNotification`:

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

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 Sybase Unwired WorkSpace. Object Query methods carry query names, parameters, and return types defined in Sybase 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 5. 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]]);
}
```

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
- NOT_START_WITH
- NOT_END_WITH
- 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)
```

```
Sybase.Persistence.Query query = new Sybase.Persistence.Query();
query.Select("a.id");
query.From("AllType", "a");
Sybase.Persistence.AttributeTest test = new
Sybase.Persistence.AttributeTest();
Sybase.Persistence.Query existQuery = new
Sybase.Persistence.Query();
existQuery.Select("b.id");
existQuery.From("AllType", "b");
Sybase.Persistence.Column cl = new Sybase.Persistence.Column();
cl.Alias = "a";
cl.Attribute = "id";
Sybase.Persistence.AttributeTest test1 = new
Sybase.Persistence.AttributeTest();
test1.Attribute = "b.id";
test1.Value = cl;
test1.SetOperator(Sybase.Persistence.AttributeTest.EQUAL);
existQuery.Where(test1);
test.Value = existQuery;
test.SetOperator(Sybase.Persistence.AttributeTest.EXISTS);
```

```
query.Where(test);
Sybase.Persistence.QueryResultSet qs = SUP101DB.ExecuteQuery(query);
```

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 `select:` method from `SUPQuery`, 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.

```
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 `- (SUPQuery*)groupBy:(SUPString)items` method from `SUPQuery`. 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 `- (SUPQuery*)having:(SUPTestCriteria*)test method from SUPQuery` 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:

- `- (SUPCompositeQuery *)union:(SUPQuery *)otherQuery`
- `- (SUPCompositeQuery *)unionAll:(SUPQuery *)otherQuery`
- `- (SUPCompositeQuery *)except:(SUPQuery *)otherQuery`
- `- (SUPCompositeQuery *)intersect:(SUPQuery *)otherQuery`

Note: Sybase Unwired Platform adds a "LONG VARCHAR" column for all MBO tables. UltraLiteJ cannot select a "LONG VARCHAR" in a union query. Ensure that in the selected fields you do not use * in the `select` of a union 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 `- (SUPQuery*)from:(SUPString)entity:(SUPString)alias` method from `SUPQuery`. 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 `SUPCompositeTest`, `SUPSortCriteria`, and `SUPQuery` 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 = [SUP101Customer findWithQuery:props];
```

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

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.

```
SUP101Customer *onecustomer = [SUP101Customer find:101];
SUPObjectList *orders = onecustomer.salesOrders;
```

Given an order, you can access its customer information.

```
SUP101Sales_order * order = [SUP101Sales_order *find: 2001];
SUP101Customer *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:

```
SUP101Customer *newcustomer = [[SUP101Customer alloc] init];
newcustomer.fname = @”John”;
...  //Set the required fields for the customer
[newcustomer create];
[newcustomer submitPending];
[SUP101SUP101DB synchronize];
```

Example 2: Supports create operations on child entities.

```
SUP101Sales_Order *order = [[SUP101Sales_Order alloc] init];
[order autorelease];
//Set the other required fields for the order
order.region = @"Eastern";
order.xxx = yyy;
```

```
SUP101Customer *customer = [SUP101Customer find:1008];
[order setCustomer:customer];
[order create];
[order.customer refresh]; //refresh the parent
[order.customer submitPending];  //call submitPending on the parent.
[SUP101SUP101DB synchronize];
```

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:

```
SUP101Customer *customer = [ SUP101Customer find: 32]
//find by the unique id
customer.city = @"Dublin"; //update any field to a new value
[customer update];
[customer submitPending];
[SUP101SUP101DB synchronize];
```

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

```
SUP101Sales_Order* order = [SUP101Sales_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
[SUP101SUP101DB synchronize];
```

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

