Tutorial: iPhone Application Development using Custom Development

Sybase Unwired Platform 1.5.3

Contents

Introduction to Getting Started Tutorials

Getting started tutorials enable users of all levels to try Sybase® Unwired Platform with minimal setup. You can also use the tutorials to demonstrate system functionality and train users.

Overview of Getting Started Tutorials

The getting started tutorials demonstrate how to develop, deploy, and test mobile business objects, device applications, and message-based mobile workflow packages.

- 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*

The getting started tutorials demonstrate a cross section of basic functionality, which includes creating MBOs that can be used in replication-based or message-based synchronization; and using various Sybase Unwired WorkSpace development tools, independent development environments, and device types.

Table 1. Tutorial summary

Tutorials	Mobile business objects (MBOs)	Synchronization types	Development tools	Device types
Tutorial: Mobile Business Object Development	Create new MBOs	Replication-based	Sybase Unwired WorkSpace	N/A
Tutorial: BlackBerry Application Development using Device Application Designer	Reuse MBOs	Replication-based	Device Application Designer	BlackBerry

Tutorials	Mobile business objects (MBOs)	Synchronization types	Development tools	Device types
Tutorial: BlackBerry Application Development using Custom Development	Create new MBOs	Replication-based	Sybase Unwired WorkSpace	BlackBerry
Tutorial: iPhone Application Development using Custom Development	Create new MBOs	Message-based	Sybase Unwired WorkSpace	iPhone
Tutorial: Windows Mobile Application Development using Device Application Designer	Reuse MBOs	Replication-based	Device Application Designer	Windows Mobile
Tutorial: Windows Mobile Device Application Development using Custom Development	Create new MBOs	Message-based	Sybase Unwired WorkSpace	Windows Mobile
Tutorial: Mobile Workflow Package Development	Create new MBOs	Message-based	Mobile Workflow Forms Editor	Windows Mobile

Understanding the Unwired Platform Development Environment

Learn more from the getting started tutorials by understanding basic development environment concepts. Sybase Unwired Platform provides an Eclipse development environment.

Development in Eclipse

Sybase Unwired WorkSpace is a plug-in to your Eclipse development environment that provides tools for creating mobile applications.

Unwired WorkSpace includes back-end integration tools that connect Unwired Server to enterprise data sources, allowing you to create mobile business objects (MBOs) from the back-end business data model.

Developers can perform MBO code generation at any time and use this MBO model code along with the user interface code that users write in a native integrated development environment (IDE). This makes the code available to transition from the Unwired WorkSpace MBO development tool to the fully extensible and open development environment provided for device platforms from third-party vendors. Optionally, use the Device Application

Designer to create prototype device applications for BlackBerry and Windows Mobile devices.

Developers can use the Mobile Workflow Forms Editor to develop message-based mobile workflow packages for Windows Mobile and iOS devices.

Understanding Fundamental Mobile Development Concepts

Learn more from the getting started tutorials by understanding basic mobile development concepts.

Learn more about these concepts:

- *Fundamentals*
- *Sybase Unwired WorkSpace – Mobile Business Object Development*

Mobile Business Objects

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). MBOs are deployed to Unwired Server, and accessed from mobile device application clients. MBOs include:

- Implementation-level details – metadata columns that include information about the data from a data source.
- Abstract-level details – attributes that correspond to instance-level properties of a programmable object in the mobile client, and map to data source output columns. Parameters correspond to synchronization parameters on the mobile client, and map to data source arguments. For example, output of a SQL SELECT query are mapped as attributes, and the arguments in the WHERE clause are mapped as synchronization parameters, so that the client can pass input to the query.

  MBO operations include parameters that map to data source input arguments. Operation parameters determine information a client passes to the enterprise information system (EIS).
- Relationships – defined between MBOs by linking attributes and parameters in one MBO, to attributes and parameters in another MBO.

You can define MBOs using either a top-down approach—first designing attributes and parameters, then binding them to a data source; or a bottom-up approach—first specifying a data source, then automatically generating attributes and parameters from it.

A mobile application package includes MBOs, roles, and data source connection mappings, and other artifacts that are delivered to the Unwired Server during package deployment.

Synchronization Methods

Developers can use either replication-based or message-based synchronization to move data and transactions between device application clients and Unwired Server.

The choice depends on the target device platform, application requirements, target platform, and the nature of data changes and activity between Unwired Server and clients, for example, mobile workflow forms always use message-based synchronization.

Unwired Server manages and maintains data freshness between multiple data sources and device application clients through synchronization.

Application Types

Sybase Unwired Platform supports two choices for application type. First is the native application type, and the other is the container-based business workflow type.

The native application model enables the developer to write custom code (C#, Java, or Objective-C, depending on the platform) to create a device application. The native application model is supported on BlackBerry, iOS, Windows Mobile, and Windows platforms. The choice depends on the functionality desired in the application, and the need to access third-party and platform-provided APIs. Optionally, use the Device Application Designer to create prototype device applications for BlackBerry and Windows Mobile devices.

The business workflow model offers a fast and simple way to build applications that support simple business workflows, such as approvals and requests. The workflow model is supported on iOS, Windows Mobile, and Windows platforms.

Data Sources

A data source is the enterprise information system where data is retrieved from and transactions are executed. A connection profile is a design-time connection to a data source. Connection profiles are created to specific data source by providing connection information such as host, port, login, and password among others. The connection profiles are used to define MBOs and operations, and mapped to existing, or used to create new, server connections when the package is deployed to Unwired Server..

Unwired Platform hides the interaction complexity with datasource-specific protocols, such as JDBC™ for database and SOAP for Web services.

Unwired Platform currently supports multiple EIS connection types. See *Supported Third-Party Software and Hardware* for information.

Switching Between Developer Profiles

Switch between basic and advanced developer profiles in the Mobile Application Diagram.

If you do not see an Unwired WorkSpace feature (wizard, property, or WorkSpace Navigator item) that you expect or need, switch to the advanced developer profile, or modify developer profile settings. To use backend data sources other than those supplied by Sybase Unwired Platform, you must switch to the advanced developer profile to see the Server Connection Mapping page when deploying the Mobile Business Object package.

1. Right-click in the Mobile Application Diagram and select **Switch Developer Profile > Basic/Advanced**.
2. You can also select **Window > Preferences > Sybase, Inc. > Mobile Development > Developer Profile** to directly view or modify the developer profile preference settings. Basic is the default developer profile.

Task Flow

This tutorial shows you how to develop a device application for an Apple iPhone device.

Table 2. Eclipse tutorials

Task	Goals	Steps required to complete the task
Getting Started	• Install all required WorkSpace components and external resources. • Start Unwired Server. • Open the Mobile Development perspective, and become familiar with the views of the perspective, the Mobile Application Diagram, and the Device Application Designer.	• Install Sybase Unwired Platform 1.5.2. • Install Sybase Unwired Platform 1.5.3. • Start the Unwired Platform services. • Start Sybase Unwired Workspace. **Note:** These tasks are prerequisites for all the other tutorials. You need to perform them only once.
Developing a Device Application	• Create an iPhone device application, and run it on the iPhone Simulator.	• Generate the Object API code for iPhone. • Set up the iPhone client in Xcode. • Register the iPhone simulator in Sybase Control Center. • Create the SUP101CallbackHandler file. • Create the iPhone application user interface. • Deploy the iPhone application to the simulator.

Getting Started

Goal: Install and learn about Sybase Unwired Platform and its associated components.

The following tasks are required, unless otherwise noted, for all tutorials, but you need to perform them only once.

1. *Installing Sybase Unwired Platform* on page 9
2. *Starting Unwired Platform Services* on page 10
3. *Starting Sybase Unwired WorkSpace* on page 10
4. (optional) *Learning the Basics* on page 11

Installing Sybase Unwired Platform

Goal: Install Sybase Unwired Platform.

Install these Sybase Unwired Platform components:

- Data Tier
- Unwired Server
- Unwired WorkSpace
- Device Application Designer
- Windows Mobile .NET components (for developing device applications in Visual Studio)

If Unwired Platform is already installed and any of these components are missing:

1. Start the Sybase Unwired Platform installer.
2. Follow the instructions in the installation wizard.
3. Select the required components, and complete the installation.

For complete installation instructions, see the *Sybase Unwired Platform Installation Guide* and *Release Bulletin*.

Installing the 1.5.3 EBF

Install Sybase Unwired Platform 1.5.2 GA before applying version 1.5.3.

Prerequisites

Back up your current Sybase Unwired Platform 1.5.2 files before applying the EBF.

Task

1. Go to *http://www.sybase.com* and select **Support > EBFs/Maintenance**
2. Enter your Sybase login information and click **Login**.
3. In **Search for a specific EBF**, enter `18245`, and click **Go**.
4. Read the end user license agreement and click **Continue** if you agree with the terms.
5. Follow the steps in the `README.html` file to install the 1.5.3 EBF.

Starting Unwired Platform Services

Goal: Start Unwired Server and the sample database.

