SYBASE®
An SAP® Company

Developer Reference for iOS

Sybase Unwired Platform 1.5.5

DOCUMENT ID: DC01217-01-0155-01
LAST REVISED: December 2010

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

Contents

Introduction to Developer Reference for iOS

This developer reference provides information about using advanced Sybase® Unwired Platform features to create applications for Apple iOS devices, including iPhone and iPad. The audience is advanced developers who are familiar working with APIs, but who may be new to Sybase Unwired Platform.

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

Companion guides include:

- *Sybase Unwired WorkSpace – Mobile Business Object*
- *Tutorial: iPhone Device Application Development (Custom Development)*
- *Troubleshooting for Sybase Unwired Platform*

HeaderDoc provides a complete reference to the APIs:

- The Framework Library HeaderDoc is installed to `<UnwiredPlatform_InstallDir>\Servers\UnwiredServer\ClientAPI\apidoc\ObjectiveC`.
- You can generate HeaderDoc from the generated Objective-C code. See *Generating HeaderDoc from Generated Code* on page 14.

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

Documentation Road Map for Unwired Platform

Learn more about Sybase® Unwired Platform documentation.

Table 1. Unwired Platform documentation

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

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

Document	Description

Getting started tutorials	Tutorials for trying out basic development functionality.
	Audience: new developers, or any interested user.
	Use: after installation.
	• Learn mobile business object (MBO) basics, and create a mobile device application:
	  • *Tutorial: Mobile Business Object Development*
	  • *Tutorial: BlackBerry Application Development using Device Application Designer*
	  • *Tutorial: Windows Mobile Device Application Development using Device Application Designer*
	• Create native mobile device applications:
	  • *Tutorial: BlackBerry Application Development using Custom Development*
	  • *Tutorial: iPhone Application Development using Custom Development*
	  • *Tutorial: Windows Mobile Application Development using Custom Development*
	• Create a mobile workflow package:
	  • *Tutorial: Mobile Workflow Package Development*

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

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

Device Application Development

A device application includes business logic, and device-resident presentation and logic.

Mobile business objects help form the business logic for mobile applications. A mobile business object (MBO) is derived from a data source (such as a database server, Web service, or SAP® server). When grouped in projects, MBOs allow mobile applications to be deployed to an Unwired Server and referenced in mobile devices (clients).

Once you have developed MBOs and deployed them to Unwired Server, you add device-resident presentation and logic to the device application. You build a native client in the Xcode

IDE using Objective-C and Generated Object API code, and by programmatically binding to the iOS Client Object API.

Development Task Flows

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

Task Flow for Xcode IDE Development

Follow this task flow to develop a device application.

Prerequisites

Before developing a device application, the developer must:

- In the Eclipse development environment, create a mobile application project and create mobile business objects as required for your application.
  See the following topics in *Sybase Unwired WorkSpace – Mobile Business Object Development* for instructions on developing mobile business objects, and configuring the mobile business object attributes, as well as synchronization and personalization parameters:
  - *Sybase Unwired WorkSpace – Mobile Business Object Development > Develop > Developing a Mobile Business Object*
  - *Sybase Unwired WorkSpace – Mobile Business Object Development > Develop > Working with Mobile Business Objects*

  Note: Ensure that you enter a package name for the mobile application project that is appropriate as a prefix for the mobile business object generated files. In the examples that follow, the package name is SampleApp.
- Verify the supported device platforms and code generation tools for your device application. See *Planning Your Sybase Unwired Platform Installation > Supported Device Platforms and Databases* in the *Sybase Unwired Platform Installation Guide*

Task

1. Create mobile business object generated code. See *Mobile Business Object Code* on page 8.
2. Import libraries and code into the Xcode IDE. See *Importing Libraries and Code into the Xcode IDE* on page 11.
3. Develop a device application in the Xcode IDE.
   a) Create HTML reference information for the methods in your generated code. This will help you to programmatically bind to the Client Object API. See *Generating HeaderDoc from Generated Code* on page 14

b) Configure your application to synchronize and retrieve data from a mobile business object. See *Configuring an Application to Synchronize and Retrieve MBO Data* on page 15.
c) Reference your application to the Client Object API code that you generated for your mobile application project. See *Referencing the iOS Client Object API* on page 17.

4. Deploy your device application to devices in your enterprise. See *Deploying Applications to Devices* on page 29.

Mobile Business Object Code

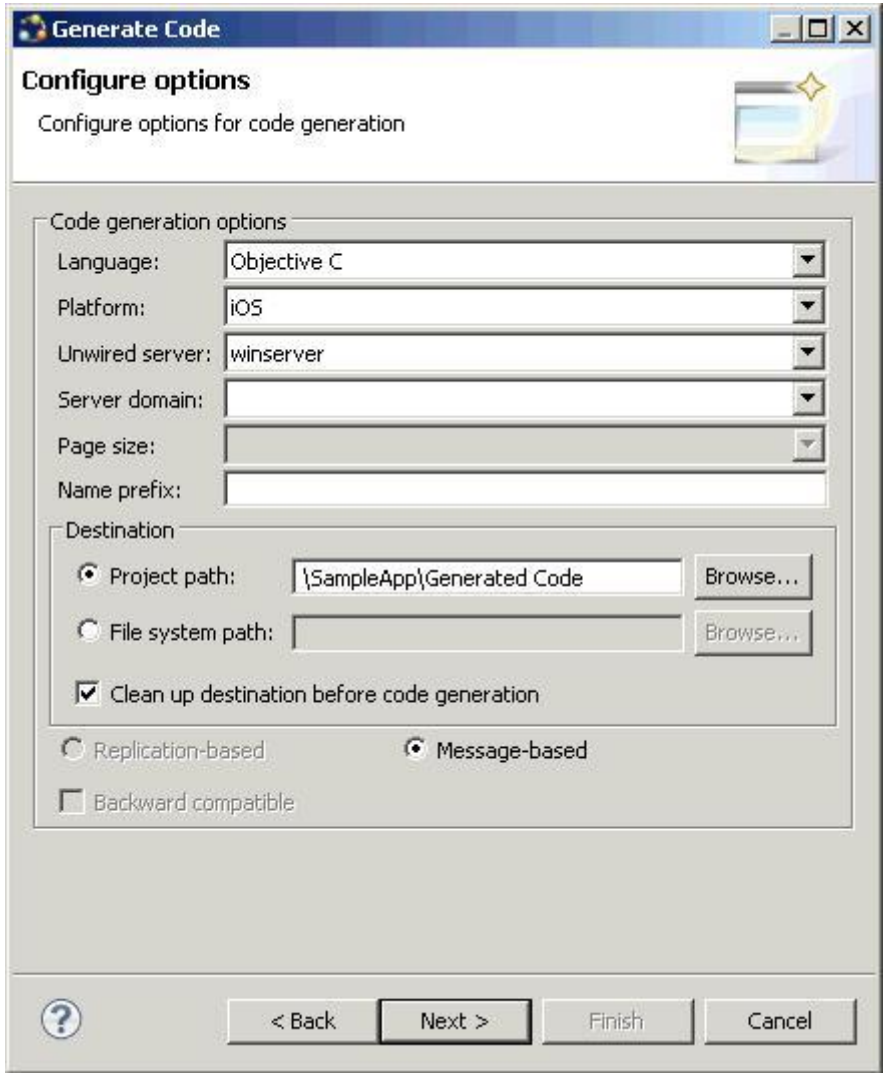
Develop a device application directly from mobile business object (MBO) generated code.

Generating Mobile Application Project Code

After developing the mobile business objects (MBOs), generate the files that implement the business logic and are required for Xcode IDE development.

Use this procedure if you are developing device applications using the Xcode IDE.

1. From Unwired WorkSpace, right-click in the Mobile Application Diagram of the project for which you are generating code and select **Generate Code**.
2. Follow the Code Generation wizard instructions to generate code appropriate for the Xcode IDE environment, by selecting **Objective C** as the language, **iOS** as the platform, and **Message-based**.



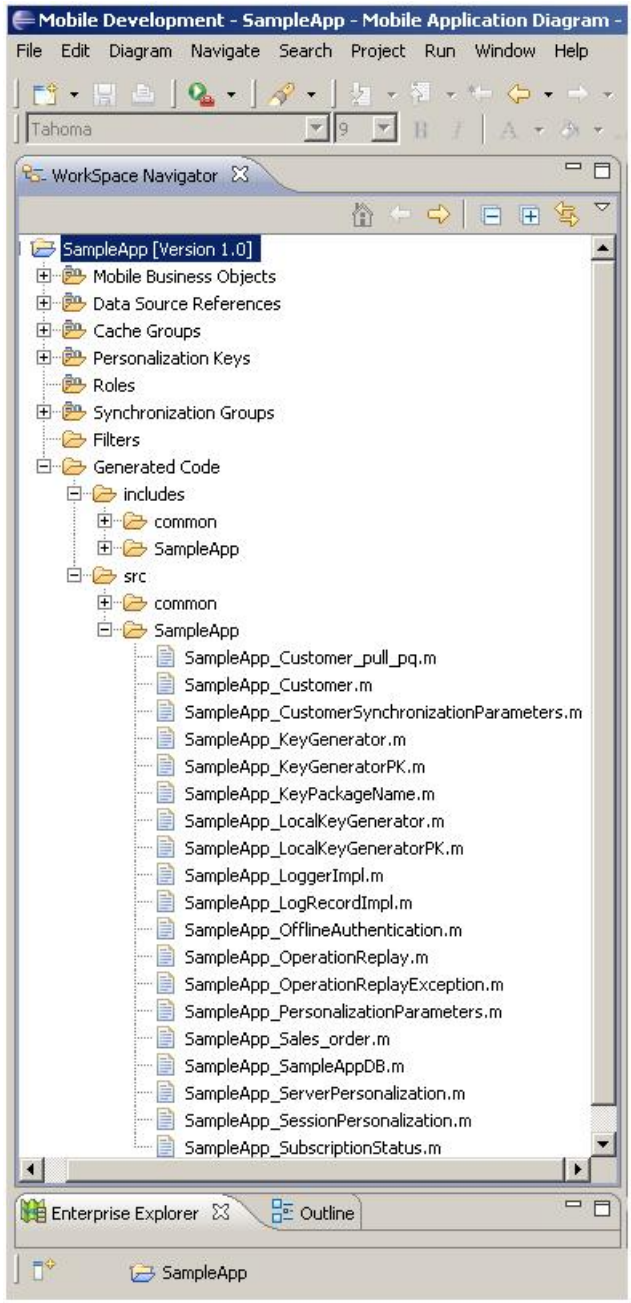
Other selections affect generated output as well.

3. Click **Finish**.

   The class files include all methods required to create connections, synchronize deployed MBOs with the device, query objects, and so on, as defined in your MBOs.

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

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



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

Table 2. Source Code File Descriptions

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

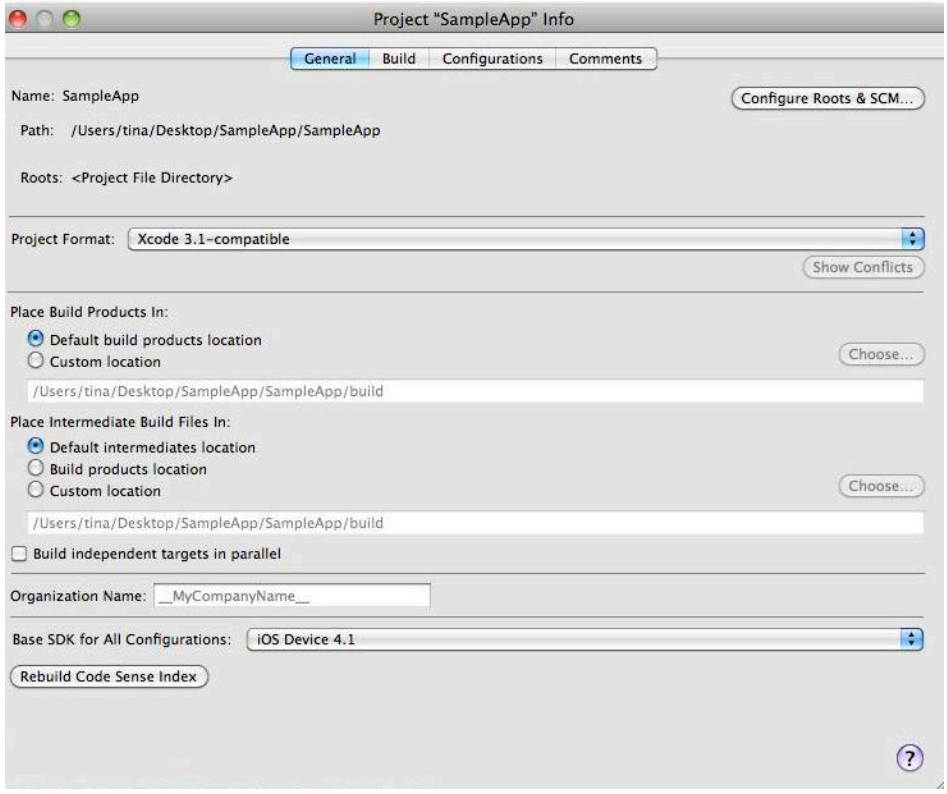
Note: Do not modify generated MBO API generated code directly. For MBO generated code, create a layer on top of the MBOs using patterns native to the mobile operating system development to extend and add functionality.