```
SUP101Customer *customer = [ SUP101Customer find: 32]
SUPObjectList* orderlist = customer.orders;
SUP101Sales_Order* order = [orderlist item:0];
order.sales_rep = @"Ram";
customer.state = @"MA" ;
[customer save];
[customer submitPending];
[SUP101SUP101DB synchronize];
```

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:

```
SUP101Customer *customer = [ SUP101Customer find: 32]
[Customer delete];
[Customer submitPending];
[SUP101SUP101DB synchronize];
```

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

```
SUP101Sales_order *order = [SUP101Sales_order find: 32]
[order delete];
[order.customer submitPending]; //Call submitPending on the parent.
[SUP101SUP101DB synchronize];
```

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

```
SUP101Customer *customer = [ SUP101Customer find: 32]
//Change some sttribute of the customer record
customer.fname= @"New Name";
[customer save];
[SUP101SUP101DB synchronize];
```

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:

```
SUP101CustomerChangeLastNameOperation *op =
[SUP101CustomerChangeLastNameOperation getInstance];
op.old_lname = @"Smith";
op.new_lname = @"Jones";
[op save];

[op submitPending];
[SUP101SUP101DB synchronize];
```

Pending Operation

You can manage the pending state.

- **`(void)submitPending`** – Submits the operation so that it can be replayed on the Unwired Server. A request is sent to the Unwired Server during a synchronization.

  ```
  [customer submitPending];
  ```
- **`(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)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"];
  ```

- **(void)cancelPendingOperations** – Cancels the pending operations for an entire entity. This method internally invokes the `cancelPending` method.

  ```
  [Customer cancelPendingOperations];
  ```

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

```
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.

Name	Type	Description
`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 (`replayFailure` 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.
`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];`

Name	Type	Description
`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

Description	Flags/Values
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
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

Description	Flags/Values
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
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

Description	Flags/Values
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:

```
Customer *cust = [Customer findById:101];
cust.fname = @"newName";
[cust refresh]; // newName is discarded
```

Generated Package Database APIs

The generated package database APIs include methods that exist in each generated package database.

Client Database APIs

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

```
+ (void) createDatabase;
+ (void) deleteDatabase;
+ (BOOL) databaseExists;
```

Typically, `createDatabase` does not need to be called since it is called internally when necessary. An application may use `deleteDatabase` when uninstalling the application.

Use the transaction API to group several transactions together for better performance.

```
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 `SUPStreamNotOpenException` if the stream is not open.

Syntax

```
- (void)close;
```

Examples

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

```
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 `SUPObjectNotSavedException` if this `SUPBigBinary` object is an attribute of an entity that has not yet been created in the database. Throws a `SUPStreamNotClosedException` 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 `SUPObjectNotSavedException` if this `SUPBigBinary` object is an attribute of an entity that has not yet been created in the database. Throws a `SUPStreamNotClosedException` 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 `SUPStreamNotOpenException` 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 `SUPObjectNotSavedException` if this `SUPBigBinary` object is an attribute of an

entity that has not yet been created in the database. Throws an `SUPObjectNotFoundException` if this object is null.

Syntax

```
- (void)openForRead;
```

Examples

- **Open for reading –** 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 `SUPObjectNotSavedException` 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

- **Open for writing –** 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 `SUPStreamNotOpenException` 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

- **Read –** 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 `SUPStreamNotOpenException` 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 `SUPStreamNotOpenException` 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 `SUPStreamNotOpenException` if the stream is not open for writing. Throws a `SUPWriteAppendOnlyException` 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 `SUPWriteOverLengthException` 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

- **Write data –** 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 `SUPStreamNotOpenException` if the stream is not open for writing. Throws a `SUPWriteAppendOnlyException` 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 `SUPWriteOverLengthException` 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 `SUPStreamNotOpenException` if the stream is not open.

Syntax

