SYBASE®
An SAP® Company

Developer Guide: Windows and Windows Mobile Native Applications

Sybase Unwired Platform 2.1

DOCUMENT ID: DC01216-01-0210-02
LAST REVISED: November 2011

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

Contents

Introduction to Developer Guide: Windows and Windows Mobile Native Applications

This developer guide provides information about using advanced Sybase® Unwired Platform features to create applications for Microsoft Windows and Windows Mobile devices. 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: Windows Mobile Application Development*
- *Troubleshooting for Sybase Unwired Platform*
- *C# documentation, which provides a complete reference to the APIs:*
  - You can integrate help for generated code from mobile business objects (MBOs) into your Visual Studio project. See *Integrating Help into a Project* on page 13.

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

Documentation Roadmap for Unwired Platform

Learn more about Sybase® Unwired Platform documentation.

See *Documentation Roadmap* in *Fundamentals* for document descriptions by user role. *Fundamentals* is available on Production Documentation.

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

Introduction to Developing Device Applications with Sybase Unwired Platform

A device application includes both business logic (the data itself and associated metadata that defines data flow and availability), and device-resident presentation and logic.

Within Sybase Unwired Platform, development tools enable both aspects of mobile application development:

- The data aspects of the mobile application are called mobile business objects (MBO), and "MBO development" refers to defining object data models with back-end enterprise information system (EIS) connections, attributes, operations, and relationships that allow segmented data sets to be synchronized to the device. Applications can reference one or more MBOs and can include synchronization keys, load parameters, personalization, and error handling.
- Once you have developed MBOs and deployed them to Unwired Server, develop device-resident presentation and logic for your device application by generating code to use as a base in a native IDE. Follow an API approach that uses your native IDE's Client Object API. Unwired WorkSpace provides MBO code generation options targeted for specific development environments, for example, BlackBerry JDE for BlackBerry device applications, or Visual Studio for Windows Mobile device applications.

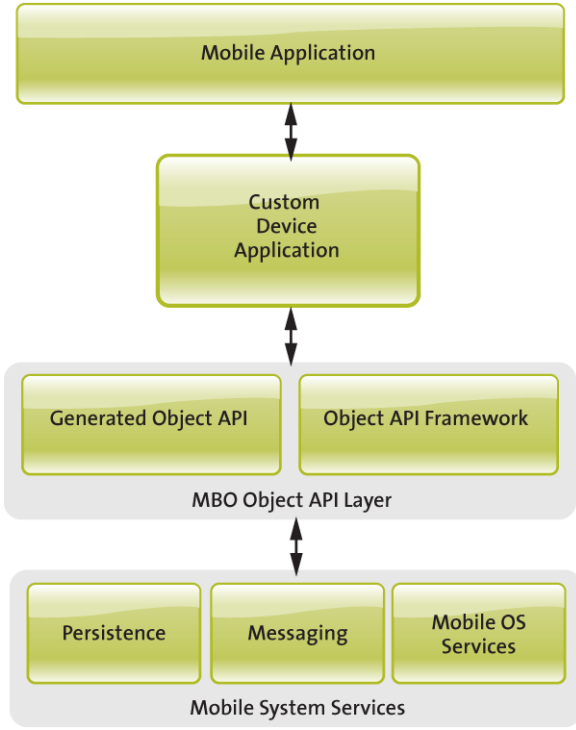
  The Client Object API uses the data persistence library to access and store object data in the database on the device. Code generation takes place in Unwired WorkSpace. You can generate code manually, or by using scripts. The code generation engine applies the correct templates based on options and the MBO model, and outputs client objects.

  Note: See *Sybase Unwired WorkSpace – Mobile Business Object Development* for procedures and information about creating and deploying MBOs.

Development Task Flow

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

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



Task Flow for C# Development

This describes a typical task flow for creating a device application using Visual Studio and C#.

Highlevel steps:

1. *Configuring Your Windows or Windows Mobile Environment* on page 4.
2. *Using Object API to Develop a Device Application* on page 8

3. *Developing a Windows or Windows Mobile Device Application Using Visual Studio* on page 13.

Configuring Your Windows or Windows Mobile Environment

This section describes how to set up your Visual Studio development environment, and provides the location of required DLL files and client object APIs.