Importing Libraries and Code into the Xcode IDE

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. In the Xcode IDE, create a new Xcode project.
2. In the project settings, set the base SDK configuration for an iPhone simulator or device to iOS Device 4.1. If your code needs to run on a device with an earlier version of the OS (3.2 for iPad, or 3.1.3 for iPhone), this can be changed by setting the “iPhone OS Deployment Target”.



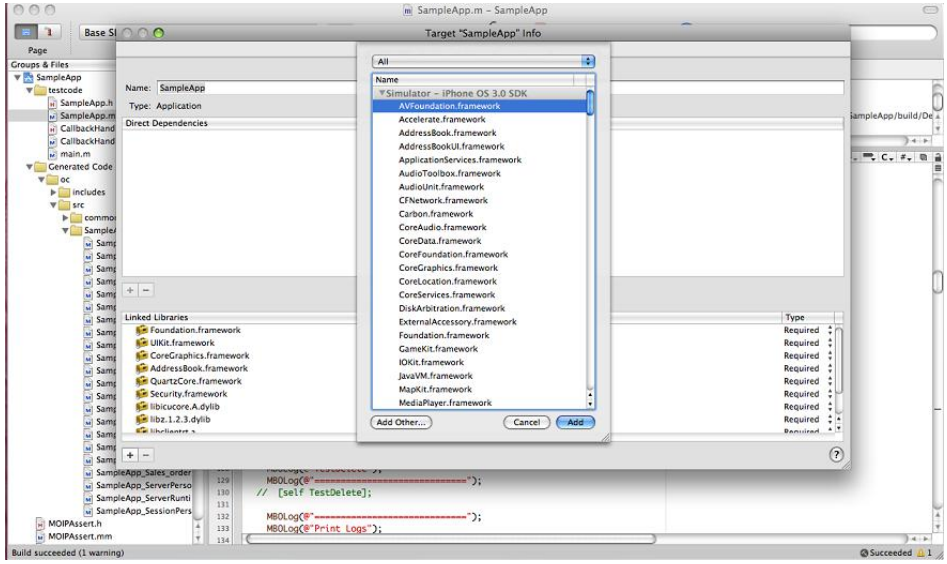
3. Copy the generated code from your Microsoft Windows environment to a location on your Mac (for example, your Home directory).
4. Copy over the include files from `<unwired server install>\ClientAPI\ObjectiveC\includes` and the libraries from `<unwired server install>\ClientAPI\ObjectiveC\libs` to a directory on your Mac (for example, your Home directory). There are two library directories: the `libs` directory (for iPhone), and the `libs.iPad` directory (for iPad). If building for iPad simulator or device, you must use the libraries in `libs.iPad`.
   a) After copying the directories into a local directory on your Mac, open **Finder** and locate the `<unwired server install>\ClientAPI\ObjectiveC\includes` folder.
   b) Drag the `<unwired server install>\ClientAPI\ObjectiveC\includes\internal` and `<unwired server install>\ClientAPI\ObjectiveC\includes\public` subfolders into Groups & Files, under the project name.
   c) If prompted to copy existing items into the destination group's folder, ensure **Copy items into destination group's folder (if needed)** is selected and then click **Add** to

copy the `include\internal` and `include\public` directories into your project's folder.

5. Add the generated *.h and *.m files to the project:
   a) In the Xcode Groups & Files pane, right-click **<Project Name>**, and create a new group in your project.
   b) Import the generated code into the new group by selecting **Add**, then **Existing Files**.
   c) Navigate to the directory that contains the generated code.
   d) Select both the `includes` and `src` folders for the generated code. Click **Add**.
   e) If prompted to copy existing items into the destination group's folder, ensure **Copy items into destination group's folder (if needed)** is selected and then click **Add** to copy the `Generated Code` folder into your project's folder. This step ensures that all .h and .m files are added to the project's search path.
6. Add `libclientrt.a`, `libSUPObj.a`, and `libMO.a` to your project.
   a) In the Xcode Groups & Files pane, select and right-click **<Project Name>** and select **Add**, then **Existing Files**.
   b) Navigate to the directory where you copied the libraries.
   c) Select the `libclientrt.a`, `libSUPObj.a`, and `libMO.a` libraries in Finder. Drag the libraries into Xcode under your project's name.
   d) Select **Copy items into destination group's folder (if needed)**, then click **Add**.

   Note: The library version should correspond 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.
7. Add `Settings.bundle` to the Xcode project:
   a) Select and right-click **<Resources>**, and select **Add**, then **Existing Files**.
   b) Navigate to the `includes` directory, select `Settings.bundle`, and add it.
   c) Select **Copy items into destination group's folder (if needed)**, then click **Add**.

   Note: This allows the device client user to use the Settings application to input their user preference information, such as server name, server port, user name, and activation code.
8. Add the following frameworks from the SDK to the project by selecting **Project** > **Edit Active Target <ProjectName>** > **General**.
   - Security.framework
   - AddressBook.framework
   - QuartzCore.framework
   - CoreFoundation.framework
   - libicucore.A.dylib
   - libz.1.2.3.dylib
   - libstdc++.dylib



9. Edit the Xcode project Library Search Paths by selecting **Project** > **Edit Active Target <ProjectName>** > **Build** > **Search Paths** > **Library Search Paths**. Specify the path to the location where you copied the libraries in step 6. Remove any libstdc++ paths (such as `usr/lib/arm-apple-darwin10/4.2.1`) from the library search path.
   a) Edit the Header Search Paths to include the `include\internal` and `include\public` directories.
10. For debug builds, check **Build Active Architecture Only**, and make sure that the armv6 architecture is selected.
11. Write your application code to reference the generated MBO code. See *Referencing the iOS Client Object API* on page 17.

Developing Applications in the Xcode IDE

After you import Unwired WorkSpace projects (mobile application) and associated libraries into the iOS development environment, use the iOS Client Object API to create or customize your device applications.

This section describes how to customize device applications in the Xcode IDE using Sybase provided APIs.

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*.

Configuring an Application to Synchronize and Retrieve MBO Data

To configure an application to synchronize and retrieve MBO data you must create a synchronization profile, start the client engine and configure the physical device settings, and subscribe to a package.

1. Create a synchronization profile by executing:

   ```
   SUPConnectionProfile* cp = [SampleApp_SampleAppDB
   getSynchronizationProfile];
   [cp setDomainName:@"default"];
   ```

2. Register a callback (if required).

   If the application requires a callback (for example, to allow the client framework to provide notification of subscription request results, or results of failure), register the callback function after setting up the connection profile, by executing:

   ```
   MyCallbackHandler* theCallbackHandler = [MyCallbackHandler
   getInstance];
       [SampleApp_SampleAppDB
   registerCallbackHandler:theCallbackhandler];
   ```

   Note: See *Reference: Administration APIs > Reference > iPhone Client Object API > Utility APIs > Callback Handlers* for more information on the Callback Handler interface. See *Reference: Administration APIs > Development Task Flows > Developing Applications in the Xcode IDE > Referencing the iPhone Client Object API* for more information on a sample application which includes a callback function.

3. Create the database and call `startBackgroundSynchronization`.

   Create a new database or make sure that the package database exists (either from a previous run, or a call to [`SampleApp_SampleAppDB createDatabase`]) and call `startBackgroundSynchronization`. You must perform these calls before you call [`SUPMessageClient start`] to connect to the Unwired Server.

```
[SampleApp_SampleAppDB createDatabase];
[SampleApp_SampleAppDB startBackgroundSynchronization];
```

When a mobile application is compiled with the client framework and deployed to a mobile device, the device must be activated before it can communicate with the Unwired Server.

To register with the Unwired Server, an application requires a user name and a unique device ID. In a typical scenario, the user receives an e-mail message from the Unwired Server with the application activation information. The user then enters the information using the Settings application, then runs the application to establish a connection to the Unwired Server. On success, the application connects with the Unwired Server. If the user name and activation code do not match, the application receives an error from the Unwired Server.

4. Register the device with the Unwired Server through Sybase Control Center. See *Developer Reference for iOS > Development Task Flows > Deploying Applications to Devices > > Device Registration > Registering the Device in Sybase Control Center*.
5. Configure Settings information on the physical device to complete device registration. See *Developer Reference for iOS > Development Task Flows > Deploying Applications to Devices > Device Registration > Configuring Physical Device Settings*.

   You must authenticate the application with the Unwired Server to allow you to subscribe to a server package. Unwired Server can provide success or failure results if you have a registered callback.
6. Start the Sybase Unwired Platform client engine by connecting to the Unwired Server:

   ```
   NSInteger stat = [SUPMessageClient start];
   ```
7. Subscribe to a server package, by executing:

   ```
       while([SUPMessageClient status] != STATUS_START_CONNECTED)
           [NSThread sleepForTimeInterval:0.2];
           [SampleApp_SampleAppDB beginOnlineLogin:@"supUser"
   password:@"s3pUser"];
          while([SampleApp_SampleAppDB getOnlineLoginStatus].status
   == SUPLoginPending)
       {
           [NSThread sleepForTimeInterval:0.2];
       }
   // After this, the status will be either SUPLoginSuccess or
   SUPLoginFailure
   if([SampleApp_SampleAppDB getOnlineLoginStatus].status ==
   SUPLoginSuccess)
       [SampleApp_SampleAppDB subscribe];
   ```

   This example also ensures the messaging client device is in a connected state. After a successful connection is established with the server to which the application has been deployed, when the application sends a request, the Client Object API puts the current user name and credentials inside the message for the Unwired Server to authenticate and authorize. The device application must set the user name and credential before sending any requests to the Unwired Server. This is done by calling the `beginOnlineLogin` API.

The device application sends a request to the server which processes the request. Any security failure results in a reject of the request. The user application then subscribes to a server package. If successful, the Unwired Server sends out a push message to the client application containing the application data. The Unwired Server also sends an acceptance message. The client receives the push and acceptance messages.

The client framework notifies the application of the result of success through an `onSubscribeSuccess` callback, if a callback function is registered. If an error occurs in the subscription process, the Unwired Server sends out a reject message for the subscription. The client receives a subscription request result notification message with failure from the Unwired Server, and may resubmit the subscription request. The client framework notifies the application of the result of failure through the `OnSubscribeFailure` callback, if a callback function is registered.

8. Unsubscribe from the server.

   The client application must send an unsubscribe request to remove the subscription from the Unwired Server:

   ```
   [SampleApp_SampleAppDB unsubscribe];
   ```

Referencing the iOS Client Object API

Example code that references the Client Object API generated for a mobile application project in the Eclipse environment.

The application uses two mobile business objects based on the Customer and SalesOrder tables in the `sampledb` Sybase SQL Anywhere® (ASA) database. A one-to-many relationship exists between the two mobile business objects.

The following figure illustrates the MBO schema that represents the relationship between the mobile business objects.

Figure 1: MBO Schema for Mobile Business Object Relationship

Device Application Example Code

The example code consists of five files.