```
- (void)close;
```

Examples

- **Close the value stream –** 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 `SUPObjectNotSavedException` if this `SUPBigString` object is an attribute of an entity that has not yet been created in the database. Throws a `SUPStreamNotClosedException` 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 `SUPObjectNotSavedException` if this `SUPBigString` object is an attribute of an entity that has not yet been created in the database. Throws a `SUPStreamNotClosedException` 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 `SUPStreamNotOpenException` 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 `SUPObjectNotSavedException` if this `SUPBigString` object is an attribute of an entity that has not yet been created in the database.

Syntax

```
- (void)openForRead;
```

Examples

- **Open for reading –** 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 `SUPObjectNotSavedException` 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

- **Open for writing –** 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 `SUPStreamNotOpenException` 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

- **Read –** 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 `SUPStreamNotOpenException` 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 `SUPStreamNotOpenException` 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 `SUPStreamNotOpenException` if the stream is not open for writing. Throws a `SUPWriteAppendOnlyException` 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 `SUPWriteOverLengthException` 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

- **Write data –** 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 `SUPStreamNotOpenException` if the stream is not open for writing. Throws a `SUPWriteAppendOnlyException` 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 `SUPWriteOverLengthException` 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 API

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

MetaData API

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

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

You can generate metadata classes using the `-md` code generation option. You can also generate metadata classes by selecting the option **Generate metadata classes** 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]);
    }
}
```

EntityMetaData

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

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.

Exception Handling

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

To use the localization features in exception handling:

- Use the SUPExceptionMessageServiceImpl to import resource bundles to your project. The default implementation provides error message strings for English. You can optionally create more localized files for other languages.
- Register an exception message service implementation through the `SUPServiceRegistry`.

Base Exceptions

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

```
/*!
 @class SUPBaseException
 @abstract  This class contains information about the exception,
error code and error messages.
 @discussion
```

```
 */
@interface SUPBaseException : NSException {
    NSArray*        _arguments;
    int             _errorCode;
    NSException*    _cause;
}


// the error code property
@property(readwrite, assign, nonatomic) int errorCode;

// the root exception
@property(readwrite, retain, nonatomic) NSException* cause;

// localized error message
@property(readwrite, copy, nonatomic) NSString* message;

...

/*!
 @method messageWithLocale
 @abstract get the error message using the locale specified
 @result the localized message
 @discussion
 */
- (NSString *)messageWithLocale:(NSString *)locale;

@end;
```

You can use the `message` and `messageWithLocale(String locale)` methods to retrieve an error message for a specified locale. `message` is the `NSString*` message property and `messageWithLocale` is the `messageWithLocale:NSString*` locale method.

```
@try
{
// …
}
@catch (SUPBaseException *e)
{
NSString* errorMessage = e.message;
NSString* errorMessageSpanish = [e messageWithLocale:@"es"];
}
```

Exception Message Service

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

To resolve localized messages, implement the `SUPExceptionMessageService` protocol.

```
/*!
 @protocol
 @abstract  SUPExceptionMessageService protocol
 @discussion  SUPExceptionMessageServiceImpl is the default
implementation provided for SUPExceptionMessageSerivce protocol can
be registered with the SUPServiceRegistry.
 */
@protocol SUPExceptionMessageService

/*!
 @method
 @abstract  Get the message of this error code.
 @param errorCode The error code for the message.
 @result the error message
 @discussion
 */
-(NSString*) messageWithErrorCode: (int) errorCode;

/*!
 @method
 @abstract Get the localized message of this error code for a
specific locale
 @param errorCode The error code for mthe message
 @param locale locale identifier
 @result the localized message
 @discussion The locale identifier is the language-specific project
(.lproj) directory name for loading resource bunlde,
ErrorMessages.strings.  It could be also the value passed to
NSString's initWithFormat method for string formatting the
arguments.

The locale value can be in one of the following two forms:

- "language":  language specific value.  eg: @"en"</li>
- "language"_"region":  language and region specific value.  eg:
@"en_US"

If the resource bundle is not found in the "language"_"region" form,
The "language" part of the value is used to load the resource bundle.
If a resource bundle is not found, go by [[NSBundle mainBundle]
preferredLocalizations].  If it is still not found, defaults to
"en".  If the value is not one of the locale identifiers available in
[NSLocale availableLocaleIdentifiers],  the locale in [[NSLocale
currentLocale] localeIdentifier] is used in string formatting the
arguments.

 */
-(NSString*) messageWithErrorCode: (int) errorCode locale:
(NSString*) locale;

@end
```