In Windows, select **Start > Programs > Sybase > Unwired Platform<version> > Start Unwired Platform Services** .

Starting Sybase Unwired WorkSpace

Goal: Start Unwired WorkSpace.

1. In Windows, select **Start > Programs > Sybase > Unwired Platform<version> > Unwired WorkSpace**.

   Sybase Unwired WorkSpace opens, and displays the Welcome page with links to product information, and to the product.
2. To read more about Sybase Unwired WorkSpace concepts and tasks, select **Help > Help Contents** from the main menu.

Connecting to Sybase Control Center

Goal: Open the Web-based Sybase Control Center administration console to manage Unwired Server and its components.

From Sybase Control Center, you can:

- View servers and their status
- Start and stop a server
- View server logs
- Deploy a mobile application package
- Set role mappings

1. Select **Start > Programs > Sybase > Sybase Control Center**.

   Note: If Sybase Control Center does not launch, make sure that the Sybase Unified Agent service is started. See the Installation Guide for details.

2. Log in using the default login:
   - User Name – `supAdmin`
   - Password – `s3pAdmin`

   Logging in to Sybase Control Center (SCC) allows you access to Unwired Platform administration features that you have been authorized to use. Administrators of any Sybase product can log into SCC. However, only users assigned to the Super Administrator or Domain Administrator roles for Unwired Platform can log in to Unwired Server from Sybase Control Center.

   Logging in to SCC only allows you access to the SCC interface. If Unwired Server has not been authenticated, you will not be able to see or administer any resources.
3. Select **Help** > **Online Documentation** for additional information on configuring, managing, and monitoring Unwired Server.

Learning the Basics

Goal: Learn about Sybase Unwired WorkSpace and how to access help.

Prerequisites

Start Unwired WorkSpace.

Task

1. From the Welcome page, select any of the links to familiarize yourself with the Unwired WorkSpace environment.

   To close this page, click the **X**. You can reopen this page by selecting **Help** > **Welcome**.
2. Select **Start Development** to access the Sybase Unwired WorkSpace development environment. Look at the area (window or view) that you will be working in to access, create, define, and update mobile business objects (MBOs).

View	Description
WorkSpace Navigator	This view displays mobile application project folders, each of which contains all project-related resources in subfolders, including MBOs, data source references to which the MBOs are bound, personalization keys, and so on. Use this view to review and modify MBO-related properties.

View	Description
Enterprise Explorer	A window that provides functionality to connect to various enterprise back-end systems; for example, database servers, SAP servers, and Sybase Unwired Server.
Mobile Application Diagram	A graphical editor where you create and define mobile business objects. Use the Mobile Application Diagram to create MBOs (including attributes and operations), then define relationships with other MBOs. You can: • Create MBOs in the Mobile Application Diagram using Palette icons and menu selections – either bind or defer binding to a data source, when creating an MBO. For example, you may want to model your MBOs before creating the data sources to which they bind. This is sometimes called the top-down approach. • Drag items from Enterprise Explorer and drop them onto the Mobile Application Diagram to create the MBO – quickly creates the operations and attributes automatically based on the data source being dropped on the Mobile Application Diagram. This is sometimes called the bottom-up approach. Each new mobile application project generates an associated Mobile Application Diagram.
Palette	Access the Palette from the Mobile Application Diagram. It provides controls, such as the ability to create MBOs, add attributes and operations, and define relationships, by dragging and dropping the corresponding icon onto the Mobile Application Diagram or existing MBO.
Properties view	Select an object in the Mobile Application Diagram to display and edit its properties in the Properties view. You cannot create an MBO from the Properties view, but generally, most development and configuration is performed here.

View	Description
Outline view	Displays an outline of the file that is currently open in the editor area, and lists structural elements. The contents are editor-specific.
Problem view	Displays problems, errors, or warnings that you may encounter.

3. To access the online help, select **Help** > **Help Contents** from the main menu bar.
4. Expand any of the documents that appear in the left pane. Some documents are for Sybase Unwired Platform, while others are for the Eclipse development environment.

Developing an iPhone Application

Goal: Generate Object API code for the iPhone platform, develop an iPhone device application with code, and test its functionality.

Prerequisites

- Complete the *Getting Started* section of this tutorial.
- *Tutorial: Developing Database Mobile Business Objects*
- Supported platforms include:
  - MacOS 10.6 (Snow Leopard), Xcode 3.2.3 (MacBook or iMac)
  - iPhone SDK 4.0.x (earlier iPhone SDKs are not supported in Sybase Unwired Platform 1.5.3)

Task

The device application communicates with the database mobile business objects that are deployed to Unwired Server.

1. Open the SUP101 Mobile Application Project if it is not already open:

   In WorkSpace Navigator, right-click the **SUP101** folder and select **Open in Diagram Editor**.
2. Deploy the database mobile business objects.
3. Generate the Object API code for iPhone.
4. Set up the iPhone client application in Xcode.
5. Register the iPhone simulator in Sybase Control Center.
6. Create the SUP101 CallbackHandler file.
7. Add the SubscribeController View controller.
8. Add the CustomerListController.
9. Add the DetailController.
10. Deploy the iPhone application to the simulator.

Deploying the Database Mobile Business Objects

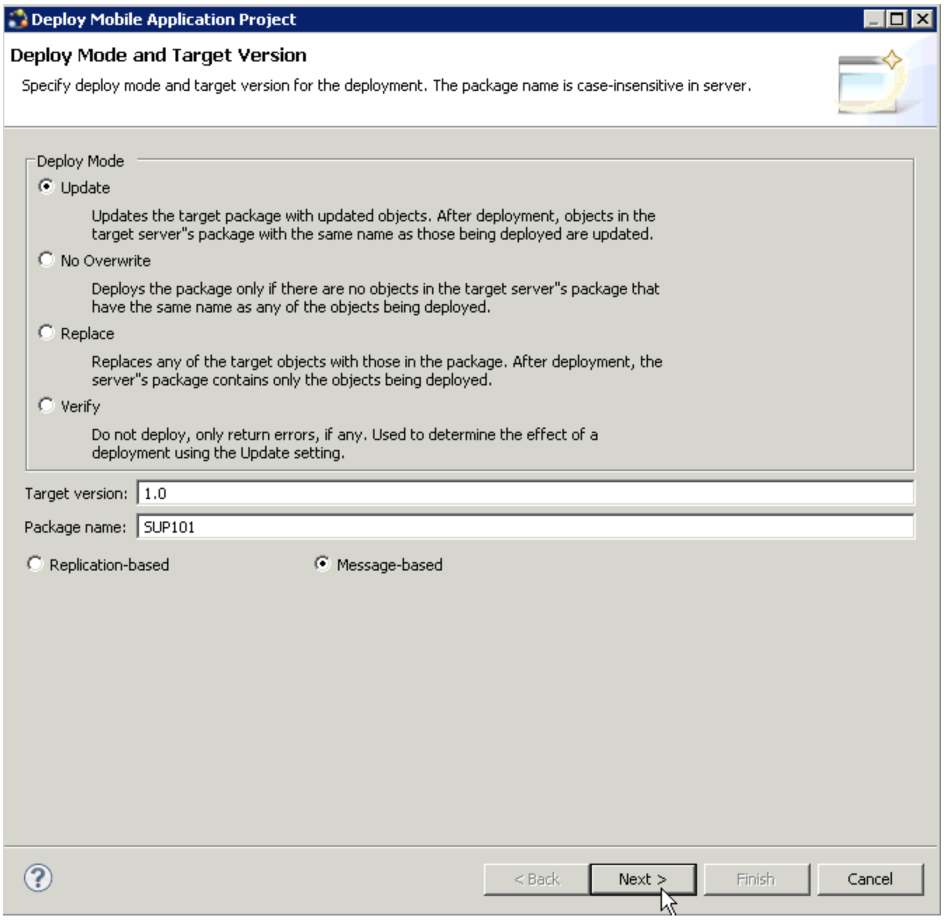
Goal: Deploy the project that contains the database mobile business objects to the server.

Prerequisites

Finish the *Tutorial: Developing Database Mobile Business Objects*. You must be connected to both the sampledb database and Unwired Server.

Task

1. Right-click in the SUP101 Mobile Application Diagram , and select **Deploy Project**.
2. On the first page of the Deploy Mobile Application Project, accept the defaults, select **Message-based**, and click **Next**.



Deploy Mode and Target Version

Specify deploy mode and target version for the deployment. The package name is case-insensitive in server.

Deploy Mode

(●) Update

Updates the target package with updated objects. After deployment, objects in the target server''s package with the same name as those being deployed are updated.

( ) No Overwrite

Deploys the package only if there are no objects in the target server''s package that have the same name as any of the objects being deployed.

( ) Replace

Replaces any of the target objects with those in the package. After deployment, the server''s package contains only the objects being deployed.

( ) Verify

Do not deploy, only return errors, if any. Used to determine the effect of a deployment using the Update setting.