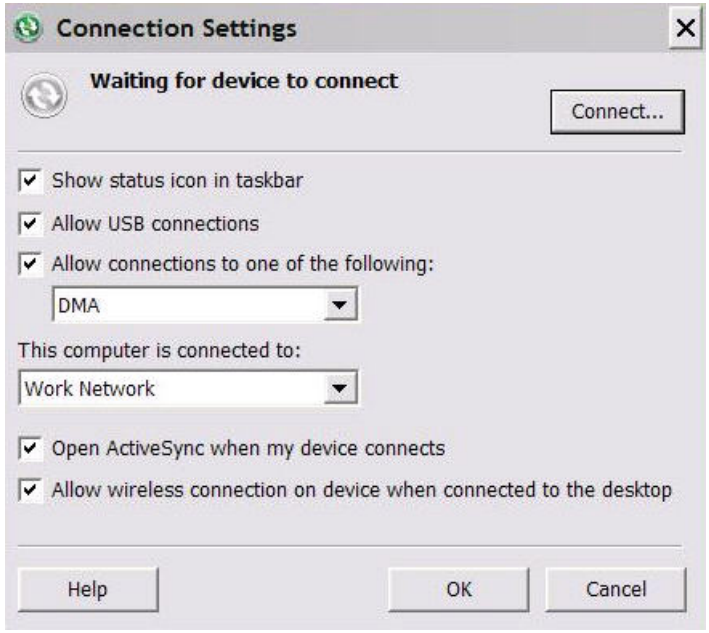
Installing the Windows Mobile Development Environment

Install and configure Microsoft ActiveSync so you can deploy and run device applications on an emulator. If you install Visual Studio 2008, the Windows Mobile Device Emulators (Windows Mobile 5) and Device Emulator Manager are already installed.

Note: Microsoft ActiveSync is for Windows XP. If you are using Windows Mobile 5.0, you must install Virtual PC 2007 SP1 to connect. If you are using Windows Mobile 6.0 or later using Active Sync to connect on Windows XP, use Windows Mobile Device Center to connect on Windows Vista and late Windows OS. You can download the Windows Mobile Device Center from *http://www.microsoft.com/windowsmobile/en-us/downloads/microsoft/device-center-download.mspx*.

1. Install the Windows Mobile 6 Professional SDK. You can download it from *http://www.microsoft.com/downloads/details.aspx?familyid=06111A3A-A651-4745-88EF-3D48091A390B&displaylang=en#AdditionalInfo*.
2. Download Microsoft ActiveSync from the *http://www.microsoft.com/windowsmobile/en-us/help/synchronize/device-synch.mspx*. Save it to your local machine. Windows XP requires version 4.5.
3. In Windows Explorer, double-click **setup.msi** to run the ActiveSync installer.
4. Follow the steps in the ActiveSync installer to complete the installation.
5. When installation is complete, restart your machine.

   ActiveSync starts automatically, and its icon appears in the Windows toolbar.
6. Double-click the **ActiveSync** icon.
7. Select **File > Connection Settings**.
8. In the Connection Settings dialog, select all the check boxes.
9. Under "Allow connections to one of the following", select **DMA**.
10. Under "This computer is connected to", select **Work Network**.



11. Click **OK**.

Configuring Windows Mobile Device Center

Before using the Windows Mobile Device Emulator, you need to change the settings of Windows Mobile Device Center.

1. Open Windows Mobile Device Center.
2. Click **Mobile Device Settings**.
3. Click **Connection Settings**.
4. Click on the **Allow connections to one of the following** checkbox.
5. Select **DMA** in the combobox.
6. On the **This computer is connected to** combobox, select **The Internet** if you want to allow the Windows Mobile device to access the Internet using Pocket IE.
7. Start the Windows Mobile Device Emulator.

Enabling Network Access from the Windows Mobile Device Emulator

Enable network access in the Windows Mobile Device Emulator for Windows Mobile 5.0. For Windows Mobile 6.0 and later, you are required to perform only step 5.

You can start the Windows Mobile Device Emulator from Visual Studio or from the Device Emulator Manager.

1. To start the Emulator from Visual Studio 2008:

a) Select **Tools** > **Device Emulator Manager**.