The exception class uses the exception message service to load resource bundles and look up error messages based on an error code.

```
id<SUPExceptionMessageService> provider = [[SUPServiceRegistry
sharedInstance] getService:@protocol(SUPExceptionMessageService)];
NSString *message = [provider messageWithErrorCode:errorCode];
```

You can use a default message provider, `SUPExceptionMessageServiceImpl`. The default implementation provides a `superr.bundle` which contains the default English resource to look up an error message using an error code.

The `SUPExceptionMessageServiceImpl` loads resource bundles from the `superr.bundle`. You must import the `superr.bundle` in *`SUP_HOME`*`/ObjectAPI/iOS/resources/superr.bundle` to the project.

You can add support for other languages by adding new error message key-value pairs to a file named `ErrorMessages.strings` inside a folder named using a *<language code>*.lproj pattern. The `superr.bundle` structure is:

```
superr.bundle
  en.lproj
    ErrorMessages.strings
  <language code>.lproj
    ErrorMessages.strings
  <language code>.lproj
    ErrorMessages.strings
```

For example, to add support for Spanish:

1. Create a new folder, for example es.lprj, inside `superr.bundle`.
2. Create a new ErrorMessage.strings text file inside the `es.lprj` folder.
3. Define new localized error messages for the same set of error message keys found using the format "<error code>" = "<error message in Spanish>".
4. Rebuild the application with the new `superr.bundle` file.

You can create a custom provider by implementing your own `SUPExceptionMessageService`.

```
@interface CustomMessageService : NSObject
<SUPExceptionMessageService>

@end

@implementation CustomMessageService


-(NSString*) messageWithErrorCode: (int) errorCode
{
return @"my own way of retrieving the message";
}

-(NSString*) messageWithErrorCode: (int) errorCode locale:
(NSString*) localName
{
return @"my own way of retrieving the localized message";
}
```

```
@end


// register our custom message provider
CustomMessageService* myProvider = [[CustomMessageService alloc]
init];
[[SUPServiceRegistry sharedInstance]
registerService:@protocol(SUPExceptionMessageService)
withImplementation:myProvider];
```

See *Service Registry* for sample code on using the default exception message provider and how to register the default provider with the service registry.

Service Registry

The service registry holds implementation instances for various services used by the entity framework and applications. To allow you to use the exception message service, you must register the exception message service implementation represented by the `SUPExceptionMessageService` protocol with the service registry.

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

```
- (id)registerService:(Protocol *)protocol withImplementation:
(id)service;

- (id)unregisterService:(Protocol *) protocol;
```

For example:

```
// register our default message service
id <SUPExceptionMessageService> service =
[SUPExceptionMessageServiceImpl exceptionMessageServiceImpl];

SUPServiceRegistry* sr = [SUPServiceRegistry sharedInstance];
[sr registerService:@protocol(SUPExceptionMessageService)
withImplementation:service];
```

Example Code for Handling Exceptions

An example of registering your interface.

```
defaultMessageProvider = [SUPExceptionMessageDefaultProvider
getInstance];

// register a custom message provider
SUPServiceRegistry* sr = [SUPServiceRegistry getInstance];
[sr registerService:@protocol(SUPExceptionMessageProvider)
withImplementation:defaultMessageProvider];
```