Target version: 1.0

Package name: SUP101

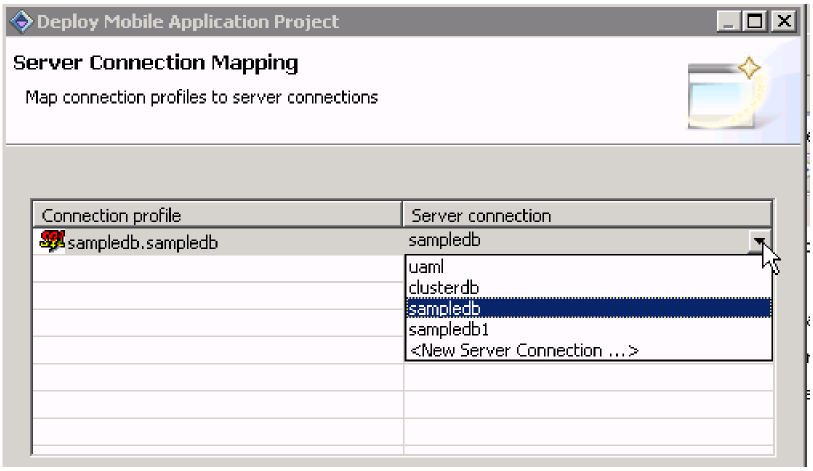
( ) Replication-based (●) Message-based

3. On the Contents page, select the **customer** and **sales_order** MBOs and click **Next**.
4. On the Package Jars page, click **Next**.

Note: This window appears only if you are using the Advanced developer profile.

5. On the Target Server page, from the list of available servers, select **My Unwired Server** and click **Refresh** (or connect to the server, if not already connected).

   Once connected, accept the default Domain and Security configuration settings, and click **Next**.
6. If you have multiple server connections, you see the Server Connection Mapping page. Select the **sampledb** server connection and click **Finish**.


Deploy Mobile Application Project
Server Connection Mapping
Map connection profiles to server connections
Connection profile
Server connection
sampledb.sampledb
sampledb
uaml
clusterdb
sampledb
sampledb1
<New Server Connection ...>

7. When the Deployment status window shows the deployment was successful, click **OK**.
8. Connect to Unwired Server and view the deployed project.
   a) In the Enterprise Explorer, click **My Unwired Server**.

      My Unwired Server is a default Unwired Server connection profile that provides access to Unwired Server, which you started in a previous step.
   b) Expand **Domains** > **default** > **Packages**. The server package *sup101:1.0*, into which you deployed the MBOs, appears in the Packages folder. The two MBOs appear in the `Mobile Business Objects` folder.

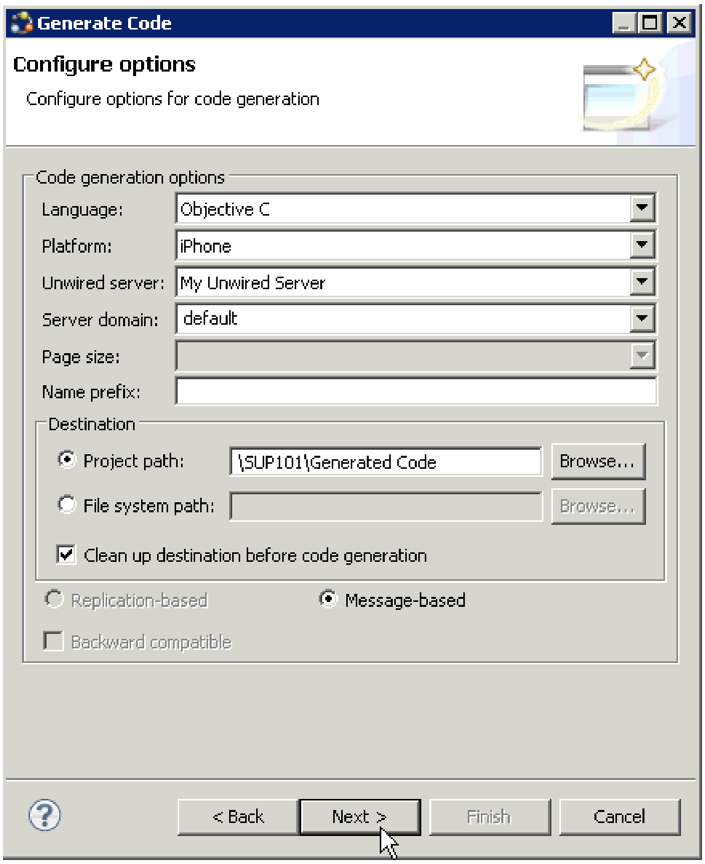
Generating Object API Code

Goal: Launch the code generation wizard and generate the Object API code for a message-based iPhone application.

1. Right-click in the SUP101 Mobile Application Diagram and select **Generate Code**.
2. In the code generation wizard, accept the default, **Continue without a configuration**, and click **Next**.
3. Enter these configuration options and click **Next**:

Option	Description
Language	Select **Objective-C**.
Platform	Accept the default, **iPhone**.
Unwired server	Select **My Unwired Server**.
Server domain	Accept **default**.

Option	Description
Name prefix	The prefix for the generated files. Leave blank.
Project path	Accept the default or enter a different location for the generated project files.
Clean up destination before code generation	Select this option to clean up the destination folder before generating the device client files.
Message-based	Selected by default when you select Objective-C as the language.



Configure options

Configure options for code generation

Code generation options

Language: Objective C

Platform: iPhone

Unwired server: My Unwired Server

Server domain: default

Page size:

Name prefix:

Destination

[✓ Project path:] \SUP101\Generated Code

File system path:

[✓] Clean up destination before code generation

Replication-based [✓ Message-based]

Backward compatible

4. In the next window, select the mobile business objects for which to generate the metadata classes and click **Finish**.
   This generates Objective-C code into the specified output location.

Setting Up an iPhone Client Application in Xcode

Goal: Set up an iPhone client application in the Xcode IDE.

Prerequisites

Create and deploy your mobile business objects, and generate Objective-C code into an output location.

Note: Ensure the directory where Sybase Unwired Platform is installed is a shared directory so you can access it from your Mac.

Task

Note: This tutorial was developed using Xcode 3.2.3 and iPhone SDK 4.0. If you use a different version of Xcode, some steps may vary. For more information on Xcode, refer to the Apple Developer Connection: *http://developer.apple.com/tools/Xcode/*.

1. In the Xcode IDE, select **Menu** > **New Project**, then select the **Windows-based Application** template.

   The project you create sets up the basic application environment.
2. In the new SUP101 project, in Active SDK, select **iPhoneSimulator 4.0** and set the Active Configuration to Debug.
3. Connect to the Microsoft Windows machine where Sybase Unwired Platform is installed:
   a) From the Apple menu, select **Go** > **Connect to Server**.
   b) Enter the name or IP address of the machine, for example, `smb://<machine DNS name>` or `smb://<IP Address>`.

      You see the shared directory.
4. In the `Home` directory on your Mac, create a new folder named `SUP101`.
5. Copy the `includes` and `libs` folders from `<Unwired Platform Installation>\Servers\UnwiredServer\ClientAPI\ObjectiveC` to the `Home/SUP101` directory on your Mac.
6. Copy the generated code for the SUP101 mobile application project from your Microsoft Windows environment, for example, `C:\Documents and Settings\administrator\workspace\SUP101\Generated Code`, to the `Home/SUP101` directory on your Mac.
7. Add the generated *.h and *.m files to the Xcode project.
   a) In the Xcode Groups & Files pane, right-click the **SUP101** project and select **Add** > **Existing Files**.

b) Select the `Generated Code` folder you copied to your `Home` directory (`Home/SUP101/Generated Code`) and click **Add**.
c) Select the **Copy items into destination group's folder (if needed)** option and click **Add**.

8. Add the `libclientrt.a`, `libSUPObj.a`, and `libMO.a` to your Xcode project:

a) In the Xcode Groups & Files pane, right-click **SUP101** and select **Add** > **Existing Files**.
b) Navigate to the directory where you copied the libraries, `Home/SUP101/libs`.
c) Open the `Debug-iphonesimulator` folder, select the `libclientrt.a`, `libSUPObj.a`, and `libMO.a` libraries and click **Add**.

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

d) Select the **Copy items into destination group's folder (if needed)** option and click **Add**.

9. Add `Settings.bundle` to the Xcode project.

This allows the iPhone device client user to use the Settings application to input his or her user preference information such as server name, server port, user name, and activation code.

a) In the Xcode Groups & Files pane, right-click **Resources**, and select **Add** > **Existing Files**.
b) Navigate to the `includes` directory that you copied from the `<Unwired Platform Installation>\Servers\UnwiredServer\ClientAPI\ObjectiveC` to the `Home/SUP101` directory on your Mac, select `Settings.bundle`, and click **Add**.
c) Select the **Copy items into destination group's folder (if needed)** option and click **Add**.