2. If a Device Emulator is not yet connected:
   a) Select a Device Emulator from the list and select **Connect**.
3. If you are using this Device Emulator for the first time:
   a) In the Emulator, select **File** > **Configure**.
   b) Click the **Network** tab.
   c) Check the **Enable NE2000 PCMCIA network adapter and bind to** checkbox.
   d) Select **Connected network card** from the list.
4. On the Emulator, configure the connection settings:
   a) In the Emulator, select **Start** > **Settings**.
   b) Select the **Connections** tab.
   c) Click **Connections**.
   d) Select the **Advanced** tab.
   e) Click on **Select Networks**.
   f) In the Settings window, select **My Work Network** in the first combobox.
   g) Select **File** > **Save State** and **Exit**.
   h) Restart the Emulator.
5. Right-click the current Emulator in Device Emulator Manager and select **Cradle**.

   ActiveSync starts. Once the connection is established, you should be able to access your PC and the Web from the Device Emulator.

Client Application Dependencies

To build device clients, some files, which are provided in the Unwired Platform installation, are required in certain situations, such as when using a secure port for synchronization.

The client API assembly DLL dependencies are installed under the `<UnwiredPlatform_InstallDir>\ClientAPI` directory. The contents of the `Client API` directory are:

- **`RBS\WM and RBS\Win32`** – Binaries of the framework classes for .NET.
  - WM: files for use on Windows CE based systems such as Windows Mobile 5+.
  - Win32: files for use on full Windows based systems like Windows XP.
- **`RBS\WM\Ultralite and RBS\Win32\cs\Ultralite`** – .NET Data Persistence Library and client database (UltraLite®) assemblies. This is used for replication-based synchronization client applications on Windows Mobile or Windows.
- **`ServerSync`** – Used in replication-based synchronization applications for push notification synchronization support.

The .NET assemblies listed above support Compact Framework 3.5+ on Visual Studio 2008. These project types are supported:

- Full .NET Framework 3.5+ Application
- Windows CE .NET CF 3.5+ Application
- Pocket PC .NET CF 3.5+ Application
- Smartphone .NET CF 3.5+ Application

If required, copy the following `.dll` files to the location used for referencing them in the Visual Studio application source project.

Platform	Location	Files	Notes
Windows Mobile Professional 6.0, 6.1, and 6.5	• `%Sybase%\Unwired Platform\ClientAPI\RBS\WM`	• `sup-client.dll` • `PUtil-TRU.dll`	
	• `%Sybase%\Unwired Platform\ClientAPI\RBS\WM\Ultralite`	• `ulnet11.dll` • `mlcrsa11.dll`	The `mlcrsa11.dll` file is needed only if you are using a secure port (HTTPS) for synchronization.
	`%Sybase%\UnwiredPlatform\ClientAPI\RBS\WM\Ultralite`	`iAnywhere.Data.UltraLite.dll`	
	`%Sybase%\UnwiredPlatform\ClientAPI\RBS\WM\Ultralite\<language>`	`iAnywhere.Data.UltraLite.resources.dll`	Copy from the respective locale-specific folders.
Windows XP, Vista, Windows 7	`%Sybase%\UnwiredPlatform\ClientAPI\RBS\Win32\cs`	• `sup-client.dll` • `pidutil.dll` • `PUtil-TRU.dll`	

Platform	Location	Files	Notes
	%Sybase%\UnwiredPlatform\ClientAPI\RBS\Win32\cs\Ultralite	• ulnet11.dll • mlcrsa11.dll	The mlcrsa11.dll file is required only if you are using a secure port (HTTPS) for synchronization.
	%Sybase%\UnwiredPlatform\ClientAPI\RBS\Win32\cs\Ultralite	iAnywhere.Data.UltraLite.dll	
	%Sybase%\UnwiredPlatform\ClientAPI\RBS\Win32\cs\Ultralite\<language>	iAnywhere.Data.UltraLite.resources.dll	Copy from the respective locale-specific folders.

Using Object API to Develop a Device Application

Generate object API code on which to build your application.

Unwired Platform provides the Code Generation wizard for generating object API code. Code generation creates the business logic, attributes, and operations for your Mobile Business Object. You can generate code for these platforms:

- Windows Mobile
- Windows

See the guidelines for generating code for each platform type.

Generating C# Object API Code

Generate object API code for applications that will run on Windows Mobile.