You can retrieve error codes using the `errorCode` property of `SUPBaseException`:

```
@try
{
  // …
}
```

```
@catch (SUPBaseException *e)
{
    if(e.errorCode != ERR_APP_NOT_REGISTERED)
    {
  }
}
```

To retrieve the error message using the preferred language for the device:

```
@try
{
  // …
}
@catch (SUPBaseException *e)
{
  NSString* errorMessage = e.message;
}
```

To retrieve the error message for a specific language:

```
@try
{
  // …
}
@catch (SUPBaseException *e)
{
  NSString* errorMessageSpanish = [e messageWithLocale:@"es"];
}
```

You can catch exceptions using the built-in support in Objective-C. The object can be either a `SUPBaseException` object or a subclass of the `SUPBaseException` object such as the `SUPPersistenceException` object.

```
@try
{
  [self CallMethodThatMightThrowException];
}
@catch (SUPPersistenceException *e)
{
  // this will catch all SUPPersistenceException type objects
}
@catch (SUPBaseException *e)
{
  // this will catch all other SUPBaseExcepiton type objects
}
@finally
{
  // finally block…
}
```

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.

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*.

Exception Classes

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

- **SUPSynchronizeException –** thrown when an exception occurs during synchronization.
- **SUPPersistenceException –** thrown when trying to access the local database.
- **SUPObjectNotFoundException –** thrown when trying to load an MBO that is not inside the local database.
- **SUPNoSuchOperationException –** thrown when trying to call a method (using the Object Manager API) but the method is not defined for the MBO.
- **SUPNoSuchAttributeException –** thrown when trying to access an attribute (using the Object Manager API) but the attribute is not defined for the MBO.
- **SUPApplicationRuntimeException –** thrown when a call to start the connection, register the application, or unregister the application cannot be completed due to an error.
- **SUPConnectionPropertyException –** 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.

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.

Error Codes

Codes for errors occuring during application execution.

HTTP Error Codes

The 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 (`LogRecord.EisCode`, and `LogRecord.Code`, respectively).

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

Table 6. Recoverable Error Codes

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

Table 7. Unrecoverable Error Codes

Error Code	Probable Cause	Manual Recovery Action
401	Backend EIS credentials wrong.	Change the connection information, or backend user password.
403	User authorization failed on the 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 8. Mapping of SAP Error Codes to HTTP Error Codes

Constant	Description	HTTP Error Code
JCO_ERROR_COMMUNICATION	Exception caused by network problems, such as connection breakdowns, gateway problems, or unavailability of the remote SAP system.	503
JCO_ERROR_LOGON_FAILURE	Authorization failures during login. Usually caused by unknown user name, wrong password, or invalid certificates.	401
JCO_ERROR_RESOURCE	Indicates that JCO has run out of resources such as connections in a connection pool.	503
JCO_ERROR_STATE_BUSY	The remote SAP system is busy. Try again later.	503

Index

A

Afaria 14, 26, 96
APNS 61
Apple gateway 157
Apple Push Notification API 157
Apple Push Notification Service 61
Application APIs
    retrieve connection properties 75
application callback handlers 154
application provisioning
    with iPhone mechanisms 61
application registration 37
arbitrary find method 160, 161, 163, 166
ARC 18, 30
AttributeTest 161, 166
AttributeTest condition 160
authentication
    online 40
AVG 163

B

beginOnlineLogin 117
beginSynchronize 119

C

callback handlers 40, 151
CallbackHandler 55
callbacks 40
CertBlobUtility 98
certificates 6, 24, 96, 104
ClassMetadata 189
ClassMetaData 188
client database 177
closeConnection 105
complex type 44
CompositeTest 166
CompositeTest condition 160
concatenate queries 164
connection profile 38, 39
ConnectionProfile 104
ConnectionProperties 79
    retrieve activation code 79
    retrieve Farm ID 82
    retrieve HTTP cookies 83
    retrieve HTTP credentials 83
    retrieve HTTP headers 82
    retrieve login certificate 79
    retrieve login credentials 80
    retrieve network protocol 79
    retrieve port number 80
    retrieve security configuration 81
    retrieve server name 80
    retrieve URL suffix 81
COUNT 163
create 46
create operation 168
createDatabase 177