10. Enter the Xcode project Header Search Paths:

a) Select **Project** > **Edit Active Target SUP101**.
b) Click **Build**.
c) Scroll down to the Search Paths section and in **Header Search Paths**, verify the location where you copied the include files, `Home/SUP101/includes/**`.

11. Click **General**, then click the + icon at the bottom-left corner of the window to add these libraries from the SDK to the project:

- Security.framework
- AddressBook.framework
- QuartzCore.framework
- CoreFoundation.framework
- libicucore.A.dylib

- libz.1.2.3.dylib
- libstdc++.dylib

12. Click the **Build** tab, go to **Library Search Paths**, and remove any path that points to `libstdc++`.

Registering the iPhone Simulator in Sybase Control Center

Goal: Register the iPhone Simulator in Sybase Control Center.

Prerequisites

Connect to Sybase Control Center.

Task

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. In the left pane, select **Device Users**.
4. In the right pane, click **Devices**.
5. Click **Register**.
6. In the Register Device window, enter the required information:
   - User name – `user1`
   - Server name – *<localhost.sybase.com>*

   Note: The information should match the input on the client and "localhost.sybase.com" should be the actual name of your machine and domain.

   - Port – the Unwired Server port, `5001`.
   - Farm ID – `0`
   - Activation code – `123`

Creating the SUP101CallbackHandler File

Goal: Configure the SUP101CallBackHandler file.

There are two threads involved in the SUP101 application—the main thread, which is driven by the client application user interface controller, and the mobile object client access thread, which takes charge of message transportation with the server and synchronization with the application through the mobile object.

1. In the SUP101 Xcode project, select **File > New File.**

2. Select **Objective-C Class** and click **Next**.
3. In File Name, enter `SUP101CallbackHandler`, select the **Also create SUP101CallbackHandler.h** option, and click **Finish**.
4. In the new `SUP101CallbackHandler.h` file, enter the code for the callback handler. For example:

```
#import "SUPDefaultCallbackHandler.h"

@interface SUP101CallbackHandler:SUPDefaultCallbackHandler
{
    SUPInt field_importCount;
    SUPInt field_replaySuccessCount;
    SUPInt field_replayFailureCount;
    SUPInt field_searchSuccessCount;
    SUPInt field_searchFailureCount;
    SUPInt field_loginSuccessCount;
    SUPInt field_importSuccessCount;
}

+ (SUP101CallbackHandler*)newInstance;
- (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)searchSuccessCount;
- (void)setSearchSuccessCount:(SUPInt)_searchSuccessCount;
@property(assign) SUPInt searchSuccessCount;
- (SUPInt)searchFailureCount;
- (void)setSearchFailureCount:(SUPInt)_searchFailureCount;
@property(assign) SUPInt searchFailureCount;
- (SUPInt)loginSuccessCount;
- (void)setLoginSuccessCount:(SUPInt)_loginSuccessCount;
@property(assign) SUPInt loginSuccessCount;
- (SUPInt)importSuccessCount;
- (void)setImportSuccessCount:(SUPInt)_importSuccessCount;
@property(assign) SUPInt importSuccessCount;
- (void)onImport:(id)theObject;
- (void)onReplayFailure:(id)theObject;
- (void)onReplaySuccess:(id)theObject;
- (void)onSearchFailure:(id)theObject;
- (void)onSearchSuccess:(id)theObject;
- (void)onLoginSuccess;
- (void)onSubscribeSuccess;
- (void)onUnsubscribeSuccess;
- (void)onImportSuccess;
- (void)dealloc;

@end
```

5. In the new `SUP101CallbackHandler.m` file, enter the code for the callback handler. For example:

```
#import "SUP101CallbackHandler.h"

@implementation SUP101CallbackHandler

+ (SUP101CallbackHandler*)newInstance
{
   SUP101CallbackHandler* _me_1 = [[SUP101CallbackHandler alloc]
init];
    [_me_1 autorelease];
    return _me_1;
}

- (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)searchSuccessCount
{
    return field_searchSuccessCount;
}

- (void)setSearchSuccessCount:(SUPInt)_searchSuccessCount
{
    field_searchSuccessCount = _searchSuccessCount;
}

- (SUPInt)searchFailureCount
```

```
{
    return field_searchFailureCount;
}

- (void)setSearchFailureCount:(SUPInt)_searchFailureCount
{
    field_searchFailureCount = _searchFailureCount;
}

- (SUPInt)loginSuccessCount
{
    return field_loginSuccessCount;
}

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

- (void)onReplaySuccess:(id)theObject
{
    self.replaySuccessCount = self.replaySuccessCount + 1;


MBOLogInfo(@"==================================================");
    MBOLogInfo(@"Replay Successful");

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

}

- (void)onSearchFailure:(id)theObject
{
    self.searchFailureCount = self.searchFailureCount + 1;
}

- (void)onSearchSuccess:(id)theObject
{
    self.searchSuccessCount = self.searchSuccessCount + 1;
}

- (void)onLoginSuccess
{

MBOLogInfo(@"================================================");
    MBOLogInfo(@"Login Successful");

MBOLogInfo(@"=================================================");
    self.loginSuccessCount++;
}
- (void)onSubscribeSuccess
{

MBOLogInfo(@"================================================");
    MBOLogInfo(@"Subscribe Successful");

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

}

- (void)onUnsubscribeSuccess
{

MBOLogInfo(@"================================================");
    MBOLogInfo(@"Unsubscribe Successful");

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

}
- (void)onImportSuccess
{

MBOLogInfo(@"================================================");
    MBOLogInfo(@"import ends Successful");

MBOLogInfo(@"=================================================");
    self.importSuccessCount++;
}

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

@end
```

6. Save the `SUP101CallbackHandler.m` and `SUP101CallbackHandler.h` files.

Creating the User Interface

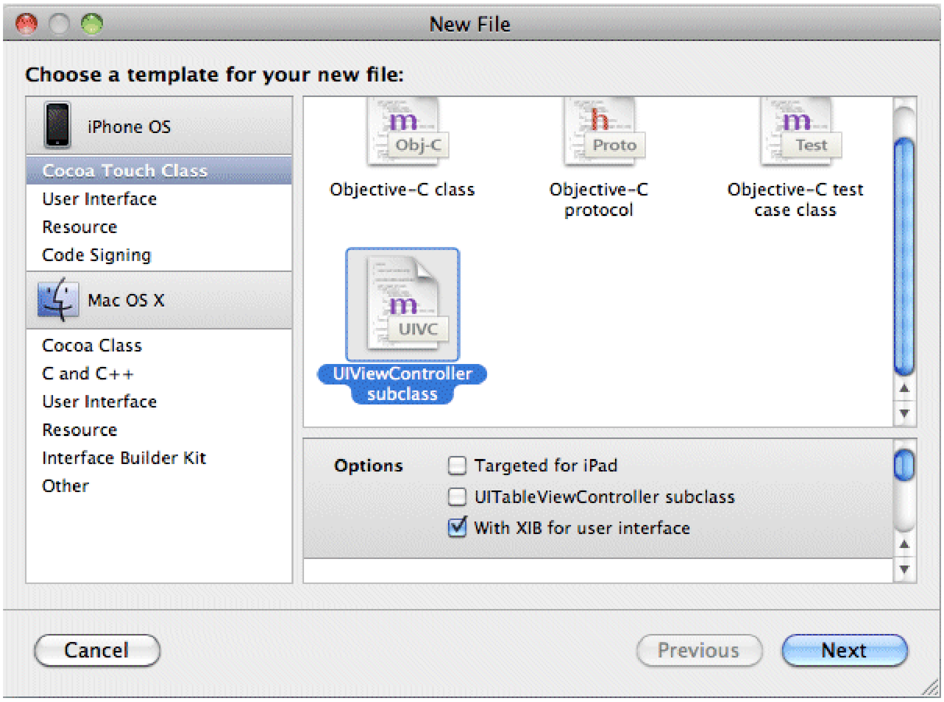
Use Interface Builder to create and configure the user interface for the SUP101 application.

Adding the SubscribeController View Controller

Goal: Create a view controller that functions as the root view screen for the SUP101 mobile application.

When you create the user interface, you assign a target action to a control object—in this example a Subscribe button so that a message (the action) is sent to another object (the target) in response to a user event, for example, a touch on the button. The view controller manages and configures the view when asked.

1. In the SUP101 Xcode project, select **File** > **New File**.
2. Select **Cocoa Touch Class**, **UIViewController subclass**, and **With XIB for user interface**. Then click **Next**.



Options
- [ ] Targeted for iPad
- [ ] UITableViewController subclass
- [✓] With XIB for user interface

The "With XIB for user interface" option creates a NextStep Interface Builder (nib) file to go with the view controller and adds it to the SUP101 Xcode project.

3. In the next window, in File Name, enter `SubscribeController.m`, ensure **Also create SubscribeController.h** is selected, and click **Finish**.

   The new source files contain stub implementations of various methods.