main.m

- **main.m –** sets up settings for the Unwired Server and calls the *start* method.
- **CallbackHandler.h –** header file for the callback handler code.
- **CallbackHandler.m –** Objective-C source file for the callback handler.
- **SampleApp.h –** header file with method definitions that call the Client Object API.
- **SampleApp.m –** Objective-C source file.

```
#import <UIKit/UIKit.h>
#import "SampleApp.h"


int main(int argc, char *argv[]) {


  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

  SampleApp *app = [SampleApp getInstance];
  [app run];

  int retVal = UIApplicationMain(argc, argv, nil, nil);
  [pool release];
  return retVal;
}
```

```
#import <Foundation/Foundation.h>
#import "SUPDefaultCallbackHandler.h"

@interface CallbackHandler : SUPDefaultCallbackHandler
```

```
{
    SUPInt field_importCount;
    SUPInt field_replaySuccessCount;
    SUPInt field_replayFailureCount;
    SUPInt field_loginSuccessCount;
    SUPInt field_loginFailureCount;
    SUPInt field_importSuccessCount;

}

+ (CallbackHandler*)getInstance;
- (CallbackHandler*)init;
- (SUPInt)importCount;
- (void)setImportCount:(SUPInt)_importCount;
@property(assign) SUPInt importCount;
- (SUPInt)replaySuccessCount;
- (void)setReplaySuccessCount:(SUPInt)_replaySuccessCount;
@property(assign) SUPInt replaySuccessCount;
- (SUPInt)replayFailureCount;
- (void)setReplayFailureCount:(SUPInt)_replayFailureCount;
@property(assign) SUPInt replayFailureCount;
- (SUPInt)loginSuccessCount;
- (void)setLoginSuccessCount:(SUPInt)_loginSuccessCount;
@property(assign) SUPInt loginSuccessCount;
- (SUPInt)loginFailureCount;
- (void)setLoginFailureCount:(SUPInt)_loginFailureCount;
@property(assign) SUPInt loginFailureCount;
- (SUPInt)importSuccessCount;
- (void)setImportSuccessCount:(SUPInt)_importSuccessCount;
@property(assign) SUPInt importSuccessCount;

- (void)onImport:(id)theObject;
- (void)onReplayFailure:(id)theObject;
- (void)onReplaySuccess:(id)theObject;
- (void)onLoginFailure;
- (void)onLoginSuccess;
- (void)onSubscribeSuccess;
- (void)onSubscribeFailure;
- (void)onImportSuccess;
- (CallbackHandler*)finishInit;
- (void)initFields;
+ (void)staticInit;
- (void)dealloc;

@end
```

```
#import "CallbackHandler.h"

@implementation CallbackHandler
+ (CallbackHandler*)getInstance
{
  CallbackHandler* _me_1 = [[CallbackHandler alloc] init];
  [_me_1 autorelease];
  return _me_1;
}
```

```
- (CallbackHandler*)init
{
  [CallbackHandler staticInit];
  [self initFields];
  return self;
}

- (SUPInt)importCount
{
  return field_importCount;
}

- (void)setImportCount:(SUPInt)_importCount
{
  field_importCount = _importCount;
}

- (SUPInt)replaySuccessCount
{
  return field_replaySuccessCount;
}

- (void)setReplaySuccessCount:(SUPInt)_replaySuccessCount
{
  field_replaySuccessCount = _replaySuccessCount;
}

- (SUPInt)replayFailureCount
{
  return field_replayFailureCount;
}

- (void)setReplayFailureCount:(SUPInt)_replayFailureCount
{
  field_replayFailureCount = _replayFailureCount;
}

- (SUPInt)loginSuccessCount
{
  return field_loginSuccessCount;
}

- (void)setLoginSuccessCount:(SUPInt)_loginSuccessCount
{
    field_loginSuccessCount = _loginSuccessCount;
}

- (SUPInt)loginFailureCount
{
    return field_loginFailureCount;
}

- (void)setLoginFailureCount:(SUPInt)_loginFailureCount
{
    field_loginFailureCount = _loginFailureCount;
}
```

```
- (SUPInt)importSuccessCount
{
    return field_importSuccessCount;
}

- (SUPInt)importSuccessCount
{
  return field_importSuccessCount;
}

- (void)setImportSuccessCount:(SUPInt)_importSuccessCount
{
  Field_importSuccessCount = _importSuccessCount;
}


- (void)onImport:(id)theObject
{
  self.importCount = self.importCount + 1;
}

- (void)onReplayFailure:(id)theObject
{
  self.replayFailureCount = self.replayFailureCount + 1;
  MBOLog(@"============================");
  MBOLogError(@"Replay Failed");
  MBOLog(@"============================");
}

- (void)onReplaySuccess:(id)theObject
{
  self.replaySuccessCount = self.replaySuccessCount + 1;
  MBOLog(@"============================");
  MBOLog(@"Replay Successful");
  MBOLog(@"============================");
}

- (void)onLoginFailure
{
    MBOLog(@"============================");
    MBOLogError(@"Login Failed");
    MBOLog(@"============================");
    self.loginFailureCount++;
}

- (void)onLoginSuccess
{
  MBOLog(@"============================");
  MBOLog(@"Login Successful");
  MBOLog(@"============================");
  self.loginSuccessCount++;
}

- (void)onSubscribeSuccess
{
```

```
  MBOLog(@"============================");
  MBOLog(@"Subscribe Successful");
  MBOLog(@"============================");

}
-(void)onSubscribeFailure
{
  MBOLog(@"============================");
  MBOLogError(@"Subscribe Failed");
  MBOLog(@"============================");
}

- (void)onImportSuccess
{
  MBOLog(@"============================");
  MBOLog(@"Import Ends Successfully");
  MBOLog(@"============================");
  self.importSuccessCount++;
}


- (CallbackHandler*)finishInit
{
  return self;
}

- (void)initFields
{
}

+ (void)staticInit
{
}

- (void)dealloc
{
  [super dealloc];
}

@end
```

```
@interface SampleApp: NSObject
{
}

+ (SampleApp*)getInstance;
- (SampleApp*)init;
- (void)run;
- (SampleApp*)finishInit;
- (void)initFields;
+ (void)staticInit;
- (void)dealloc;
-(void)runAPITests;


/*Test functions that call Client Object APIs */
```

```
-(void)Testfind;
-(void)TestSynchronizationParameters;
-(void)TestPersonalizationParameters;
-(void)TestCreate;
-(void)TestUpdate;
-(void)TestDelete;
-(void)printLogs;
-(void)PrintCustomerSalesOrderData;

@end
```

```
#import "SampleApp.h"
#import "SampleApp_Customer.h"
#import "CallbackHandler.h"
#import "SampleApp_SampleAppDB.h"
#import "SampleApp_LogRecordImpl.h"
#import "SampleApp_Sales_order.h"
#import "SampleApp_LocalKeyGenerator.h";
#import "SampleApp_KeyGenerator.h"
#import "SUPMessageClient.h"

@implementation SampleApp

+ (SampleApp*)getInstance
{
  SampleApp* _me_1 = [[SampleApp alloc] init];
  [_me_1 autorelease];
  return _me_1;
}

- (SampleApp*)init
{
  [SampleApp staticInit];
  [self initFields];
  return self;
}


- (void)run
{
   NSInteger connectionResult ;

  // Set log level
  [MBOLogger setLogLevel:LOG_INFO];

  //Delete the old database and create a new one.
  if([SampleApp_SampleAppDB databaseExists])
  [SampleApp_SampleAppDB deleteDatabase];
  [SampleApp_SampleAppDB createDatabase];

// Set up synchronization profile .
  SUPConnectionProfile* cp = [SampleApp_SampleAppDB
getSynchronizationProfile];
  [cp setDomainName:@"default"];
```

```
  //Register a callback handler.
  CallbackHandler* databaseCH = [CallbackHandler getInstance];
  [SampleApp_SampleAppDB registerCallbackHandler:databaseCH];

  //Start backgroundsynchronization.
  [SampleApp_SampleAppDB startBackgroundSynchronization];
 //Connect to the server
  connectionResult = [SUPMessageClient start];

  if(connectionResult == kSUPMessageClientSuccess)

  {
          NSLog(@"Cannot start SUPMessageClient");
          exit(0);
     }
    while([SUPMessageClient status] != STATUS_START_CONNECTED)
[NSThread sleepForTimeInterval:0.2];
    [SampleApp_SampleAppDB beginOnlineLogin:@"supUser"
password:@"s3pUser"];
    while([SampleApp_SampleAppDB getOnlineLoginStatus].status ==
SUPLoginPending)
     {
          [NSThread sleepForTimeInterval:0.2];
          if(databaseCH.loginFailureCount > 0)
          {
              NSLog(@"SampleApp_SampleAppDB login failed.");
              exit(0);
          }
      }

    //Subscribe to the package.
    [SampleApp_SampleAppDB subscribe];

    // Wait for imports to come back from server
    while([databaseCH importSuccessCount] < 1)
    [NSThread sleepForTimeInterval:0.2];

    //Call the functions that execute the client APIs for
synchronization
    //parameters, personalization keys read, create, update and
delete
    [self runAPITests];
    // Unsubscribe
    [SampleApp_SampleAppDB unsubscribe];
    //Disconnect from server.
    [SUPMessageClient stop];

  }
}

- (SampleApp*)finishInit
{
  return self;
}

- (void)initFields
```

```
{
}

+ (void)staticInit
{
}

- (void)dealloc
{
  [super dealloc];
}


-(void)runAPITests
{
  MBOLog(@"============================");
  MBOLog(@"TestPersonalizationParameters");
  MBOLog(@"============================");
  [self TestPersonalizationParameters];

  MBOLog(@"============================");
  MBOLog(@"TestSynchronizationParameters");
  MBOLog(@"============================");
  [self TestSynchronizationParameters];

  MBOLog(@"============================");
  MBOLog(@"TestfindAll");
  MBOLog(@"============================");
  [self Testfind];

  MBOLog(@"============================");
  MBOLog(@"TestCreate");
  MBOLog(@"============================");
  [self TestCreate];

  MBOLog(@"============================");
  MBOLog(@"TestUpdate");
  MBOLog(@"============================");
  [self TestUpdate];

  MBOLog(@"============================");
  MBOLog(@"TestDelete");
  MBOLog(@"============================");
  [self TestDelete];

  MBOLog(@"============================");
  MBOLog(@"Print Logs");
  MBOLog(@"============================");
  [self printLogs];

}

-(void)PrintCustomerSalesOrderData
{
    SampleApp_Customer *onecustomer = nil;
    SUPObjectList *cl = nil;
```

```
    MBOLog(@"Customer data is:");
    cl = [SampleApp_Customer findAll];

  if(cl &&  [cl length] > 0 )
  {
    int i;
    for(i=0; i<[cl length]; i++)
    {
      onecustomer = [cl item:i];
      if (onecustomer) {
        MBOLog(@"%@ %@, %@, %@, %@",onecustomer.fname,
        onecustomer.lname,onecustomer.address,onecustomer.city,
        onecustomer.state);
        SUPObjectList *sl = [onecustomer salesOrders];
                    if(sl)
                    {
                         if([sl length] > 0)
           MBOLog(@"   This customer's sales orders are");
          else
            MBOLog(@"   This customer has no sales orders");
          for(SampleApp_Sales_order *so in sl)
           MBOLog(@"%@ %@,%d",so.order_date,so.region,so.sales_rep);
        }
      }
    }

  }
}


/***Retrieve data based on the synchronization parameter value.***/

- (void)TestSynchronizationParameters
{
  SampleApp_CustomerSynchronizationParameters* sp
    = [SampleApp_Customer getSynchronizationParameters];

  sp.size = 3;
  sp.user = @"userone";
  sp.param_city = @"Raleigh";
  [sp save];
  while ([SampleApp_SampleAppDB hasPendingOperations])
  {
    [NSThread sleepForTimeInterval:1];
  }

  [self PrintCustomerSalesOrderData];
}



/******Retrieve data based on the personalization parameter
value*****/

- (void)TestPersonalizationParameters
{
```

```
  SampleApp_PersonalizationParameters *pp = nil;
  pp = [SampleApp_SampleAppDB getPersonalizationParameters];

  pp.PKCity = @"New York";
  [pp save];
  while ([SampleApp_SampleAppDB hasPendingOperations])
  {
    [NSThread sleepForTimeInterval:1];
  }
  [self PrintCustomerSalesOrderData];
}

/*******Print logs record data from LogrecordImpl*****/
-(void)printLogs
{
  MBOLog(@"******* printLogs *******");
  SUPQuery *query = [SUPQuery getInstance];
  SUPObjectList *loglist = [SampleApp_SampleAppDB
getLogRecords:query];

  for(id o in loglist)
  {
    SampleApp_LogRecordImpl *log = (SampleApp_LogRecordImpl*)o;
    MBOLog(@"Log Record %llu: Operation = %@, Timestamp = %@, MBO =
%@,
      key = %@, message = %@",log.messageId,log.operation,
    [SUPDateTimeUtil
toString:log.timestamp],log.component,log.entityKey,log.message);
  }
}


/
************************************find*************************
*********/
/***Find all the customer records and print the first record to the
console*/

-(void)Testfind
{
  SampleApp_Customer *onecustomer = nil;
  SUPObjectList *cl = [SampleApp_Customer findAll];
  if(cl && [cl length] > 0 )
  {
    onecustomer = [cl item:0];
    if (onecustomer)
    {
      MBOLog(@"the full customer record data is : %@", onecustomer);
    }
  }

}


/****************************Create
```

```
**********************************/
/*****Create new customer and sales order records in the local
database
       and call submitPending to send the changes  to the server
*****/

-(void)TestCreate
{
  long key1 = [SampleApp_KeyGenerator generateId];
  long key2 = [SampleApp_KeyGenerator generateId];
  [SampleApp_KeyGenerator submitPendingOperations];
  while ([SampleApp_SampleAppDB hasPendingOperations])
  {
    [NSThread sleepForTimeInterval:1];
  }
  SampleApp_Customer *c = [[ SampleApp_Customer alloc] init];

  c.id_ = [SampleApp_LocalKeyGenerator generateId];
  c.fname = @"Dorothi";
  c.lname = @"Scranton";
  c.address = @"One Money Street";
  c.city = @"smallVille";
  c.state = @"MA";
  c.zip = @"97429";
  c.phone = @"2112222345";
  c.company_name = @"iAnywhere";
  c.surrogateKey = key1;
  SUPObjectList *orderlist = [ SampleApp_Sales_orderList
getInstance];
  SampleApp_Sales_order *o1 = [[SampleApp_Sales_order alloc] init];

  o1.id_ = [SampleApp_LocalKeyGenerator generateId];
  o1.order_date = [NSDate date];
  o1.fin_code_id = @"r1";
     o1.region = @"Eastern";
     o1.sales_rep = 902;
     o1.surrogateKey = key2;
     [ o1 setCustomer:c];
     [orderlist add:o1];
     [c setSalesOrders:orderlist];
     [c save];
     [c refresh];
     [c submitPending];
     assert(c.pending == YES);
     while ([SampleApp_SampleAppDB hasPendingOperations])
     {
         [NSThread sleepForTimeInterval:1];
     }
 }


/*****************************Update
*********************************/
/****Update an existing customer and sales record in the device
database
  and call submitPending to send the changes to the server.
```

```
**********/

- (void)TestUpdate
{
  SUPObjectList *cl = [SampleApp_Customer findAll];
  SampleApp_Customer *onecustomer = [cl item:0];
  SampleApp_Sales_order *order = [onecustomer.salesOrders item:0];

  onecustomer.fname = @"Johnny";
  order.region = @"South";
  [onecustomer save];
  [onecustomer refresh];
  [order refresh];
  [onecustomer submitPending];
  while ([SampleApp_SampleAppDB hasPendingOperations])
  {
    [NSThread sleepForTimeInterval:1];
  }


}

/*************************** Delete ************************/
/*Delete an existing record from the database and call
  submitPending to send the changes to the server.****/

-(void) TestDelete
{
  SUPObjectList *sl = [SampleApp_Sales_order findAll];
  SampleApp_Sales_order *order = [sl item:0];
  [order delete];
  [order.customer submitPending];
  while ([SampleApp_SampleAppDB hasPendingOperations])
  {
    [NSThread sleepForTimeInterval:1];
  }
}



@end
```

Deploying Applications to Devices

Deploy mobile applications to devices and register the devices with Unwired Server.