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

From	Action
The Mobile Application Diagram	Right-click within the Mobile Application Diagram and select **Generate Code**.

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

2. (Optional) Enter the information for these options:

Note: This page of the code generation wizard is seen only if you are using the Advanced developer profile.

Option	Description

Select code generation configuration	Select either an existing configuration that contains code generation settings, or generate device client code without using a configuration:
	• Continue without a configuration – select this option to generate device code without using a configuration.
	• Select an existing configuration – select this option to either select an existing configuration from which you generate device client code, or create a new configuration. Selecting this option enables:
	  • Select code generation configuration – lists any existing configurations, from which you can select and use for this session. You can also delete any and all existing saved configurations.
	  • Create new configuration – enter the **Name** of the new configuration and click **Create** to save the configuration for future sessions.
	  Select an existing configuration as a starting point for this session and click **Clone** to modify the configuration.

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

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

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

5. Click **Next**.
6. Enter the information for these configuration options:

Option	Description
Language	Select **C#**.

Option	Description
Platform	Select the platform (target device) from the drop-down list for which the device client code is intended. • NET Framework for Windows • NET Compact Framework 3.5 for Windows Mobile
Unwired Server	Specify a default Unwired Server connection profile to which the generated code connects at runtime.
Server domain	Choose the domain to which the generated code will connect. If you specified an Unwired Server to which you previously connected successfully, the first domain in the list is chosen by default. You can enter a different domain manually. **Note:** This field is only enabled when an Unwired Server is selected.
Page size	Optionally, select the page size for the generated client code. If the page size is not set, the default page size is 16KB at runtime. The default is a proposed page size based on the selected MBO's attributes. The page size should be larger than the sum of all attribute lengths (a binary length greater than 32767 is converted to a Binary Large Object (BLOB), and is not included in the sum; a string greater than 8191 is converted to a Character Large Object (CLOB), and is also not included) for any MBO that is included with all the selected MBOs. If an MBO attribute's length sum is greater than the page size, some attributes are automatically converted to BLOB or CLOB, and therefore, these attributes cannot be put into a `where` clause. **Note:** This field is only enabled when an Unwired Server is selected. The page size option is not enabled for message-based applications.

Option	Description
Namespace	Enter a namespace for C#. **Note:** The namespace name should follow naming conventions for C#. Do not use ".com" in the namespace.
Destination	Specify the destination of the generated device client files. Enter (or **Browse**) to either a **Project path** (Mobile Application project) location or **File system path** location. Select **Clean up destination before code generation** to clean up the destination folder before generating the device client files.
Replication-based	Select to use replication-based synchronization.
Backward compatible	Select so the generated code is compatible with the SUP 1.2 release.

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

   The object manager allows you to invoke MBOs using metadata instead of the object instances.
9. Click **Finish**.

Generated Code Location and Contents

Generated object API code is stored in the project's Generated Code sub-folder by default, for example, `C:\Documents and Settings\administrator\workspace \<Unwired Platform project name>\Generated Code\src`. Language, platform, and whether or not you select the Generate metadata classes option determines the class files generated in this folder.

Assuming you generate code in the default location, you can access it from WorkSpace Navigator by expanding the Mobile Application project folder for which the code is generated, and expand the Generated Code folder.

The contents of the folder is determined by the options you selected from the Generate Code wizard, and include generated class (.cs) files that contain:

- MBO - the business logic of your MBO.
- Synchronization parameters - any synchronization parameters for the MBOs.

- Personalization - personalization and personalization synchronization parameters used by the MBOs.
- Metadata - if you selected **Generate metadata classes**, the metadata classes which allow you to use code completion and compile-time checking to ensure that run-time references to the metadata are correct.

Validating Generated Code

Validation rules are enforced when generating client code for C# and Java. Define prefix names in the Mobile Business Object Preferences page to correct validation errors.

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

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

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

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

Creating a Project

Build a device application that runs on Windows or Windows Mobile platforms.

1. From the Visual Studio main menu select **File > New > Project**.
2. Select:
   - Target platform:
     - Windows Mobile 5.0 Professional
     - Windows Mobile 6.0, 6.1, and 6.5 Professional
     - Windows
   - Library version – .NET version 3.5
   - Language – the language used in the resource DLLs, to be included in the generated project.