Configuring the SUP101Appdelegate Files

Goal: The `SUP101Appdelegate.h` and `SUP101Appdelegate.m` files are created when you create the Xcode project, but you must add the view controller property and create the view controller instance.

The delegate file extends the functionality of reusable objects. A delegate allows one object to send messages to another object specified as its delegate to ask for input, or to be notified when an event occurs.

1. Add the view controller property to the application delegate:
   a) Open the `SUPAppdelegate.h` file and add this code:

```
@interface SUP101AppDelegate : NSObject <UIApplicationDelegate>
{
    UIWindow *window;
    UINavigationController *navController;
}

@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) IBOutlet UINavigationController
*navController;
- (void)showNoTransportAlert:(NSInteger) ret;
@end
```

   b) Save the `SUP101Appdelegate.h` file.
2. Create an instance of the view controller and set it as the value for the property, import the view controller's header file, synthesize the accessor methods, and make sure the view controller is released in the **dealloc** method:
   a) In the SUP101 Xcode project, open the `SUP101Appdelegate.m` file and enter this code:

```
#import "SUP101AppDelegate.h"
#import "SUPMessageClient.h"
#import "SUP101_SUP101DB.h"
#import "SUP101CallbackHandler.h"

@implementation SUP101AppDelegate

@synthesize window;
@synthesize navController;

- (void)showNoTransportAlert:(NSInteger) ret
{
    NSString *message = nil;
    if (ret == kSUPMessageClientNoSettings) {
       message = [[NSString alloc] initWithString:@"No required
settings, use the Settings app to enter the provisioning
information."];
    } else if (ret == kSUPMessageClientKeyNotAvailable) {
        message = [[NSString alloc] initWithString:@"Unable to
access the key."];
    } else {
        message = [[NSString alloc] initWithString:@"Operation
fails."];
    }
    UIAlertView * noTransportAlert = [[[UIAlertView alloc]
initWithTitle:@"Messaging Client Fails to Start"
message:message delegate:self
cancelButtonTitle:NSLocalizedString(@"OK", @"Label for alert
button OK") otherButtonTitles:nil] autorelease];
    [noTransportAlert show];
}
- (void)applicationDidFinishLaunching:(UIApplication
*)application
{
    // Override point for customization after application
launch
    if([SUP101_SUP101DB databaseExists])
       [SUP101_SUP101DB deleteDatabase];
    [SUP101_SUP101DB createDatabase];

    [MBOLogger setLogLevel:LOG_INFO];
    SUP101CallbackHandler* databaseCH = [SUP101CallbackHandler
newInstance];
    [SUP101_SUP101DB registerCallbackHandler:databaseCH];
    [SUP101_SUP101DB createDatabase];
    [SUP101_SUP101DB startBackgroundSynchronization];
    [NSThread sleepForTimeInterval:2];

    NSInteger result = [SUPMessageClient start];

    if (result == kSUPMessageClientSuccess)
    {
        [SUP101_SUP101DB asyncOnlineLogin:@"supuser"
password:@"s3pUser"];
        while([databaseCH loginSuccessCount] < 1) {
            [NSThread sleepForTimeInterval:1];
        }
        [window addSubview:navController.view];
        [window makeKeyAndVisible];
    } else
    {
        [self showNoTransportAlert:result];
    }
}
- (void)applicationWillTerminate:(UIApplication *)application
{
    [SUP101_SUP101DB unsubscribe];
    [SUPMessageClient stop];

}

- (void)dealloc {
    [navController release];
    [window release];
    [super dealloc];
}
- (void)alertView:(UIAlertView *)actionSheet
clickedButtonAtIndex:(NSInteger)buttonIndex {
    //button index 0 is the cancel button
    if (buttonIndex == 0){
        exit (0);
    }
}
@end
```

b) Save the `SUP101Appdelegate.m` file.

Configuring the SubscribeController View

Goal: Use Interface Builder to configure the `SubscribeController.xib` file and create the user interface.

1. Double-click the `SubscribeController.xib` file to open Interface Builder.

The file contains three objects:

- File's Owner – the object that is set to be the owner of the user interface, which is typically the object that loads the interface. In this tutorial, this is the SubscribeController.
- First Responder – the first responder proxy object handles events. Connecting an action to the first responder means that when the action is invoked, it is dynamically sent to the responder chain.
- View – displayed in a separate window to allow you to edit it.

2. To make connections to and from the File's Owner, you must use the Identity Inspector to tell Interface Builder the class of the object:

a) In the SubscribeController.xib document window, select the **File's Owner** icon, then select **Tools** > **Identity Inspector**.

b) In the Class field, select **SubscribeController**.

c) Select the **View** icon, then select **Tools > Identity Inspector**.

d) In the Identity Inspector Class field, select **UIView**.

3. Add the user interface elements to the View. In this case, you will be adding a button.

a) In Interface Builder, select **Tools** > **Library**.

b) Scroll through the Library and select the Button icon, then drag and drop it onto the View window.

You can resize the button using the resize handles and position the button by dragging it to the desired location.

c) Double-click inside the button and type: `Subscribe`.

4. To make connections to the user interface from the view controller, you must specify outlets in the `SubscribeController.h` file. You must also add property declarations for the instance variables and a declaration for the action method:

a) Open the `SubscribeController.h` file and add this code:

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

@interface SubscribeController : UIViewController {
    CustomerListController *listController;
}
-(IBAction)buttonPressed:(id)sender;
@end
```

Note: This code references a view controller (CustomerListController) you will create later in this tutorial. This code says that when the user touches the Subscribe button, the CustomerList view is called.

b) Save the `SubscribeController.h` file.

5. In the `SubscribeController.m` file, add the implementation code:

a) Open the `SubscribeController.m` file and add:

```
#import "SubscribeController.h"
#import "SUP101_SUP101DB.h"
#import "SUP101CallbackHandler.h"
#import "SUP101AppDelegate.h"

@implementation SubscribeController

- (IBAction)buttonPressed:(id)sender
{

    [SUP101_SUP101DB subscribe];
    while ([(SUP101CallbackHandler *)[SUP101_SUP101DB
callbackHandler] importSuccessCount] < 1)
    {
        sleep(1);
    }
    if (listController == nil)
    {
        listController = [[CustomerListController alloc]
initWithStyle:UITableViewStylePlain];
    }
    SUP101AppDelegate *delegate = [[UIApplication
sharedApplication] delegate];
    [delegate.navController pushViewController:listController
animated:YES];
}

/*
 // The designated initializer.  Override if you create the
controller programmatically and want to perform customization
that is not appropriate for viewDidLoad.
 - (id)initWithNibName:(NSString *)nibNameOrNil bundle:
(NSBundle *)nibBundleOrNil {
 if (self = [super initWithNibName:nibNameOrNil
bundle:nibBundleOrNil]) {
 // Custom initialization
 }
 return self;
 }
 */
```

```
// Implement viewDidLoad to do additional setup after loading
the view, typically from a nib.
- (void)viewDidLoad {
    self.title = @"Subscribe";
    [super viewDidLoad];
}


/*
 // Override to allow orientations other than the default
portrait orientation.
 - (BOOL)shouldAutorotateToInterfaceOrientation:
(UIInterfaceOrientation)interfaceOrientation {
 // Return YES for supported orientations
 return (interfaceOrientation ==
UIInterfaceOrientationPortrait);
 }
 */

- (void)didReceiveMemoryWarning {
    // Releases the view if it doesn't have a superview.
    [super didReceiveMemoryWarning];

    // Release any cached data, images, etc that aren't in use.
}

- (void)viewDidUnload {
    // Release any retained subviews of the main view.
    // e.g. self.myOutlet = nil;
}


- (void)dealloc {
    if (listController)
    {
        [listController release];
        listController = nil;
    }
    [super dealloc];
}