Device Registration

Messaging devices contain applications that send and receive data through messaging. An administrator must configure the device activation template properties for message-based synchronization (MBS) devices. Device activation requires user registration. Upon successful registration, the device is activated and set up with the template the administrator has selected.

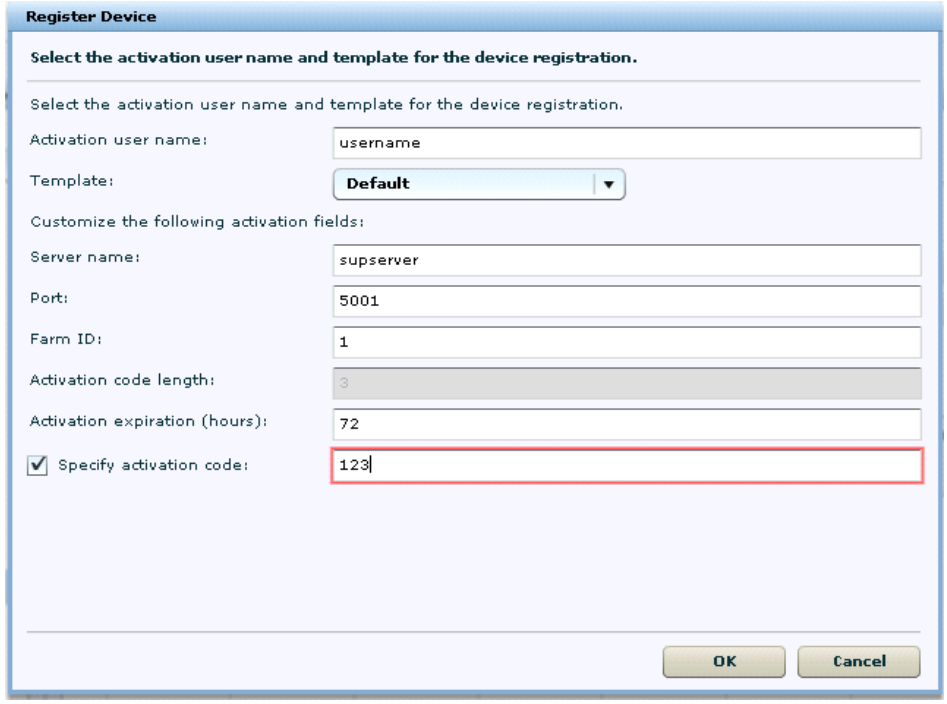
Device registration pairs a user and a device once the user supplies the correct activation code. This information is stored in the messaging database, which contains extensive information about users and their corresponding mobile devices.

See *System Administration > System Administration > Device and User Management > Messaging Devices > Device Registration and Activation and > System Administration > Device and User Management > Messaging Devices > Device Provisioning.*

Registering the Device in Sybase Control Center

Register the device in Sybase Control Center.

1. Log in to Sybase Control Center using the supAdmin/s3pAdmin user name and password.
2. In Sybase Control Center, select **View > Select > Unwired Server Cluster Management View**.
3. Expand the **Servers** folder in the left pane, and select **Device Users**.
4. In the right pane, click **Devices**.
5. Select **MBS** to display only MBS devices.
6. Click **Register**.
7. In the Register Device window, enter the required information:
   - User name
   - Server name
   - Port



Note: "localhost" should be the actual name of your machine.

Configuring Physical Device Settings

Access the Settings information on the physical device to complete device registration.

1. On your device, select **Settings** and select the name of your application, such as SampleApp.
2. In the Connection Info screen, enter the server name, user name, server port, company ID, and activation code. These entries must correspond to the Unwired Server name, the user name registered with the Unwired Server, the port the Unwired Server is listening to, the company ID, and the device activation code, respectively.

   If you are using a Relay Server, "Company ID" maps to the farm ID configured for messaging-based requests on the Relay Server, and the "Server Name" and "Server Port" map to the Relay Server name and port.

Deploying Applications to the Enterprise

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

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

1. Sign up for the iPhone Developer Program, which gives you access to the Developer Connection portal. Registering as an enterprise developer gets you the certificate you need to sign applications.
2. Configure the application to use make use of Keychain as persistent storage for the database encryption key. See *Configuring Application Security Using Key Chains.* on page 61
3. Create a certificate request on your Mac through Keychain.
4. Log in to the Developer Connection portal.
5. Upload your certificate request.
6. Download the certificate to your Mac. Use this certificate to sign your application.
7. 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.
8. 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.
9. 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.

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 on iPhones. APNS:

- Works only with iPhone physical devices
- Is not required for any iOS application
- Cannot be used on a on an iPhone simulator
- Cannot be used with iPod touch or iPad devices
- Must be set up and configured by an administrator on the server
- Must be enabled by the user on the device

Preparing an Application for Apple Push Notification Service

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

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

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

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

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

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

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

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

- (void)application:(UIApplication *)app
didRegisterForRemoteNotificationsWithDeviceToken:
```

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

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

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

// You can alternately implement the pushRegistrationFailed API:

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


* Callback when notification is sent.


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

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

Provisioning an Application for Apple Push Notification Service

If your internal users do not have an App Store account, use iTunes as an alternative method of provisioning the Sybase-packaged iPhone application. You can also use this method if you are building your own iPhone application.

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

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

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

   If the ports are open, you can connect to the Apple push gateway and receive feedback from it.

2. Copy the enterprise certificate (`*.p12`) to the computer on which Sybase Control Center has been installed. Save the certificate in `<SUP_Home>\Servers\MessagingServer\bin\`.
3. In Sybase Control Center, expand the **Servers** folder and click **Server Configuration** for the primary server in the cluster.
4. In the **Messaging** tab, select **Apple Push Configuration**, and:
   a) Configure Application name with the same name used to configure the product name in Xcode. If the certificate does not automatically appear, browse to the directory.
   b) Change the push gateway information to match that used in the production environment.
   c) Restart Unwired Server.
5. Verify that the server environment is set up correctly:
   a) Open `<SUP_Home>\Servers\UnwiredServer\logs\APNSProvider`.
   b) Open the log file that should now appear in this directory. The log file indicates whether the connection to the push gateway is successful or not.
6. Deploy the application and the enterprise distribution provisioning profile to your users' computers.
7. Instruct users to use iTunes to install the application and profile, and how to enable notifications. In particular, device users must:
   - Download the Sybase application from the App Store.
   - In the iPhone Settings app, slide the **Notifications** control to **On**.
8. Verify that the APNS-enabled iPhone is set up correctly:
   a) Click **Device Users**.
   b) Review the Device ID column. The application name should appear correctly at the end of the hexadecimal string.
   c) Select the Device ID and click **Properties**.
   d) Check that the APNS device token has been passed correctly from the application by verifying that a value is in the row. A device token appears only after the application runs.
9. Test the environment by initiating an action that results in a new message being sent to the client.

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

`<SUP_Home>\Servers\MessagingServer\Data\TraceConfig.xml` and configuring the tracing level for the APNSProvider module to debug for short periods.

Reference

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

iOS Client Object API

The Sybase Unwired Platform iOS Client Object API consists of generated business object classes that represent the mobile business object model built and designed in the Unwired WorkSpace development environment.

The iOS Client Object API is used by device applications to synchronize and retrieve data and invoke mobile business object operations. The iOS Client Object API supports only message-based synchronization.

Connection APIs

The iOS Client Object API contains classes and methods for managing local database information, and managing connections to the Unwired Server through a synchronization connection profile.

ConnectionProfile

The `ConnectionProfile` class manages local database information. You can use it to set the encryption key, which you must do before creating a local database.

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

SynchronizationProfile

Before synchronizing with Unwired Server, you must configure a client with information for establishing a connection with the Unwired Server where the mobile application has been deployed.

```
SUPConnectionProfile* cp = [SampleApp_SampleAppDB
getSynchronizationProfile];
[cp setDomainName:@”default”];
```

Authentication

The generated package database class provides a valid synchronization connection profile. You can log in to the Unwired Server with your user name and credentials.

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

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

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

- **`loginToSync` –** tries `offlineLogin` first. `offlineLogin` authenticates against the last successfully authenticated credential. There is no communication with the Unwired Server in this method. If `offlineLogin` fails, this method tries `onlineLogin`.
- **`beginOnlineLogin` –** sends the login request asynchronously (it returns without waiting for a server response). See *Reference: Administration APIs > Reference > iPhone Client Object API > Synchronization APIs*.

Synchronization APIs

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

- **+ (void)loginToSync:(NSString *)user password:(NSString *)pass –** `loginToSync` synchronizes the KeyGenerator from the Unwired Server with the client. The KeyGenerator is an MBO for storing key values that are known to both the server and the client. On `loginToSync` from the client, the server sends a value that the client can use when creating new records (by using the method `[KeyGenerator generateId]` to create key values that the server accepts).

  The KeyGenerator value increments each time the `generateId` method is called. A periodic call to `submitPending` by the `KeyGenerator generateId` MBO sends the most recently used value to the Unwired Server, to let the Unwired Server know what keys have been used on the client side. Place this call within a try/catch block and ensure that the client application does not attempt to send any more messages to the server if `loginToSync` throws an exception.

- **+ (void)beginOnlineLogin:(NSString *)user password:(NSString *)pass –** `beginOnlineLogin` is the recommended login method. It functions similarly to `loginToSync`, except it sends the login request asynchronously (it returns without waiting for a server response). This method checks the `SUPMessageClient` status and immediately fails if the status is not `STATUS_START_CONNECTED`. Make sure the connection is active before calling `beginOnlineLogin`, or implement the `onLoginFailure` callback handler to catch cases where it may fail.

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

Setting Synchronization Parameters

Synchronization parameters let an application change the parameters used to retrieve data from an MBO during a synchronization session. The primary purpose is to partition data. Change the synchronization parameter to affect the data that is retrieved.

When a synchronization parameter value is changed, the call to `save` automatically propagates the change to the Unwired Server; you need not call `submitPending` after the `save`. Consider the "Customer" MBO that has a "cityname" synchronization parameter.

This example shows how to retrieve customer data corresponding to Kansas City.

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

Query APIs

The Query APIs allow you to retrieve data from mobile business objects, to retrieve relationship data and paging data, and to retrieve and filter a query result set.

Retrieving Data from an MBO

To retrieve data from a local database use the `find`, `findAll`, or `findByPrimaryKey` methods in the MBO class.

The following examples show how to use the `find`, `findAll`, or `findByPrimaryKey` methods in the MBO class to retrieve data.

- **+ (<Name Prefix>_Customer*)find:(int32_t)id_ –** The `find` method retrieves a Customer by the given ID. The parameter `id_` is the surrogate key (the primary key used in the local database). The parameter is of type `int32_t` in this example, but could be another type based on the key type. The value "101" in this example is the surrogate key value (automatically generated from the KeyGenerator). To use this method, the client application must be able to retrieve the surrogate key.

  ```
  SampleApp_Customer *customer = [SampleApp_Customer find:101];
  ```

  Note: The Eclipse IDE allows you to specify a value for "name prefix" when generating the MBO Objective-C code. When a value is specified, all the MBO entity names are prefixed with that value. When no such prefix is specified, the name prefix is by default the package name.

- **+ (SUPObjectList*)findAll –** Call the `findAll` method to list all customers:

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

- **+(SUPObjectList*) findAll:(int32_t)skip take:(int32_t)take –** To define more than one `findAll` attribute, and return a collection of objects that match the specified search criteria, use:

  ```
  SUPObjectList *customers = [ SampleApp_Customer findAll: 100 take:
  5];
  ```

Methods Generated if Dynamic Queries are Enabled

- **+ (SUPObjectList*)findWithQuery:(SUPQuery*)query; –** Returns a collection of objects that match the result of executing a specific query. The method takes one parameter, `query` which is an `SUPQuery` object representing the actual query to be executed.

  ```
  SUPQuery *myquery = [SUPQuery getInstance];
  myquery.testCriteria = [SUPAttributeTest
  match:@"fname" :@”Erin”];
  SUPObjectList* customers = [SampleApp_Customer findWithQuery:
  myquery]
  ```

- **+ (int32_t)countWithQuery:(SUPQuery*)query; –** Returns a count of the records returned by the specific query.