Different sets of DLLs are included in the project based on your selections. The project contains all assemblies and runtime support DLLs required to access the Object API.

3. Click **OK** to generate the Visual Studio Project with the dependent Sybase Unwired Platform .NET assemblies.
4. Build the Solution. From the Visual Studio main menu select **Build > Build Solution**.

   The DLLs are copied to the target directory and the directory structure is flattened.

   Once generated and built, you can write custom mobile applications based on your selections.
5. Develop the mobile business objects (MBOs) that implement the business logic.

   See these online help topics:
   - *Sybase Unwired Platform > Sybase Unwired Workspace – Eclipse Edition > Develop > Developing a Mobile Business Object*
6. Use the Code Generation wizard to generate the C# Object API code for the mobile business object.
7. Add the generated code to the new project you created from the template.

   For more information, see the *Rebuilding the Generated Solution in Visual Studio*.

Developing a Windows or Windows Mobile Device Application Using Visual Studio

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

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

Windows Mobile Development

Develop a Windows Mobile application by generating the Visual Studio 2008 projects in C#, and running the application in the device or on a simulator to test.

1. Generate Mobile Business Objects (MBOs), then create a new Visual Studio project, then import generated MBOs, and create the user interface.
2. Add business logic to the generated code through the Windows Mobile Client Object API. See *Developer Guide for Windows and Windows Mobile > Reference > Client Object API*.
3. Run the application in the device or on a simulator.

Integrating Help into a Project

When you generate MBOs or client applications for Windows Mobile from Unwired WorkSpace, an XML file is generated for the MBOs. The generated Visual Studio project for

the forms can also generate a XML file. When you compile a project, an XML file is generated. You can use these XML files to generate online help.

To generate online help for Visual Studio 2008, you can use Sandcastle and Sandcastle Help File Builder. You can download and install Sandcastle and Sandcastle Help File Builder from these locations:

- *http://sandcastle.codeplex.com/Wikipage*
- *http://shfb.codeplex.com/releases*

To integrate help into your project build:

1. Add the /doc option in your project build, so that it can generate an XML file from the comments. You can also configure this option in the Visual Studio project properties. On the Build tab, select **XML documentation** and provide a file name.
2. Create a SandCastle Help File Builder project (.shfb file). Specify the assemblies and the XML file generated from the comments as input. You can also specify other help properties.
3. Use the `.shfb` project file in a script to build the document. For example:

```
<Target Name="Documentation">
   <Exec Command="$(SandCastleHelpBuilderPath) <shfb project
file>.shfb" />
</Target>
```

Debugging Windows and Windows Mobile Device Development

Device client and Unwired Server troubleshooting tools for diagnosing Microsoft Windows and Windows Mobile development problems.

Client-Side Debugging

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

- Unwired Server connection failed.
- Data does not appear on the client device.
- Physical device problems, such as low battery or low memory.

To find out more information on the device client side:

- If you have implemented debugging in your generated or custom code (which Sybase recommends), turn on debugging and review the debugging information. See *Developer Guide for Windows and Windows Mobile* about using the `MBOLogger` class to add log levels to messages reported to the console.
- Check the log record on the device. Use the `DatabaseClass.GetLogRecord(Sybase.Persistence.Query)` or `Entity.GetLogRecords` methods.

  This is the log format

  ```
  level,code,eisCode,message,component,entityKey,operation,requestI
  d,timestamp
  ```

This log format generates output similar to:

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

- `level` – the log level currently set. Values include: 1 = TRACE, 2 = DEBUG, 3 = INFO, 4 = WARN, 5 = ERROR, 6 = FATAL, 7 = OFF.
- `code` – Unwired Server administration codes.
  - Replication-based synchronization codes:
    - 200 – success.
    - 500 – failure.
- `eisCode` – maps to HTTP error codes. If no mapping exists, defaults to error code 500 (an unexpected server failure).
- `message` – the message content.
- `component` – MBO name.
- `entityKey` – MBO surrogate key, used to identify and track MBO instances and data.
- `operation` – operation name.
- `requestId` – operation replay request ID or messaging-based synchronization message request ID.
- `timestamp` – message logged time, or operation execution time.