@end
```

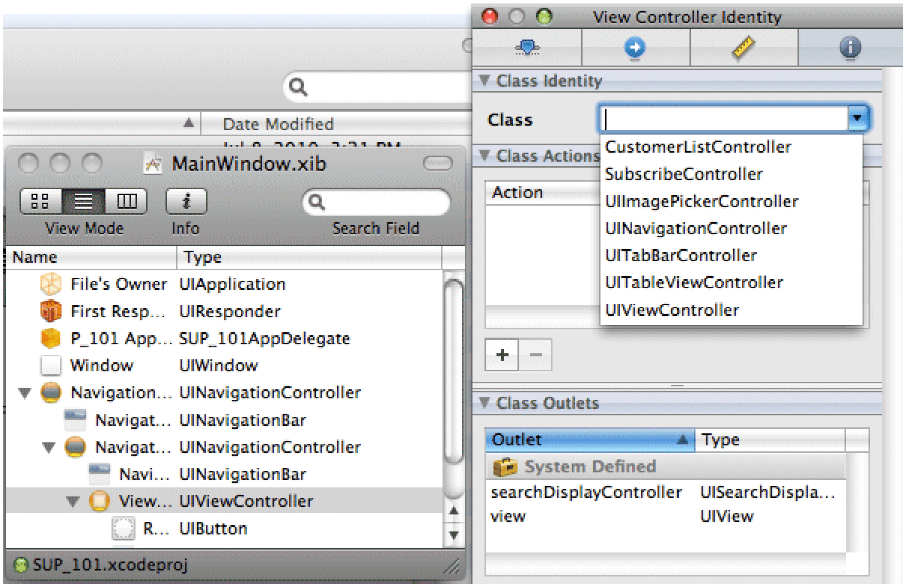
b) Save the `SubscribeController.m` file.

Making Connections

Goal: Add a Navigation Controller to the MainWindow.xib and create a connection from the AppDelegate to the Navigation Controller .

1. Double-click the the `MainWindow.xib` file.
2. In the Interface Builder **Tools** > **Library** menu, drag and drop the Navigation Controller onto the **MainWindow.xib** documents window.

3. Create a connection from the AppDelegate to the Navigation Controller.
   a) Control-click the **AppDelegate** icon to show available outlets and actions.
   b) In the MainWindow.xib, Control-drag from the **AppDelegate** icon to the new Navigation Controller icon and select the **navController** outlet.
   c) Select the Navigation Controller icon and change the **View Mode** to list view (click the middle icon).
   d) From the Navigation Controller list, select **UIViewController** and open the Identity Inspector in **Tools** > **Identity Inspector**.
   e) In the Identity Inspector, in Class, select **SubscribeController**.



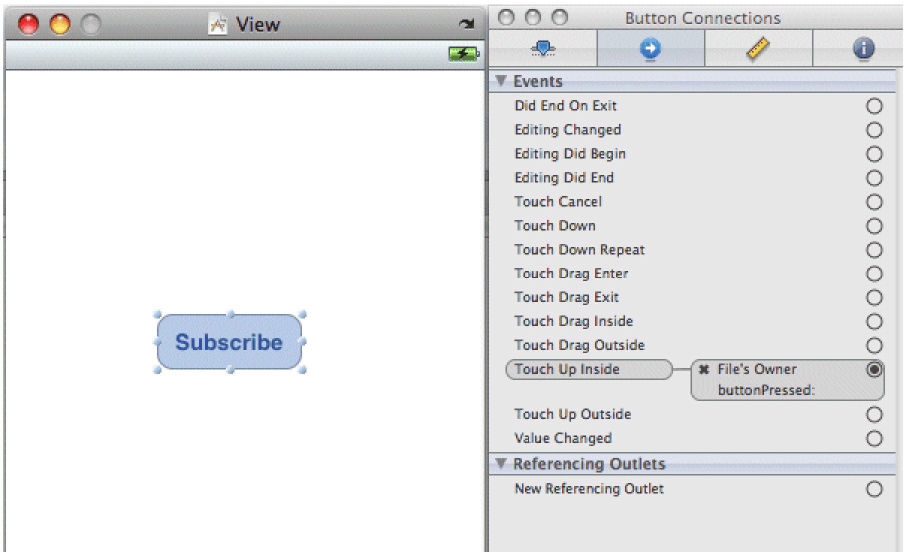
Name	Type
File's Owner	UIApplication
First Resp...	UIResponder
P_101 App...	SUP_101AppDelegate
Window	UIWindow
Navigation...	UINavigationController
Navigat...	UINavigationBar
Navigat...	UINavigationController
Navi...	UINavigationBar
View...	UIViewController
R...	UIButton

CustomerListController
SubscribeController
UIImagePickerController
UINavigationController
UITabBarController
UITableViewController
UIViewController

Outlet	Type
searchDisplayController	UISearchDispla...
view	UIView

   UIViewController changes to SubscribeController.
   f) Save the `MainWindow.xib` file.
4. Establish a connection between the Subscribe button you added to the `SubscribeController.xib` and the button's action method.
   a) Open the `SubscribeController.xib` file.
   b) In the View window, control-click the **Subscribe** button to show the inspector, then drag from the open circle in the **Touch Up Inside Events** list to the **File's Owner** icon and select **buttonPressed**.

   This shows the way the button is connected to the buttonPressed 76 event.



c) Save the `SubscribeController.xib` file.

Adding the CustomerListController

Goal: Create the customer list view.

1. In the SUP101 Xcode project, select **File** > **New File**.
2. In the new File window, select the **Cocoa Touch Class** group, the **UIViewController subclass** and the **UITableViewController subclass** option. Unselect **With XIB for user interface**, then click **Next**.
3. In the next window, in File Name, enter `CustomerListController.m`, ensure that **Also create CustomerListController.h** is selected, and click **Finish**.

   The new source files contain stub implementations of various methods.
4. Open the `CustomerListController.m` file, and add this code:

```
#import "CustomerListController.h"
#import "SUP101AppDelegate.h"
#import "DetailController.h"

#import "SUP101_Customer.h"


@implementation CustomerListController
@synthesize customerList;

- (id)initWithStyle:(UITableViewStyle)style {
    // Override initWithStyle: if you create the controller
programmatically and want to perform customization that is not
appropriate for viewDidLoad.
    if (self = [super initWithStyle:style]) {
    }
    return self;
}



- (void)viewDidLoad {
    // Uncomment the following line to display an Edit button in
the navigation bar for this view controller.
    // self.navigationItem.rightBarButtonItem =
self.editButtonItem;
    [super viewDidLoad];
}

- (void)viewWillAppear:(BOOL)animated {

    self.title = @"Customers";
    NSMutableArray *array = [[NSMutableArray alloc] init];

    SUP101_CustomerList *customers = [SUP101_Customer findAll];
    if ([customers length] > 0)
    {
        for (SUP101_Customer * oneRec in customers)
        {
            [array addObject:oneRec];
        }
    }
    self.customerList = array;
    [array release];
    [[self tableView] reloadData];
    [super viewWillAppear:animated];

}


/*
- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
}
*/
/*
- (void)viewDidAppear:(BOOL)animated {
    [super viewDidAppear:animated];
}
*/
/*
- (void)viewWillDisappear:(BOOL)animated {
    [super viewWillDisappear:animated];
}
*/
/*
- (void)viewDidDisappear:(BOOL)animated {
    [super viewDidDisappear:animated];
}
*/

/*
```

```
// Override to allow orientations other than the default portrait
orientation.
- (BOOL)shouldAutorotateToInterfaceOrientation:
(UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation ==
UIInterfaceOrientationPortrait);
}
*/

- (void)didReceiveMemoryWarning {
    // Releases the view if it doesn't have a superview.
    [super didReceiveMemoryWarning];

    // Release any cached data, images, etc that aren't in use.
}

- (void)viewDidUnload {
    // Release any retained subviews of the main view.
    // e.g. self.myOutlet = nil;
}


#pragma mark Table view methods

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 1;
}


// Customize the number of rows in the table view.
- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section {
    return [self.customerList count];
}


// Customize the appearance of table view cells.
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
initWithStyle:UITableViewCellStyleDefault
reuseIdentifier:CellIdentifier] autorelease];
    }

    // Set up the cell...
    NSUInteger row = [indexPath row];
    SUP101_Customer *customer = [customerList objectAtIndex:row];
    cell.textLabel.text = [NSString stringWithFormat:@"%@%@%@",
[customer fname], @" ", [customer lname]];
    cell.accessoryType =
UITableViewCellAccessoryDisclosureIndicator;
    return cell;
}


- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath {

    [self tableView:tableView
accessoryButtonTappedForRowWithIndexPath:indexPath];

}


/*
// Override to support conditional editing of the table view.
- (BOOL)tableView:(UITableView *)tableView canEditRowAtIndexPath:
(NSIndexPath *)indexPath {
    // Return NO if you do not want the specified item to be
editable.
    return YES;
}
*/


/*
// Override to support editing the table view.
- (void)tableView:(UITableView *)tableView commitEditingStyle:
(UITableViewCellEditingStyle)editingStyle forRowAtIndexPath:
(NSIndexPath *)indexPath {

    if (editingStyle == UITableViewCellEditingStyleDelete) {
        // Delete the row from the data source
        [tableView deleteRowsAtIndexPaths:[NSArray
arrayWithObject:indexPath] withRowAnimation:YES];
    }
    else if (editingStyle == UITableViewCellEditingStyleInsert) {
       // Create a new instance of the appropriate class, insert it
into the array, and add a new row to the table view
    }
}
*/


/*
// Override to support rearranging the table view.
- (void)tableView:(UITableView *)tableView moveRowAtIndexPath:
(NSIndexPath *)fromIndexPath toIndexPath:(NSIndexPath
*)toIndexPath {
}
*/