  ```
  int count = [SampleApp_Customer countWithQuery:myquery];
  ```

Object Queries

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

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

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

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;
```

Supported Aggregate Functions

You can use aggregate functions including `GroupBy` in object queries. However, the `sum`, `avg`, and greater than (>) aggregate functions are not supported.

```
select count(x.id), x.id from AllType x where x.surrogatekey > :minSk
group by x.id having
x.id < :maxId order by x.id
```

Arbitrary Find

The arbitrary find method provides custom device application the ability to dynamically build queries based on user input. These queries operate on multiple MBOs through the use of joins.

SUPAttributeTest

In addition to allowing for arbitrary search criteria, the arbitrary find method lets the user specify a desired ordering of the results and object state criteria. A `SUPQuery` class is included in one of the client runtime libraries, `libclientrt.a`. 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.

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

Table 3. SUPQuery and Related Classes

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

Class	Description
SUPQueryResultSet	Provides for querying a result set for the dynamic query API.

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

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`.

An `SUPAttributeTest` defines a filter condition using an MBO attribute, and supports these conditions:

- IS_NULL
- NOT_NULL
- EQUAL
- NOT_EQUAL
- LIKE
- NOT_LIKE
- LESS_THAN
- LESS_EQUAL
- MATCH
- NOT_MATCH
- GREATER_THAN
- GREATER_EQUAL
- CONTAINS
- STARTS_WITH
- ENDS_WITH
- NOT_START_WITH
- NOT_END_WITH
- NOT_CONTAIN

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

Methods
```
add:(SUPTestCriteria*)operand;
```

The following example shows a detailed construction of the test criteria and join criteria for a query:

```
SUPQuery *query2 = [SUPQuery getInstance];
[query2 select:@"c.fname,c.lname,s.order_date,s.region"];
[query2 from:@"Customer":@"c"];
//
// Convenience method for adding a join to the query
//
//[query2 join:@"Sales_order":@"s":@"s.cust_id":@"c.id"];
//
// Detailed construction of the join criteria
SUPJoinCriteria *joinCriteria = [SUPJoinCriteria getInstance];
SUPJoinCondition* joinCondition = [SUPJoinCondition getInstance];
joinCondition.alias = @"s";
joinCondition.entity = @"Sales_order";
joinCondition.leftItem = @"s.cust_id";
joinCondition.rightItem = @"c.id";
joinCondition.joinType = [SUPJoinCondition INNER_JOIN];
[joinCriteria add:joinCondition];
query2.joinCriteria = joinCriteria;
//
// Convenience method for adding test criteria
//query2.testCriteria = (SUPTestCriteria*)[[SUPAttributeTest
// equal:@"c.fname":@"Douglas"] and:
[SUPAttributeTest
// equal:@"c.lname":@"Smith"]];
//
// Detailed construction of the test criteria
SUPCompositeTest *ct = [SUPCompositeTest getInstance];
ct.operands = [SUPObjectList getInstance];
[ct.operands add:[SUPAttributeTest equal:@"c.fname":@"Douglas"]];
[ct.operands add:[SUPAttributeTest equal:@"c.lname":@"Smith"]];
ct.operator = [SUPCompositeTest AND];
query2.testCriteria = (SUPTestCriteria*)ct;
SUPQueryResultSet* resultSet = [TestCRUD_TestCRUDDB
executeQuery:query2];
```

Dynamic Query

User can use query to construct a query SQL statement as he wants to query data from local database. This query may across multiple tables (MBOs).

```
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 = [SampleApp_SampleAppDB
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]]);
}
```

Paging Data

On low memory devices, retrieving up to 30,000 records from the database may cause the custom client to fail and throw an OutOfMemoryException.

Consider using the Query object to limit the result set.

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

SUPQueryResultSet

The `SUPQueryResultSet` class provides for querying a result set for the dynamic query API. `SUPQueryResultSet` is returned as a result of executing a query.

Example

This example shows how to filter a result set and get values by taking data from two mobile business objects, creating an 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"];
query.testCriteria = [SUPAttributeTest match:@"c.lname":@"Devlin"];
SUPQueryResultSet* resultSet = [SampleApp_SampleAppDB
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]]);
}
```

Retrieving Relationship Data

A relationship between two MBOs allows the parent MBO to access the associated MBO. If the relationship is bi-directional, it 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 SalesOrder on the customer ID column. The Orders application is parameterized to return order information for the customer ID.

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

Given an order, you can access its customer information.

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

Operations APIs

The create, update, and delete and related operations allow you to perform operations on data on the local client database, and to propagate that data to the Unwired Server.

Create Operation

The `create` operation allows the client to create a new record in the local database. To propagate the changes to the server, call `submitPending`.

(void)create

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

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

Example 2: Supports create operations on child entities.

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

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

```
while ([SampleApp_SampleAppDB hasPendingOperations])
        [NSThread sleepForTimeInterval:0.2];
```

Update Operation

The `update` operation updates a record in the local database on the device. 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:

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

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

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

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

```
SampleApp_Customer *customer = [ SampleApp_Customer find: 32]
SUPObjectList* orderlist = customer.orders;
SampleApp_sales_order* order = [orderlist item:0];
order.sales_rep  = @"Ram";
customer.state = @"MA" ;
[customer save];
[customer submitPending];
while ([SampleApp_SampleAppDB hasPendingOperations])
        [NSThread sleepForTimeInterval:0.5];
```

Delete Operation

The `delete` operation allows the client to delete a new record in the local database. 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:

```
SampleApp_Customer *customer = [ SampleApp_Customer find: 32]
[customer delete];
```

```
[customer submitPending];
while ([SampleApp_SampleAppDB hasPendingOperations])
        [NSThread sleepForTimeInterval:0.2];
```

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

```
SampleApp_Sales_order  *order = [SampleApp_Sales_order find: 32]
[order delete];
[order.customer submitPending]; //Call submitPending on the parent.
while ([SampleApp_SampleAppDB hasPendingOperations])
        [NSThread sleepForTimeInterval:0.2];
```

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

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

Other Operation

Operations other than `create`, `update`, or `delete` operations are called “other” operations.

This is an example of an "other" operation:

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

Multilevel Insert (MLI)

Multilevel insert allows a single synchronization to execute a chain of related insert operations. This example demonstrates a multilevel insert:

```
-(void)TestCreate
{
     long key1 = [SampleApp_KeyGenerator generateId];
     long key2 = [SampleApp_KeyGenerator generateId];
     [SampleApp_KeyGenerator submitPendingOperations];
     while ([SampleApp_SampleAppDB hasPendingOperations])
     {
         [NSThread sleepForTimeInterval:1];
     }
     SampleApp_Customer *c = [[ SampleApp_Customer alloc] init];
     c.id_ = [SampleApp_LocalKeyGenerator generateId];
```

```
     c.fname = @"Dorothi";
     c.lname = @"Scranton";
     c.address = @"One Money Street";
     c.city = @"smallVille";
     c.state = @"MA";
     c.zip = @"97429";
     c.phone = @"2112222345";
     c.company_name = @"iAnywhere";
     c.surrogateKey = key1;
     SUPObjectList *orderlist = [ SampleApp_Sales_orderList
getInstance];
     SampleApp_Sales_order *o1 = [[SampleApp_Sales_order alloc]
init];
     o1.id_ = [SampleApp_LocalKeyGenerator generateId];
     o1.order_date = [NSDate date];
     o1.fin_code_id = @"r1";
     o1.region = @"Eastern";
     o1.sales_rep = 902;
     o1.surrogateKey = key2;
     [ o1 setCustomer:c];
     [orderlist add:o1];
     [c setSalesOrders:orderlist];
     [c save];
     [c refresh];
     [c submitPending];
     assert(c.pending == YES);
     while ([SampleApp_SampleAppDB hasPendingOperations])
     {
          [NSThread sleepForTimeInterval:1];
     }
}
```

Pending Operation

There are five methods you can use to manage the pending state.

- **`(void)cancelPending`** – Cancels a pending record. A pending record is one that has been updated in the local client database, but not yet sent to the Unwired Server.

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

  ```
  [Customer cancelPendingOperations];
  ```
- **`(void)submitPending`** – Submits a pending record to the Unwired Server. For MBS, a replay request is sent directly to the Unwired Server.

  ```
  [customer submitPending];
  ```
- **`+(void)submitPendingOperations`** – Submits all data for all pending records to the Unwired Server. This method internally invokes the `submitPending` method.

  ```
  [Customer submitPendingOperations];
  ```
- **`+(void)submitPendingOperations: (NSString*)synchronizationGroup`** – Submits all data for pending records

from MBOs in this synchronization group to the Unwired Server. This method internally invokes the `submitPending` method.

```
[SampleApp_SampleAppDB submitPendingOperations:@”default”];
```

```
SampleApp_Customer *customer = [SampleApp_Customer 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];
```

Local Business Object

A business object can be either local or mobile. A local business object is a client-only object. Unlike a mobile business object, a local business object cannot be synchronized with the Unwired Server. Local business objects do not call `submitPending`, or perform a replay or import from the Unwired Server.

The following code example creates a row for a local business object called "clientObj", saves it, and finds it in the database.

```
//Create a client only  MBO...");
ClientObj *o = [ClientObj getInstance];
  o.attribute1 = @"This";
  o.attribute2 = @"is";
  o.attribute3 = @"a";
  o.attribute4 = @"client only mbo";
[o save];

//Read from the created MBO");
SUPObjectList *objlist = [ClientObj findAll];
MBOLogError(@"ClientObj MBO has %ld rows",[objlist size]);
  for(ClientObj *o in objlist)
MBOLogError([[o json:0] toString]);
```

Personalization APIs

Personalization keys allow the mobile user to define (personalize) certain input field values within the mobile application. The `PersonalizationParameters` class is generated automatically for managing personalization keys. 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.

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 after the device application terminates.

A personalization parameter can be a primitive or complex type. This is shown in the code example.

Getting and Setting Personalization Key Values

Consider a personalization key "pkcity" that is associated with the synchronization parameter "cityname". When a personalization parameter value is changed, the call to `save` automatically propagates the change to the server; you need not call `submitPending` after the `save`.

The following example shows how to get and set personalization key values:

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

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

Note: You are not required to call `submitPending` after `save`, as is the case with synchronization parameters.

Passing Arrays of Values, Objects

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

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

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

Object State APIs

The object state APIs include status indicator APIs for returning information about entities in the database, and a method to refresh the MBO entity in the local database.

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	Objective-C Type	Description
isNew	BOOL	Returns true if this entity is new (but has not been created in the client database).
isCreated	BOOL	Returns true if this entity has been newly created in the client database, and one the following is true: • The entity has not yet been submitted to the server with a replay request. • The entity has been submitted to the server, but the server has not finished processing the request. • The server rejected the replay request (replayFailure message received).
isDirty	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 was subsequently deleted.
isUpdated	BOOL	Returns true if this entity has been updated or changed in the database, and one of the following is true: • The entity has not yet been submitted to the server with a replay request. • The entity has been submitted to the server, but the server has not finished processing the request. • The server rejected the replay request (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, then 'C' (create), 'U' (update), 'D' (delete), 'P' (to indicate that this MBO is a parent in a cascading relationship for one or more pending child objects, but this MBO itself has no pending create, update or delete operations). If pending is false, then 'N'.

Name	Objective-C Type	Description
replayCounter	long	Returns a long value that is updated each time a row is created or modified by the client. This value is derived from the time in seconds since an epoch, and increases each time a row is changed. `int64_t result = [customer replayCounter];`
replayPending	long	Returns a long value. When a pending row is submitted to the server, the value of `replayCounter` is copied to `replayPending`. This allows the client code to detect if a row has been changed since it was submitted to the server (that is, if the value of`replayCounter` is greater than `replayPending`). `int64_t result = [customer replayPending];`
replayFailure	long	Returns a long value. When the server responds with a `replayFailure` message for a row that was submitted to the server, the value of `replayCounter` is copied to `replayFailure`, and `replayPending` is set to 0. `int64_t result = [customer replayFailure];`

Entity State Example

This table 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 the following entity behaviors:

- The `isDirty` flag is set if the entity changes in memory but is not yet written to the database. Once you save the MBO, this flag clears.
- The `replayCounter` value that gets sent to the Unwired Server is the value in the database before you call `submitPending`. After a successful replay, that value is imported from the Unwired Server.
- The last two entries in the table are two possible results from the operation; only one of these results can occur for a replay request.

Description	Flags/Values
After reading from the database, before any changes are made.	isNew=false isCreated=false isDirty=false isDeleted=false isUpdated=false pending=false pendingChange='N' replayCounter=33422977 replayPending=0 replayFailure=0
One or more attributes are changed, but changes not saved.	isNew=false isCreated=false isDirty=**true** isDeleted=false isUpdated=false pending=false pendingChange='N' replayCounter=33422977 replayPending=0 replayFailure=0

Description	Flags/Values
After `[entity save]` or `[entity update]` is called.	isNew=false isCreated=false isDirty=**false** isDeleted=false isUpdated=**true** pending=**true** pendingChange=**'U'** replayCounter=**33424979** replayPending=0 replayFailure=0
After `[entity submitPending]` is called to submit the MBO to the server	isNew=false isCreated=false isDirty=false isDeleted=false isUpdated=true pending=true pendingChange='U' replayCounter=33424981 replayPending=**33424981** replayFailure=0

Description	Flags/Values
Possible result: the Unwired Server accepts the update, sends an import and a `replayResult` for the entity, and the refreshes the entity from the database.	isNew=false isCreated=false isDirty=false isDeleted=false isUpdated=**false** pending=**false** pendingChange=**'N'** replayCounter=**33422977** replayPending=**0** replayFailure=0
Possible result: The Unwired Server rejects the update, sends a `replayFailure` for the entity, and refreshes the entity from the database	isNew=false isCreated=false isDirty=false isDeleted=false isUpdated=true pending=true pendingChange='U' replayCounter=33424981 replayPending=**0** replayFailure=**33424981**

Pending State Pattern

When a create, update, delete, or save operation is called on an entity in a message-based synchronization application, the requested change becomes pending. To apply the pending change, call `submitPending` on the entity, or `submitPendingOperations` on the mobile business object (MBO) class:

```
Customer *e = [Customer getInstance];
e.name = @"Fred";
e.address = @"123 Four St.";
[e create]; // create as pending
// Then do this....
[e submitPending]; // submit to server
// ... or this.
```

```
[Customer submitPendingOperations]; // submit all pending Customer
rows to server
```

`submitPendingOperations` submits all the pending records for the entity to the Unwired Server. This method internally invokes the `submitPending` method on each of the pending records.

The call to `submitPending` causes a JSON message to be sent to the Unwired Server with the `replay` method, containing the data for the rows to be created, updated, or deleted. The Unwired Server processes the message and responds with a JSON message with the `replayResult` method (the Unwired Server accepts the requested operation) or the `replayFailure` method (the server rejects the requested operation).

If the Unwired Server accepts the requested change, it also 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. These changes are written to the client database and marked as rows that are not pending. When the `replayResult` message is received, the pending row is removed, and the row remaining in the client database now contains data that has been imported from and validated by the Unwired Server. The Unwired Server may optionally send a log record to the client indicating a successful operation.

If the Unwired Server rejects the requested change, the client receives a `replayFailed` message, and the entity remains in the pending state, with its `replayFailed` attribute set to indicate that the change was rejected.

If the Unwired Server rejects the requested change, it also sends one or more log record messages to the client. The `SUPLogRecord` interface has the following getter methods to access information about the log record:

Method Name	Objective-C Type	Description
component	NSString*	Name of the MBO for the row for which this log record was written.
entityKey	NSString*	String representation of the primary key of the row for which this log record was written.

Method Name	Objective-C Type	Description
code	int32_t	One of several possible HTTP error codes: • 200 indicates success. • 401 indicates that the client request had invalid credentials, or that authentication failed for some other reason. • 403 indicates that the client request had valid credentials, but that the user does not have permission to access the requested resource (package, MBO, or operation). • 404 indicates that the client tried to access a nonexistent package or MBO. • 405 indicates that there is no valid license to check out for the client. • 500 to indicate an unexpected (unspecified) server failure.
message	NSString*	Descriptive message from the server with the reason for the log record.
operation	NSString*	The operation (create, update, or delete) that caused the log record to be written.
requestId	NSString*	The id of the replay message sent by the client that caused this log record to be written.
timestamp	NSDate*	Date and time of the log record.

If a rejection is received, the application can use the entity method `getLogRecords` to access the log records and get the reason:

```
SUPObjectList* logs = [e getLogRecords];
for(id<SUPLogRecord> log in logs)
{
  MBOLogError(@"entity has a log record:\n\
    code = %ld,\n\
    component = %@,\n\
    entityKey = %@,\n\
    level = %ld,\n\
    message = %@,\n\
    operation = %@,\n\
    requestId = %@,\n\
    timestamp = %@",
    [log code],
    [log component],
    [log entityKey],
    [log level],
    [log message],
    [log operation],
```

```
    [log requestId],
    [log timestamp]);
}
```

`cancelPendingOperations` cancels all the pending records for an entity. This method internally invokes the `cancelPending` method on each of the pending records.

Refresh

The `refresh` method allows the client to refresh the MBO entity from the local database.

(void)refresh

```
[order refresh];
```

where `order` is an instance of the MBO entity.

Clear Relationship Objects

The `clearRelationshipObjects` method releases relationship attributes and sets them to null. Attributes get filled from the client database on the next getter method call or property reference. You can use this method to conserve memory if an MBO has large child attributes that are not needed at all times.

(void)clearRelationshipObjects

Security APIs

Unwired Server supports encryption of client data and the database.

Encryption of Client Data

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

Encryption of the Database

The following methods set or change encryption keys for the database.

-(void)setEncryptionKey:(SUPString)value

Sets the encryption key for the database in `SUPConnectionProfile`. Call this method before any database operations.

```
[cp setEncryptionKey:@”test”];
```

+(void)changeEncryptionKey:(NSString*) newKey

Changes the encryption key to the `newKey` value and saves the `newKey` value to the connection profile. Call this method after the call to `createDatabase`.

```
[SampleApp_SampleAppDB changeEncryptionKey:@"newkey" ];
```

Configuring Application Security Using Keychain

An application can make use of security features that use Keychain as persistent storage for a database encryption key by using the `SUPKeyVault` APIs defined by the `SUPKeyVault` class.

The `SUPKeyVault` class controls setting a key to the keychain, retrieving a key from the keychain, encrypting/decrypting a key with an application PIN, locking/unlocking a key vault with a PIN, and PIN management. An application explicitly retrieves and saves a database encryption key using the `SUPKeyVault` APIs, then sets the retrieved encryption key to `SUPConnectionProfile`.

1. Modify the application to use `SUPKeyVault` to retrieve the database encryption key from Keychain at start-up:

```
SUPKeyVault * keyvault = [SUPKeyVault
getSUPKeyVault:MESSAGING_VAULT_ID];

// keyVault must be unlocked by the application before the
connection to server.

if ( [keyVault isLocked] )
{
  // Get the PIN from user through ENTER PIN dialog

  // Now unlock the KeyVault with the PIN
  result = [keyVault unlock: pin];
  if ( result == error )
  {
    // Take necessary actions
  }
}
NSData *dbKey = [keyVault key];

// start up Sybase messaging client after the keyVault is
unlocked.
NSInteger result = [SUPMessageClient start];
if (result == kSUPMessageClientSuccess)
{
...
}
```

2. Modify the application to set an encryption key to the current `SUPConnectionProfile`, to allow database operations to use this encryption key. Call these methods before performing any database operations:

```
SUPConnectionProfile *cp = [SampleApp_SampleAppDB
connectionProfile];
[cp setEncryptionKey:dbKey];
```

3. Modify the application to save the database encryption key to the Keychain by calling these methods:

```
if ( ![keyVault isLocked] )
{
```

```
    [keyVault setKey:dbKey];
}
```

Utility APIs

The iOS Client Object API provides utility APIs to support a variety of tasks.

- Writing and retrieving log records.
- Configuring log levels for messages reported to the console.
- Enabling the printing of server message headers and message contents, database exceptions, and `SUPLogRecords` written for each import.
- Viewing detailed trace information on database calls.
- Registering a callback handler to receive callbacks.
- Assigning a unique ID for an application which requires a primary key.
- Managing date/time objects for iOS through defined classes.
- Enabling Apple Push Notification to allow applications to provide push notifications to devices.

Using the Log Record APIs

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).

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

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

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

```
SUPQuery *query = [SUPQuery getInstance];
       SUPObjectList *loglist = [SampleApp_SampleAppDB
```

```
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 = [SampleApp_Sales_order findAll];
if([salesorders size] > 0)
{
  SampleApp_Sales_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 = [SampleApp_SampleAppDB 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 = [SampleApp_SampleAppDB
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]];
```

Viewing Error Codes in Log Records

You can view any EIS error codes and the logically mapped HTTP error codes in the log record.

For example, you could observe a "Backend down" or "Backend login failure" after the following sequence of events:

1. Deploying packages to Unwired Server.
2. Performing an initial synchronization.
3. Switching off the backend or change the login credentials at the backend.
4. Invoking a create operation by sending a JSON message.

```
JsonHeader
{"id":"684cbe16f6b740eb930d08fd626e1551","cid":"111#My1:1","ppm":
"eyJ1c2VybmFtZSI6InN1cEFkbWluIiwicGFzc3dvcmQiOiJzM3BBZG1pbiJ9","p
id":"moca://
Emulator17128142","method":"replay","pbi":"true","upa":"c3VwQWRta
W46czNwQWRtaW4=","mbo":"Bi","app":"My1:1","pkg":"imot1:1.0"}

JsonContent
{"c2":null,"c1":1,"createCalled":true,"_op":"C"}
```

The Unwired Server returns a response. The code is included in the `ResponseHeader`.

```
ResponseHeader
{"id":"684cbe16f6b740eb930d08fd626e1551","cid":"111#My1:1","loginFa
iled":false,"method":"replayFailed","log":
[{"message":"com.sybase.jdbc3.jdbc.SybSQLException:SQL Anywhere
Error -193: Primary key for table 'bi' is not unique : Primary key
value ('1')","replayPending":
0,"eisCode":"","component":"Bi","entityKey":"0","code":
500,"pending":false,"disableSubmit":false,"?":"imot1.server.LogReco
rdImpl","timestamp":"2010-08-26
14:05:32.97","requestId":"684cbe16f6b740eb930d08fd626e1551","operat
ion":"create","_op":"N","replayFailure":
0,"level":"ERROR","pendingChange":"N","messageId":200001,"_rc":
0}],"mbo":"Bi","app":"My1:1","pkg":"imot1:1.0"}

ResponseContent
{"id":100001}
```

Log Levels 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");
```

Server Log Messages

The generated code for a package contains an `MBODebugLogger` source and header file and an `MBODebugSettings.h` file. The `MBODebugLogger` class contains methods that enable printing of server message headers and message contents, database exceptions and `SUPLogRecords` written for each import.

The client application can turn on printing of the desired messages by modifying the `MBODebugSettings.h`. In the default configuration, setting "#define __DEBUG__" to true prints out the server message headers and database exception messages, but does not print the full contents of server messages.

Note: For more information, examine the `MBOLogger.h` and `MBOLogInterface.h` header files in the `includes` directory.

Tracing APIs

To see detailed trace information on database calls, including actual SQL statements sent to SQLite, a Debug build of your application can turn on or off the following macros in `MBODebugSettings.h`:

- **LOGRECORD_ON_IMPORT –** creates a log record in the database for each import of server data for an MBO.
- **PRINT_PERSISTENCE_MESSAGES –** prints to the console the database exception messages.
- **PRINT_SERVER_MESSAGES –** prints to the console the JSON headers of messages going to and from the Unwired Server. This allows you to see while debugging that an application is subscribing successfully to the Unwired Server, and that imports are being

sent from the Unwired Server. When this macro is defined, the contents of client-initiated "replay" messages are also printed to the console.

- **PRINT_SERVER_MESSAGE_CONTENT –** prints to the console the full contents of messages from the Unwired Server to the client. The messages include all the data being imported from the Unwired Server, and usually result in a large amount of printing. Developers may find it useful to print all the data during detailed debugging; doing so allows them detailed debugging to see the data coming from the Unwired Server. In general, do not turn this macro on, as doing so considerably slows the data import process.

Printing Log Messages

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

```
SUPQuery * query = [SUPQuery newInstancegetInstance];
SampleApp_LogRecordImplList* loglist =
(SampleApp_LogRecordImplList*)[SampleApp _ SampleAppDB
getLogRecords:query];
  for(SampleApp_LogRecordImpl* log in loglist)
  {
    MBOLogError(@"Log Record %llu: Operation = %@, Component = %@,
message = %@", log.messageId, log.operation,
log.component,log.message);
  }
```

generateGuid

You can use the `generateGuid` method (in the LocalKeyGenerator class) to generate an ID when creating a new object for which you require a primary key. This generates a unique ID for the package on the local device.

```
+ (NSString*)generateGuid;
```

Callback Handlers

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, an entity, or both. You can use `SUPDefaultCallbackHandler` as the base class. In your handler, override the particular callback you want to use (for example, `onImport`).

Because both the database and entity handler can be registered, your handler may get called twice for a mobile business object import activity. The callback is executed in the thread that is performing the action (for example, import). When you receive the callback, the particular activity is already complete.

The `SUPCallbackHandler` protocol consists of these callbacks:

- **onImport:(id)entityObject; –** invoked when an import is received.

- **onReplayFailure:(id)entityObject; –** invoked when a replay failure is received from the Unwired Server.
- **onReplaySuccess:(id)entityObject; –** invoked when a replay success is received from the Unwired Server.
- **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.
- **onSubscribeSuccess; –** invoked when a subscribe success message is received from the Unwired Server.
- **- (int32_t)onSynchronize:(SUPObjectList*)syncGroupList withContext: (SUPSynchronizationContext*)context; –** invoked when the synchronization status changes. This method is called by the database class `beginSynchronize` methods when the client initiates a synchronization, and is called again when the server responds to the client that synchronization has finished, or that synchronization failed.