- If you have implemented `onConnectionStatusChange` for message-based synchronization in `CallbackHandler`, the connection status between Unwired Server and the device is reported on the device. See the *Developer Guide for Windows and Windows Mobile* for `CallbackHandler` information. The device connection status, device connection type, and connection error message are reported on the device:
  - 1 – current device connection status.
  - 2 – current device connection type.
  - 3 – connection error message.

Server-Side Debugging

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

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

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

To find out more information on the Unwired Server side:

- Check the Unwired Server log files.
- For message-based synchronization mode, you can set the log level to DEBUG to obtain detailed information in the log files:
  1. Check the global SUP MSG log level in `<serrver_install_folder>\UnwiredPlatform\Servers\UnwiredServer\Repository\logging-configuration.xml` to ensure the `Log level` of `<Entity EntityTypeId="MSG">` is set to DEBUG.
  2. Modify the log level for the module `SUPBridge` and `JmsBridge` in `<server_install_folder>\UnwiredPlatform\Servers\MessagingServer\Data\TraceConfig.xml` to DEBUG.
  3. Check the `SUPBridge` and `JMSBridge` logs, for detailed information.

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

  Note: It is important to return to INFO mode as soon as possible, since DEBUG mode can effect system performance.

Creating a Mobile Application Project

This describes how to set up a project in Visual Studio. You must add the required libraries as references in the Visual Studio project.

You can use this method to create replication-based synchronization client projects.

1. Add the following libraries for the appropriate device platform as references in the Visual Studio project:

   For Windows Mobile:

- `sup-client.dll` – from `<UnwiredPlatform_InstallDir>\ClientAPI\RBS\WM`.
- `iAnywhere.Data.UltraLite.dll` – from `<UnwiredPlatform_InstallDir>\ClientAPI\RBS\WM\Ultralite`.
- `iAnywhere.Data.UltraLite.resources.dll` (several languages are supported) – from `<UnwiredPlatform_InstallDir>\ClientAPI\RBS\WM\Ultralite\<language>`.

For Windows:

- `sup-client.dll` – from `<UnwiredPlatform_InstallDir>\ClientAPI\RBS\Win32\cs`.
- `iAnywhere.Data.UltraLite.dll` – from `<UnwiredPlatform_InstallDir>\ClientAPI\RBS\Win32\cs\Ultralite`.
- `iAnywhere.Data.UltraLite.resources.dll` (several languages are supported) – from `<UnwiredPlatform_InstallDir>\ClientAPI\RBS\Win32\cs\Ultralite\<language>`.

2. Add the following libraries for the appropriate device platform as items in the Visual Studio project. Set "Build Action" to "Content" and "Copy to Output Directory" to **Copy always**.

   For Windows Mobile:

   - `ulnet11.dll` – from `<UnwiredPlatform_InstallDir>\ClientAPI\RBS\WM\Ultralite`.
   - `mlcrsa11.dll` (if HTTPS protocol is used) – from `<UnwiredPlatform_InstallDir>\ClientAPI\RBS\WM\Ultralite`.
   - `PUtilTRU.dll` - from `<UnwiredPlatform_InstallDir>\ClientAPI\RBS\WM`.

   For Windows:

   - `ulnet11.dll` – from `<UnwiredPlatform_InstallDir>\ClientAPI\RBS\Win32\cs\Ultralite`.
   - `mlcrsa11.dll` (if HTTPS protocol is used) – from `<UnwiredPlatform_InstallDir>\ClientAPI\RBS\Win32\cs\Ultralite`.
   - `mlczlib11.dll` (if using compression) - from `<UnwiredPlatform_InstallDir>\ClientAPI\RBS\Win32\cs\Ultralite`.

Configuring an Application to Synchronize and Retrieve MBO Data

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

1. Configure a synchronization profile to point to your host and port.

   ```
   TestDB.GetSynchronizationProfile().ServerName = "localhost";
   TestDB.GetSynchronizationProfile().PortNumber = 2480;
   ```

2. Log in to Unwired Server using a user name and password. This step is required for application initialization.

   ```
   TestDB.LoginToSync("supAdmin", "s3pAdmin");
   ```

3. Subscribe to Unwired Server. Unwired Server creates a subscription for this particular application.

   ```
   TestDB.Subscribe();
   ```

4. Synchronize with Unwired Server. Synchronization uploads all the local changes and downloads new data with related subscriptions.