/*
```

```
// Override to support conditional rearranging of the table view.
- (BOOL)tableView:(UITableView *)tableView canMoveRowAtIndexPath:
(NSIndexPath *)indexPath {
    // Return NO if you do not want the item to be re-orderable.
    return YES;
}
*/
/*
- (UITableViewCellAccessoryType)tableView:(UITableView
*)tableView accessoryTypeForRowWithIndexPath:(NSIndexPath
*)indexPath
{
    return UITableViewCellAccessoryDisclosureIndicator;
}
*/
- (void)tableView:(UITableView *)tableView
accessoryButtonTappedForRowWithIndexPath:(NSIndexPath *)indexPath
{
    if (childController == nil)
        childController = [[DetailController alloc]
                            initWithNibName:@"DetailController"
bundle:nil];

    NSUInteger row = [indexPath row];

    SUP101_Customer *selectedCustomer = [customerList
objectAtIndex:row];
    childController.title = [NSString stringWithFormat:@"%d",
[selectedCustomer id]];
    childController.originalObj = selectedCustomer;

    SUP101AppDelegate *delegate = [[UIApplication
sharedApplication] delegate];
    [delegate.navController pushViewController:childController
animated:YES];

}


- (void)dealloc {
    [customerList release];
    [childController release];
    [super dealloc];
}


@end
```

5. Save the `CustomerListController.m` file.
6. Open the `CustomerListController.h` file, and add this code:

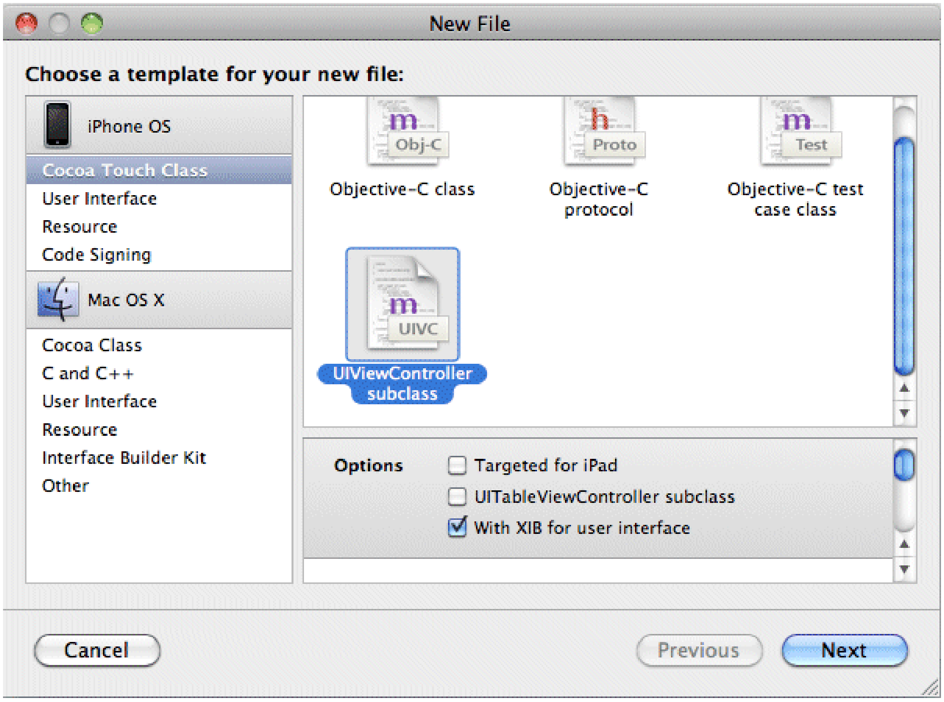
```
#import <UIKit/UIKit.h>
#import "DetailController.h"

@interface CustomerListController : UITableViewController
        <UITableViewDelegate, UITableViewDataSource> {
            NSArray *customerList;
            DetailController *childController;
}
@property (nonatomic, retain) NSArray *customerList;
@end
```

7. Save the `CustomerListController.h` file.

Adding the DetailController

Goal: Create the `DetailController.xib`.

1. In the SUP101 Xcode project, select **File** > **New File**.
2. Select **Cocoa Touch Class**, **UIViewController subclass**, and **With XIB for user interface**. Then click **Next**.



Options
- [ ] Targeted for iPad
- [ ] UITableViewController subclass
- [✓] With XIB for user interface

The "With XIB for user interface" option creates a NextStep Interface Builder (nib) file to go with the view controller and adds it to the SUP101 Xcode project.

3. In the next window, in File Name, enter `DetailController.m`, ensure that **Also create DetailController.h** is selected, and click **Finish**.

   The new source files contain stub implementations of various methods.

Configuring the DetailController View

Goal: Add the user interface to the customer detail view and specify the outlets in the `DetailController.m` and `DetailController.h` files.

1. Double-click the `DetailController.xib` file to open Interface Builder.
2. Add the user interface elements to the View. In this case, you will be adding three text fields with labels, and a button.
   a) In Interface Builder, select **Tools** > **Library**.
   b) Scroll through the Library and select the Text field icon (UITextField), then drag and drop it onto the View window. Repeat this step until you have three text fields on the View.

      You can resize the text fields using the resize handles and position the button by dragging it to the desired location.
   c) From the Library, drag and drop the Label (UILabel) onto the View window next to the text fields. Replace the text "Label" with:
      - First Name
      - Last Name
      - Phone
   d) From the Library, drag and drop the Button (UIButton) control onto the View window.
   e) Double-click inside the button and type: `Submit`.
3. To make connections to the user interface from the view controller, you must specify outlets in the `DetailController.h` file. You must also add property declarations for the instance variables and declaration for the action method:
   a) Open the `DetailController.h` file and add this code:

```
#import <UIKit/UIKit.h>
#import "SUP101_Customer.h"

@interface DetailController : UIViewController {
    IBOutlet UITextField *fname;
    IBOutlet UITextField *lname;
    IBOutlet UITextField *phone;
    SUP101_Customer * originalObj;
}
@property (nonatomic, retain) UITextField *fname;
@property (nonatomic, retain) UITextField *lname;
@property (nonatomic, retain) UITextField *phone;
@property (nonatomic, retain) SUP101_Customer *originalObj;
- (SUP101_Customer *)originalObj;
- (void)setOriginalObj: (SUP101_Customer *)newObj;
-(IBAction)buttonPressed:(id)sender;
@end
```

   b) Save the `DetailController.h` file.
4. In the `DetailController.m` file, add the implementation code:

a) Open the `DetailController.m` file and add:

```
#import "DetailController.h"
#import "SUP101_SUP101DB.h"
#import "SUP101AppDelegate.h"
@implementation DetailController
@synthesize fname;
@synthesize lname;
@synthesize phone;

- (SUP101_Customer *)originalObj
{
    return originalObj;
}
- (void)setOriginalObj: (SUP101_Customer *)newObj
{
    if (originalObj != newObj) {
        [originalObj release];
        originalObj = [newObj retain];
    }
}

 // The designated initializer.  Override if you create the
controller programmatically and want to perform customization
that is not appropriate for viewDidLoad.
- (id)initWithNibName:(NSString *)nibNameOrNil bundle:
(NSBundle *)nibBundleOrNil {
    if (self = [super initWithNibName:nibNameOrNil
bundle:nibBundleOrNil]) {
        // Custom initialization
    }
    return self;
}

- (IBAction)buttonPressed:(id)sender
{

    if (([lname.text compare:originalObj.fname] !=
NSOrderedSame) ||
        ([fname.text compare:originalObj.lname] !=
NSOrderedSame) ||
        ([phone.text compare:originalObj.phone] !=
NSOrderedSame))
    {
        SUP101_Customer *newCustomer = [SUP101_Customer find:
[originalObj id]];
        if (newCustomer) {
            newCustomer.lname = lname.text;
            newCustomer.fname = fname.text;
            newCustomer.phone = phone.text;

            [newCustomer save];
            [newCustomer submitPending];
            while ([SUP101_SUP101DB hasPendingOperations])
            {
                sleep(1);
            }
SUP101AppDelegate *delegate = [[UIApplication
sharedApplication] delegate];
[delegate.navController popViewControllerAnimated:YES];
        }

    }

}

/*
// Implement viewDidLoad to do additional setup after loading
the view, typically from a nib.
- (void)viewDidLoad {
    [super viewDidLoad];
}
*/


// Override to allow orientations other than the default
portrait orientation.
- (BOOL)shouldAutorotateToInterfaceOrientation:
(UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation ==
UIInterfaceOrientationPortrait);
}

- (void)viewWillAppear:(BOOL)animated {

    fname.text = originalObj.fname;
    lname.text = originalObj.lname;
    phone.text = originalObj.phone;
    [super viewWillAppear:animated];
}


- (void)didReceiveMemoryWarning {
    // Releases the view if it doesn't have a superview.
    [super didReceiveMemoryWarning];

    // Release any cached data, images, etc that aren't in use.
}

- (void)viewDidUnload {
    // Release any retained subviews of the main view.
    // e.g. self.myOutlet = nil;
}


- (void)dealloc {
    [fname release];
    [lname release];
    [phone release];
    [originalObj release];
    [super dealloc];