D

data synchronization protocol 3, 4
data vault 136
    access group 148
    change password 147
    creating 134
    deleting 136
    exists 136
    locked 143
    locking 142
    retrieve data names 137
    retrieve string 144
    retrieve value 146
    set string 144
    set value 145
    unlocking 143
database
    client 177
database connections
    managing 105
debugging 55, 57
delete 46
delete operation 169
deleteDatabase 177
documentation roadmap 4
downloading Xcode IDE 6, 24
dynamic query 43, 44

E

EIS error codes 196, 197
encryption key 133, 134
entity states 172, 174
error codes
    EIS 196, 197
    HTTP 196, 197
    mapping of SAP error codes 197
    non-recoverable 196
    recoverable 196
EXCEPT 164
exceptions
    client-side 195
    server-side 195

F

filtering results 164
FROM clause 165

G

generated code contents 12, 25
generated code, location 12, 25
getLogRecords 123
group by 164

H

HeaderDoc 13, 26
HTTP error codes 196, 197

I

infrastructure provisioning
    with iPhone mechanisms 61
INTERSECT 164
iPhone
    iTunes provisioning 63
    provisioning 61

J

Javadocs, opening 67
JMSBridge 55

L

listeners 40
localization 59, 60
LogRecord API 123
LogRecordImpl 123, 127

M

MAX 163
maxDbConnections 105
MBO 42–44, 46
MBOLogger 55, 126
messaging protocol 3, 4
MetaData API 188
MIN 163
mobile middleware services 4

N

newLogRecord 123
NoSuchAttributeException 195
NoSuchOperationException 195

O

Object API code
    location of generated 12, 25
Object Manager API 188
object query 43, 159
ObjectNotFoundException 195
OnImportSuccess 116
onlineLogin 108
openConnection 105
other operation 170

P

paging data 160, 163
password policy 142
    set 138
pending operation 170
pending state 46
personalization keys 115
    types 114
provisioning
    employee iPhone applications 63
provisioning devices
    with iPhone mechanisms 61
push notifications 157

Q

Query class 160
Query object 161, 163, 166

R

recover 121
Refresh operation 177
relationships 167
resumeSubscription 120

S

save operation 170
SelectItem 165
setting the database file location on the device 106
setting the databaseFile location 106
signing 61
simultaneous synchronization 116
Skip 166
Skip condition 160
SortCriteria 163, 166
SortCriteria condition 160
status methods 172, 174
submitLogRecords 123
subqueries 165
subscribe 117
subscribe() 116
SUM 163
SUPAbstractClassException.h 195
SUPAttributeMetaData 189
SUPBigBinary 178
SUPBigString 183
SUPBridge 55
SUPDatabaseMetaData 188
SUPDataVault 134
SUPDataVaultException 134
SUPInvalidDataTypeException.h 195
SUPNoSuchAttributeException.h 195
SUPNoSuchClassException.h 195
SUPNoSuchOperationException.h 195
SUPNoSuchParameterException.h 195
SUPObjectNotFoundException.h 196
SUPPersistenceException.h 196
SUPQuery class 160
SUPQuery object 163
SUPQueryResultSet 167
SUPWrongDataTypeException.h 195
suspendSubscription 119
synchronization 41
    MBO package 116
    of MBOs 116
    replication-based 116
    simultaneous 116
synchronization parameters 42
synchronization profile 38
SynchronizationProfile 106, 107
SynchronizeException 195

T

TestCriteria 166
TestCriteria condition 160

U

UNION 164
UNION_ALL 164
unsubscribe 118
update 46
update operation 169

V

value
    deleting 146

X

X.509 certificates 6, 24
Xcode 14, 18, 26, 30