  The `SUPSynchronizationContext` object passed into this method has a "status" attribute that contains the current synchronization status. The possible statuses are defined in the `SUPSynchronizationStatusType` enum, and include:

  - **SUPSynchronizationStatusSTARTING –** passed in when `beginSynchronize` is called.
  - **SUPSynchronizationStatusUPLOADING –** synchronization status upload in progress.
  - **SUPSynchronizationStatusDOWNLOADING –** synchronization status download in progress.
  - **SUPSynchronizationStatusFINISHING –** synchronization completed successfully.
  - **SUPSynchronizationStatusERROR –** synchronization failed.

  This callback handler returns `SUPSynchronizationActionCONTINUE`, unless the user cancels synchronization, in which case it returns `SUPSynchronizationActionCANCEL`. This code example prints out the groups in a synchronization status change:

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

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

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

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

     if(context != nil)
```

```
    {
        MBOLogInfo(@"context: %ld,
%@",context.status,context.userContext);
    } else {
        MBOLogInfo(@"context is null");
    }

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

    return SUPSynchronizationActionCONTINUE;
}
```

- **onSuspendSubscriptionFailure; –** invoked when a call to suspend fails.
- **onSuspendSubscriptionSuccess; –** invoked when a suspend call is successful.
- **onResumeSubscriptionFailure; –** invoked when a resume call fails.
- **onResumeSubscriptionSuccess; –** invoked when a resume call is successful.
- **onUnsubscribeFailure; –** invoked when an unsubscribe call fails.
- **onUnsubscribeSuccess; –** invoked when an unsubscribe call is successful.
- **onImportSuccess; –** invoked when `onImport` succeeds.
- **onMessageException:(NSException*e); –** invoked when an exception occurs during message processing. Other callbacks in this interface (whose names begin with "on") are invoked inside a database transaction. If the transaction is rolled back due to an unexpected exception, this operation is called with the exception (before the rollback occurs).
- **onTransactionCommit; –** invoked on transaction commit.
- **onTransactionRollback; –** invoked on transaction rollback.
- **onResetSuccess; –** invoked when reset is successful.
- **onSubscriptionEnd; –** invoked on subscription end.
- **onStorageSpaceLow; –** invoked when storage space is low.
- **onStorageSpaceRecovered; –** invoked when storage space is recovered.
- **onConnectionStatusChange:(SUPDeviceConnectionStatus)connStatus: (SUPDeviceConnectionType)connType:(int32_t)errCode:(NSString*)errString; –** the application should call the register callback handler with a database class, and implement the `onConnectionStatusChange` method in the callback handler. The API allows the device application to see what the error is in cases where the client cannot connect to the Unwired Server. `SUPDeviceConnectionStatus` and `SUPDeviceConnectionType` are defined in `SUPConnectionUtil.h`:

```
typedef enum {
  WRONG_STATUS_NUM = 0,
  // device connected
  CONNECTED_NUM = 1,
  // device not connected
  DISCONNECTED_NUM = 2,
  // device not connected because of flight mode
  DEVICEINFLIGHTMODE_NUM = 3,
  // device not connected because no network coverage
  DEVICEOUTOFNETWORKCOVERAGE_NUM = 4,
  // device not connected and waiting to retry a connection
  WAITINGTOCONNECT_NUM = 5,
```

```
  // device not connected becauseroaming was set to false
  // and device is roaming
  DEVICEROAMING_NUM = 6,
  // device not connected because of low space.
  DEVICELOWSTORAGE_NUM = 7
} SUPDeviceConnectionStatus;

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

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

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

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

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 = [TestCRUD_TestCRUDDB
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];
```

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).

Note: This API works only on iPhone devices, and does not work on iPod, iPod Touch, or a simulator.

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

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

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

```
@interface SUPPushNotification : NSObject
{

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

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

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

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

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

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

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

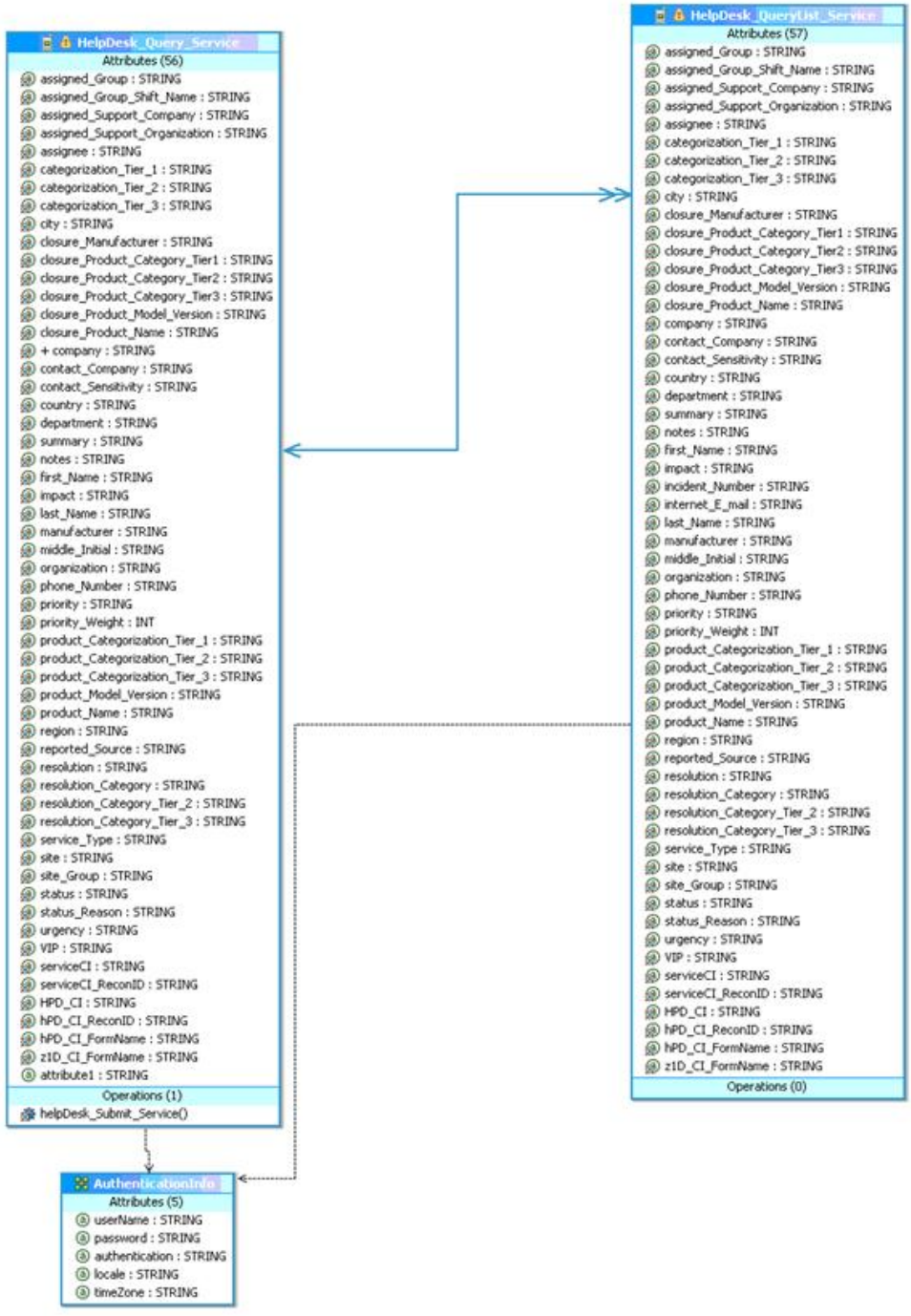
```
+(void)pushNotification:(UIApplication*)application
notifyData:(NSDictionary *)userInfo
```

Complex Attribute Types

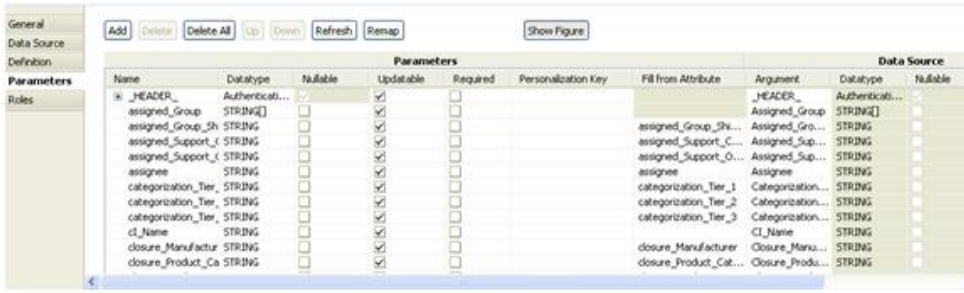
The MBO examples previously described have attributes that are primitive types (such as `int`, `long`, `string`), and make use of the basic database operations (create, update, and delete). To support interactions with certain back-end datasources, such as SAP® and Web services, an MBO may have more complex attributes: an integer or string list, a class or MBO object, or a list of objects. Some back-end datasources require complex types to be passed in as input parameters. The input parameters can be any of the allowed attribute types, including primitive lists, objects, and object lists.

In the following example, a Sybase Unwired Platform project is created to interact with a Remedy Web service back-end. The project includes two MBOs, `HelpDesk_Query_Service` and `HelpDesk_QueryList_Service`.

Note: Each project will have different requirements because each back-end datasource requires a different configuration for parameters to be sent to successfully execute a database operation.


HelpDesk_Query_Service
Attributes (56)
assigned_Group : STRING
assigned_Group_Shift_Name : STRING
assigned_Support_Company : STRING
assigned_Support_Organization : STRING
assignee : STRING
categorization_Tier_1 : STRING
categorization_Tier_2 : STRING
categorization_Tier_3 : STRING
city : STRING
closure_Manufacturer : STRING
closure_Product_Category_Tier1 : STRING
closure_Product_Category_Tier2 : STRING
closure_Product_Category_Tier3 : STRING
closure_Product_Model_Version : STRING
closure_Product_Name : STRING
+ company : STRING
contact_Company : STRING
contact_Sensitivity : STRING
country : STRING
department : STRING
summary : STRING
notes : STRING
first_Name : STRING
impact : STRING
last_Name : STRING
manufacturer : STRING
middle_Initial : STRING
organization : STRING
phone_Number : STRING
priority : STRING
priority_Weight : INT
product_Categorization_Tier_1 : STRING
product_Categorization_Tier_2 : STRING
product_Categorization_Tier_3 : STRING
product_Model_Version : STRING
product_Name : STRING
region : STRING
reported_Source : STRING
resolution : STRING
resolution_Category : STRING
resolution_Category_Tier_2 : STRING
resolution_Category_Tier_3 : STRING
service_Type : STRING
site : STRING
site_Group : STRING
status : STRING
status_Reason : STRING
urgency : STRING
VIP : STRING
serviceCI : STRING
serviceCI_ReconID : STRING
HPD_CI : STRING
hPD_CI_ReconID : STRING
hPD_CI_FormName : STRING
z1D_CI_FormName : STRING
attribute1 : STRING
Operations (1)
helpDesk_Submit_Service()
HelpDesk_QueryList_Service
Attributes (57)
assigned_Group : STRING
assigned_Group_Shift_Name : STRING
assigned_Support_Company : STRING
assigned_Support_Organization : STRING
assignee : STRING
categorization_Tier_1 : STRING
categorization_Tier_2 : STRING
categorization_Tier_3 : STRING
city : STRING
closure_Manufacturer : STRING
closure_Product_Category_Tier1 : STRING
closure_Product_Category_Tier2 : STRING
closure_Product_Category_Tier3 : STRING
closure_Product_Model_Version : STRING
closure_Product_Name : STRING
company : STRING
contact_Company : STRING
contact_Sensitivity : STRING
country : STRING
department : STRING
summary : STRING
notes : STRING
first_Name : STRING
impact : STRING
incident_Number : STRING
internet_E_mail : STRING
last_Name : STRING
manufacturer : STRING
middle_Initial : STRING
organization : STRING
phone_Number : STRING
priority : STRING
priority_Weight : INT
product_Categorization_Tier_1 : STRING
product_Categorization_Tier_2 : STRING
product_Categorization_Tier_3 : STRING
product_Model_Version : STRING
product_Name : STRING
region : STRING
reported_Source : STRING
resolution : STRING
resolution_Category : STRING
resolution_Category_Tier_2 : STRING
resolution_Category_Tier_3 : STRING
service_Type : STRING
site : STRING
site_Group : STRING
status : STRING
status_Reason : STRING
urgency : STRING
VIP : STRING
serviceCI : STRING
serviceCI_ReconID : STRING
HPD_CI : STRING
hPD_CI_ReconID : STRING
hPD_CI_FormName : STRING
z1D_CI_FormName : STRING
Operations (0)
AuthenticationInfo
Attributes (5)
userName : STRING
password : STRING
authentication : STRING
locale : STRING
timeZone : STRING

You can determine from viewing the properties of the create operation, `helpdesk_Submit_Service()`, that the operation requires parameters to be passed in. The first parameter, `_HEADER_`, is an instance of the `AuthenticationInfo` class, and the second parameter, `assigned_Group`, is a list of strings.



When you generate iOS code for this project, the generated code includes the `RemedyCU_AuthenticationInfo` class, in addition to the MBO classes `RemedyCU_HelpDesk_Query_Service` and `RemedyCU_HelpDesk_QueryList_Service`. The `AuthenticationInfo` class holds information that must be passed to the Unwired Server to authenticate database operations.