- (IBAction)touchedEnded:(NSSet*)touches withEvent:
(UIEvent*)event
{
   UITextView* fname1 = (UITextView*) [[self view] viewWithTag:
1];
   UITextView* lname1 = (UITextView*) [[self view] viewWithTag:
2];
   UITextView* phone1 = (UITextView*) [[self view] viewWithTag:
3];

   [fname1 resignFirstResponder];
   [lname1 resignFirstResponder];
   [phone1 resignFirstResponder];
}

@end
```

b) Save the `DetailController.m` file.

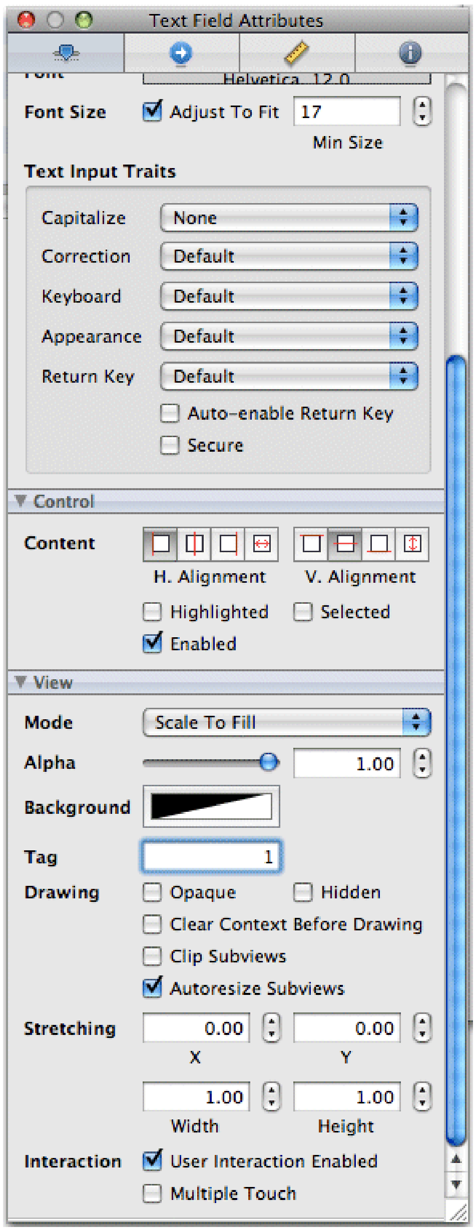
5. Change the tags in the `DetailController.xib` file:

a) Double-click the `DetailController.xib` file to open it in Interface Builder.

b) In the View window, select the First Name text field.

c) Select the **Tools** > **Attributes Inspector** to open the Atrributes Inspector for the text box.

d) In Attributes Inspector, scroll to the **View** section and in the **Tag** field, enter `1`.



Font: Helvetica, 12.0

Font Size: [✓] Adjust To Fit — 17 Min Size

Text Input Traits

- Capitalize: None
- Correction: Default
- Keyboard: Default
- Appearance: Default
- Return Key: Default
- [ ] Auto-enable Return Key
- [ ] Secure

Control

Content: H. Alignment, V. Alignment

- [ ] Highlighted
- [ ] Selected
- [✓] Enabled

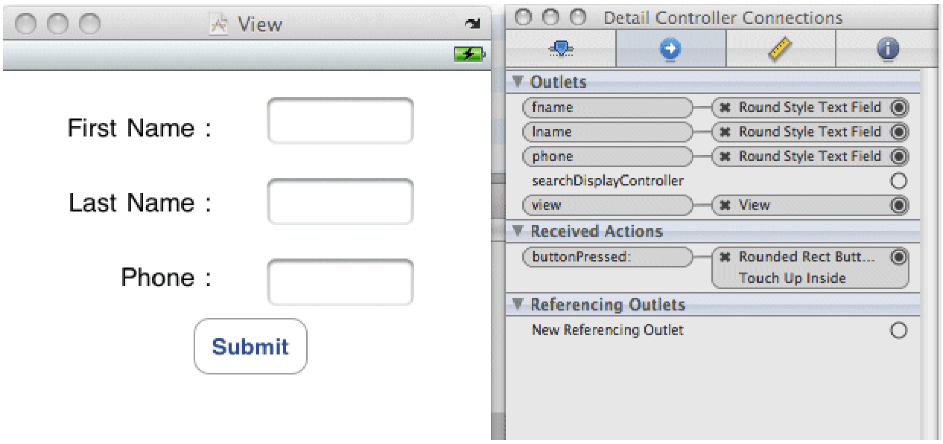
View

- Mode: Scale To Fill
- Alpha: 1.00
- Background
- Tag: 1
- Drawing: [ ] Opaque, [ ] Hidden, [ ] Clear Context Before Drawing, [ ] Clip Subviews, [✓] Autoresize Subviews
- Stretching: X 0.00, Y 0.00, Width 1.00, Height 1.00
- Interaction: [✓] User Interaction Enabled, [ ] Multiple Touch

e) Repeat the same steps for the Last Name and Phone text fields respectively, and in the **Tag** field in the Text Field Attributes, for the Last Name text field, enter 2 and for the Phone text field, enter 3.

6. Add the connections between the text fields and the outlets defined in the `DetailController.m` file (fname, lname, phone).

Inside the Main Window, Control-drag a connection from **File's Owner** to each of the text fields and select **fname**, **lname**, and **phone** outlets, respectively. This will ensure outlets are linked to the text fields. The end result looks like this:



Outlets
fname — Round Style Text Field
lname — Round Style Text Field
phone — Round Style Text Field
searchDisplayController
view — View

Received Actions
buttonPressed: — Rounded Rect Butt... Touch Up Inside

Referencing Outlets
New Referencing Outlet

7. **7.** Add the connection for the **Submit** button:
   a) Control-drag a connectin from **File's Owner** to **Submit**.
   b) Select **Touch up Inside** for the button.
8. **8.** Save the `DetailController.xib` file.

Deploying the Device Application

Goal: Deploy the SUP101 application to the iPhone Simulator for testing.

Prerequisites

Have a registered device user in Sybase Control Center.

You must be connected to the server where the mobile application project is deployed.

Task

1. **1.** In XCode, select **Build > Build and Run**.

   The project builds and the iPhone Simulator starts.
2. **2.** In the Settings screen of the iPhone simulator, choose **SUP101** and enter the connection settings:
   - ServerNameSetting – the machine that hosts the server where the SUP101 mobile application project is deployed.
   - ServerPortSetting – Unwired Server port number. The default is **5001**.

- CompanyIDSetting – the company ID you entered when you registered the device in Sybase Control Center, in this case, `0`.
- UserNameSetting – the user you registered in Sybase Control Center, `user1`.
- ActivationCodeSetting – the activation code for the user, `123`.

3. In the iPhone applications screen, open the **SUP101** application.
4. Click **Subscribe**.
   The customer list appears.
5. Select a customer record from the customer list and double-click to open the detail view.
   The customer detail shows the fields: First Name, Last Name, and Phone.
6. Change the First Name to something else, and click **Submit**.
7. You return to the customer list screen, where the changed record appears with an indicator.

Learn More about Sybase Unwired Platform

Once you have finished, try some of the other samples or tutorials, or refer to other development documents in the Sybase Unwired Platform documentation set.

Getting Started Tutorials

Try out some of the other getting started tutorials to get a broad view of the development tools available to you.

Advanced Tutorials

Tutorials are available that demonstrate how to use some of Sybase Unwired Platform advanced features.

Check the Sybase Web site regularly for updates. Navigate to *Support > Product Documentation > Sybase Unwired Platform*, then select the most current version of the document.

Samples

Sample applications are fully developed, working applications that demonstrate the features and capabilities of Sybase Unwired Platform.

Check the Sybase Web site regularly for updates. Navigate to the Sybase Web site, then select *Products > Sybase Unwired Platform > Use tab*: *http://www.sybase.com/products/mobileenterprise/sybaseunwiredplatform?htab=USE*.

Online Help

See the online help that is installed with the product, or the Product Documentation Web site.

Check the Sybase Web site regularly for updates. Navigate to *Support > Product Documentation > Sybase Unwired Platform*, then select the most current version of the document.

Developer References

See the Developer References to learn about using the API to custom code device applications using the API.

- *Developer Reference for BlackBerry*
- *Developer Reference for iOS*
- *Developer Reference for Mobile Workflow Packages*
- *Developer Reference for Windows and Windows Mobile*

Check the Sybase Web site regularly for updates. Navigate to *Support > Product Documentation > Sybase Unwired Platform*, then select the most current version of the document.

Javadocs are also available in the installation directory.

Programmer References

See the Programmer References to learn how to use the Administration API and Server API to extend functionality.

- *Reference: Administration APIs* – integrate your own administrative tools with Unwired Platform to monitor and manage Unwired Platform.
- *Reference: Custom Development for Unwired Server* – customize some Unwired Server features.

Check the Sybase Web site regularly for updates. Navigate to *Support > Product Documentation > Sybase Unwired Platform*, then select the most current version of the document.

Javadocs are also available in the installation directory.

Index