The project includes the create operation `helpdesk_Submit_Service`. Call this method instead of using the iOS MBO create method directly. The `helpdesk_Submit_Service` method is defined in `RemedyCU_HelpDesk_Query_Service.h`:

```
- (void)helpDesk_Submit_Service:
(RemedyCU_AuthenticationInfo*)_HEADER_
withAssigned_Group:(SUPNullableStringList*)assigned_Group
withCI_Name:(NSString*)cI_Name
withLookup_Keyword:(NSString*)lookup_Keyword
withResolution_Category_Tier_1:
(NSString*)resolution_Category_Tier_1
withAction:(NSString*)action
withCreate_Request:(NSString*)create_Request
withWork_Info_Summary:(NSString*)work_Info_Summary
withWork_Info_Notes:(NSString*)work_Info_Notes
withWork_Info_Type:(NSString*)work_Info_Type
withWork_Info_Date:(NSDate*)work_Info_Date
withWork_Info_Source:(NSString*)work_Info_Source
withWork_Info_Locked:(NSString*)work_Info_Locked
withWork_Info_View_Access:(NSString*)work_Info_View_Access
withMiddle_Initial:(SUPNullableStringList*)middle_Initial
withDirect_Contact_First_Name:(NSString*)direct_Contact_First_Name
withDirect_Contact_Middle_Initial:
(NSString*)direct_Contact_Middle_Initial
withDirect_Contact_Last_Name:(NSString*)direct_Contact_Last_Name
withTemplateID:(NSString*)templateID;
```

The following code example initializes a Remedy instance of the `HelpDesk_Query_Service` MBO on the device, creates the instance in the client database, and submits it to the Unwired Server. The example shows how to initialize the `AuthorizationInfo` class instance and the `assigned_Group` string list, and pass them as parameters into the create operation.

```
RemedyCU_AuthenticationInfo* authinfo;
  int64_t key= 0;
  authinfo = [RemedyCU_AuthenticationInfo getInstance];
  authinfo.userName=@"Francie";
  authinfo.password=@"password";
  authinfo.authentication=nil;
  authinfo.locale=nil;
  authinfo.timeZone=nil;

  SUPNullableStringList *assignedgrp = [SUPNullableStringList
getInstance];
  [assignedgrp add:@"Frontoffice Support"];

  RemedyCU_HelpDesk_Query_Service *cr =
[[RemedyCU_HelpDesk_Query_Service alloc] init];

cr.company = @"Calbro Services";

[cr helpDesk_Submit_Service:authinfo
  withAssigned_Group:assignedgrp
  withCI_Name:nil
  withLookup_Keyword:nil
  withResolution_Category_Tier_1:nil
  withAction:@"CREATE"
  withCreate_Request:@"YES"
  withWork_Info_Summary:[NSString stringWithFormat:@"create %@",
[NSDate date]]
  withWork_Info_Notes:nil
  withWork_Info_Type:nil
  withWork_Info_Date:nil
  withWork_Info_Source:nil
  withWork_Info_Locked:nil
  withWork_Info_View_Access:nil
  withMiddle_Initial:nil
  withDirect_Contact_First_Name:nil
  withDirect_Contact_Middle_Initial:nil
  withDirect_Contact_Last_Name:nil
  withTemplateID:nil];

[cr submitPending];
// wait for response from server
while([RemedyCU_RemedyCUDB hasPendingOperations])
  [NSThread sleepForTimeInterval:1.0];
```

Exceptions

Reviewing exceptions allows you to identify where an error has occurred during application execution.

Handling Exceptions

The iOS Client Object API defines server-side and client-side exceptions.

Server-Side Exceptions

A server-side exception occurs when a client tries to update or create a record and the Unwired Server throws an exception.

A server-side exception results in a stack trace appearing in the server log, and a log record (`LogRecordImpl`) being imported to the client with information on the problem. The client receives both the log record and a `replayFailed` message.

HTTP Error Codes

Unwired Server examines the EIS code received in a server response message and maps it to a logical HTTP error code, if a corresponding error code exists. If no corresponding code exists, the 500 code is assigned to signify either a Sybase Unwired Platform internal error, or an unrecognized EIS error. The EIS code and HTTP error code values are stored in log records.

The following is a list of recoverable and non-recoverable error codes. Beginning with Unwired Platform version 1.5.5, all error codes that are not explicitly considered recoverable are now considered unrecoverable.

Table 4. Recoverable Error Codes

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

Table 5. Non-recoverable Error Codes

Error Code	Probable Cause	Manual Recovery Action
401	Backend EIS credentials wrong.	Change the connection information, or backend user password.
403	User authorization failed on Unwired Server due to role constraints (applicable only for MBS).	N/A
404	Resource (table/webservice/BAPI) not found on Backend EIS.	Restore the EIS configuration.
405	Invalid license for the client (applicable only for MBS).	N/A

Error Code	Probable Cause	Manual Recovery Action
412	Backend EIS threw a constraint exception.	Delete the conflicting entry in the EIS.
500	SUP internal error in modifying the CDB cache.	N/A

Beginning with Unwired Platform version 1.5.5, error code 401 is no longer treated as a simple recoverable error. If the `SupThrowCredentialRequestOn401Error` context variable is set to true (which is the default), error code 401 throws a `CredentialRequestException`, which sends a credential request notification to the user's inbox. You can change this default 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

The following is a list of SAP error codes mapped to HTTP error codes. SAP error codes which are not listed map by default to HTTP error code 500.

Table 6. Mapping of SAP error codes to HTTP error codes

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

Client-Side Exceptions

The HeaderDoc for the iOS Client Object API lists the possible exceptions for the client.

Attribute Datatype Conversion

When a non-nullable attribute's datatype is converted to a non-primitive datatype (such as class NSNumber, NSDate, and so on), you must verify that the the corresponding property for the MBO instance is assigned a non-nil value, otherwise the application may receive a runtime exception when creating a new MBO instance.

A typical scenario is when an attribute exists in ASE's identity column with a numeric datatype. For example, for a non-nullable attribute with a decimal datatype, the corresponding datatype in the generated Objective-C MBO code is `NSNumber`. When creating a new MBO instance, ensure that you assign this property a non-nil value.

Operation Name Conflicts

Operation names that conflict with special field types are not handled.

For example, if an MBO has attributes named id and description, those attributes are stored with the name `id_ description_`. If you create an operation called "description" and generated Object-C code, you see an error during compilation because of conflicting methods in the classes.

Exception Classes

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

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.

MetaData and Object Manager API

The MetaData and Object Manager API allows you to access metadata for database, classes, entities, attributes, operations, and parameters.

SUPDatabaseMetaData

You can use the `SUPDatabaseMetaData` class 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.

SUPAttributeMetaData

The `SUPAttributeMetaData` class holds metadata for attributes such as attribute name, column name, type, and maxlength.

Code Example for Accessing Metadata

The following code example for a package named "SampleApp" shows how to access metadata for database, classes, entities, attributes, operations, and parameters.

```
NSLog(@"List classes that have metadata....");
SUPDatabaseMetaData *dmd = [SampleApp_ SampleAppDB 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]);
  }
}
```

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.

Subscribe Data

The subscribe method allows the application to subscribe to a server package.

+(void) subscribe

The preconditions for the subscribe are that 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.

A subscription message is sent to the Unwired Server and the application receives a subscription request result notification from the Unwired Server .

```
[SampleApp_SampleAppDB subscribe];
```

Unsubscribe Data

The `unsubscribe` method allows the application to remove the existing subscription from server. The device application must already have a subscription with the server.

+(void) unsubscribe

On success, 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.

On failure, the client application receives subscription request result notification from server as notification with a failure message.

```
[SampleApp_SampleAppDB unsubscribe];
```

Suspend Subscription

The `suspendSubscription` operation allows a device application to send a suspend request to the Unwired Server. This notifies the server to stop delivering data changes.

+(void) suspendSubscription

```
[SampleApp_SampleAppDB suspendSubscription];
```

Synchronize Data

The `beginSynchronize` methods send a message to the Unwired Server to synchronize data between the client and the server.

+(void) beginSynchronize

This method is used to synchronize all data.

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

This method synchronizes only those MBOs that are part of certain synchronization groups. The parameter `synchronizationGroups` is a list of `SUPSynchronizationGroup` objects representing the groups to be synchronized. The parameter `context` is a reference string that is referred to when the server responds to the synchronization request. See the discussion of the `onSynchronize` callback handler method in *Developer Reference for iOS > Reference > iPhone Client Object API > Utility APIs > Callback Handlers*.

```
[SampleApp_SampleAppDB beginSynchronize];
```

Resume Subscription

The `resumeSubscription` operation allows a device application to send a resume request to the Unwired Server. This request notifies the Unwired Server to resume sending data changes since the last suspension.

+(void) resumeSubscription

```
[SampleApp_SampleAppDB resumeSubscription];
```

Recover Subscription

The `recover` operation allows the device application to send a recover request. This notifies the Unwired Server to send down all the data related to the package.

+(void) recover

```
[SampleApp_SampleAppDB recover];
```

Start or Stop Background Synchronization

Message-based synchronization is performed at the package level. The generated package database class provides methods for starting and stopping the background processing of the incoming messages.

To start background synchronization:

```
[SampleApp_SampleAppDB startBackgroundSynchronization];
```

To stop background synchronization:

```
[SampleApp_SampleAppDB stopBackgroundSynchronization];
```

When an incoming message is processed, callbacks are triggered. See *Reference: Administration APIs > iPhone Client Object APIs > Message-Based Synchronization APIs > Callback Handlers* for information on how to register a callback handler.

Replay Results

The client application can call the `hasPendingOperations` method after a `submitpending` call to the server to wait for replay results. This method returns true if there are replay pending requests, otherwise, it returns false.

+(void)hasPendingOperations

```
while ([SampleApp_SampleAppDB hasPendingOperations])
       [NSThread sleepForTimeInterval:0.2];
```

The preceding code example waits indefinitely if the client application does not receive a replay result from the Unwired Server, and if a record has the `replayPending` flag set. To exit this loop after a particular time interval has passed, you can add a timer.

```
BOOL shouldWait = YES;
  long sleepTime = 1;
  long timeout = 10*60;
  while (shouldWait && (sleepTime < timeout))
  {
    shouldWait = [SampleApp_SampleAppDB hasPendingOperations];
    if (shouldWait)
    {
      [NSThread sleepForTimeInterval:0.2];
    }
    if (sleepTime <= timeout)
    {
    timeout = timeout - sleepTime;
    }
  }
  if (shouldWait) {
    MBOLogError(@"Cannot wait , Timeout");
  }
```

Messaging Client API

The Sybase Unwired Platform messaging client (`SUPMessageClient`) API is part of the `libclientrt` library. The messaging client is responsible for setting up a connection between the user application and the server, as well as sending client messages up to the Unwired Server and receiving the import messages sent down to the client.

The Messaging Client API consists of the following methods:

+(void)setAssertionState:(BOOL)hideAssertions;

Determines whether assertions should appear or not.

+(NSInteger)start

Starts the messaging client and connects to the Unwired Server. You must use the settings application to enter the Sybase Unwired Platform user preferences information, including server name, port, user name, and activation code.

The parameters server name, user name, serverport, companyID and activation correspond to the Unwired Server name, the user name registered with the Unwired Server, the port the Unwired Server is listening to, the company ID, and activation code, respectively. If a Relay Server is used, 'companyID' corresponds to the farm ID of the Relay Server.

To ensure that messages are routed to the correct client application, the messaging client code sends the application executable name (specifically, the first 16 characters of the `CFBundleExecutable` value from the application's `Info.plist`) to the Unwired Server. The Unwired Server requires that each application on a device (or simulator) connect to the Unwired Server with a different user name.

This call returns one of the following values as defined in SUPMessageClient.h.

- kSUPMessageClientSuccess
- kSUPMessageClientFailure
- kSUPMessageClientKeyNotAvailable
- kSUPMessageClientNoSettings

Note: Ensure that the package database exists (either from a previous run, or a call to [`SampleApp_SampleAppDB createDatabase`]) and that [`SampleApp_SampleAppDB startBackgroundSynchronization`] is called before calling [`SUPMessageClient start`].

The following code example shows the `start` method:

```
NSInteger result = [SUPMessageClient start];

if (result == kSUPMessageClientSuccess)
{
  //Continue with your application
}
// At this point, if the result is a NO, then the client
// application can decide to quit or throw a message alerting
```

```
// the user that the connection to the server was
// unsuccessful.
```

+(NSInteger)stop

Stops the messaging client.

```
[SUPMessageClient  stop];
```

+(NSInteger)restart

Restarts the messaging client. Returns YES when successful, otherwise, if the required preferences are not set, or an error occurred when restarting the client, returns NO.

```
NSInteger result = [SUPMessageClient restart];
```

+(BOOL)provisioned

Checks if all the required provisioning information is set. Returns NO when required preferences are not set, and *YES* when all the required information is set.

```
BOOL result = [SUPMessageClient provisioned];
```

+(int)status

Returns the last status received from messaging client, as one of the following values:

- **0 –** not started
- **1 –** started, not connected
- **2 –** started, connected

```
int result = [SUPMessageClient status];
```

Index

P

Q

R

S

T

U

X