   ```
   GenericList<ISynchronizationGroup> sgs = new
   GenericList<ISynchronizationGroup>();
   sgs.Add(TestDB.GetSynchronizationGroup("default"));
   TestDB.BeginSynchronize(sgs, "mycontext");
   ```

5. List all customer MBO instances from the local database using an object query. `FindAll` is a pre-defined object query.

   ```
   List<Customer> customers = Customer.FindAll();
   foreach (Customer customer in customers)
   {
     Console.WriteLine("customer: " + customer.Fname + " " +
   customer.Lname + " " + customer.Id  + customer.City);
   }
   ```

6. Find and update a particular MBO instance, and save to the local database.

   ```
   Customer cust = Customer.FindByPrimaryKey(441);
   cust.Address = "1 Sybase Dr.";
   cust.Phone = "9252360000";
   cust.Save();
   ```

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

   ```
   cust.SubmitPending();
   ```

8. Upload the pending changes to the Unwired Server and get the replay results and all the changed MBO instances.

   ```
   TestDB.BeginSynchronize(sgs, "mycontext");
   ```

9. Unsubscribe the device application if the application is no longer used.

   ```
   TestDB.Unsubscribe();
   ```

Localizing a Windows Mobile Application

Localize a Windows Mobile application by generating resource files, adding a resource file template and strings, and localizing the application code.

Reference

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

Windows Mobile Client Object API

The Sybase Unwired Platform Windows Mobile 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 Windows Mobile Client Object API is used by device applications to retrieve data and invoke mobile business object operations.

Connection APIs

The 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.

```
ConnectionProfile cp = SampleAppDB.GetConnectionProfile();
cp.SetEncryptionKey("Your key");
cp.Save();
```

Managing Device Database Connections

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

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

The `CloseConnection()` method closes the current database connection, and releases it from the used connection pool.

Improving Device Application Performance with Multiple Database Reader Threads
The `maxDbConnections` property improves device application performance by allowing multiple threads to read data concurrently from the same local database.

Note: Message based synchronization clients do not support a single write thread concurrently with multiple read threads. That is, when one thread is writing to the database, no read threads are allowed access at the same time. However, replication based synchronization clients do support a single write thread concurrently with multiple read threads. Both replication and message-based clients support multiple concurrent read threads.

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

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

You can specify the amount of reader threads using `maxDbConnections`. The default value is 2.

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

```
ConnectionProfile connectionProfile =
MyPackageDB.GetConnectionProfile();
```

To allow 6 concurrent read threads, set the maxDbConnections property to 6 in ConnectionProfile before accessing the package database at the beginning of the application.

```
    connectionProfile.MaxDbConnections = 6;
```

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.

```
ConnectionProfile cp = SampleAppDB.GetSynchronizationProfile();
cp.DomainName = "default";
```

You can set certificate information in SynchronizationProfile.

```
ConnectionProfile profile = MyDatabase.GetSynchronizationProfile();
profile.DomainName = "default";
profile.ServerName = "host-name";
profile.PortNumber = 2481;
profile.NetworkProtocol = "https";
profile.NetworkStreamParams =
"trusted_certificates=rsa_public_cert.crt";
```

You can allow clients to compress traffic as they communicate with the Unwired Server by including "compression=zlib" into the sync parameters:

```
MyDatabase.DB.GetSynchronizationProfile().NetworkStreamParams =
"compression=zlib;zlib_upload_window_size=12;zlib_download_window_s
ize=12";
```

Connect through a Relay Server

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

Edit <package-name>DB by modifying the values of the Relay Server properties for your Relay Server environment.

To update properties for Relay Server installed on Apache on Linux:

```
getSynchronizationProfile().setServerName("examplexp-vm1");
GetSynchronizationProfile().setPortNumber(2480);
GetSynchronizationProfile().setNetworkProtocol("http");
GetSynchronizationProfile().setNetworkStreamParams("trusted
certificates=;url_suffix=/cli/iarelayserver/<FarmName>");
GetSynchronizationProfile().setDomainName("default");
```

To update properties for Relay Server installed on Internet Information Services (IIS) on Microsoft Windows:

```
getSynchronizationProfile().setServerName("examplexp-vm1");
GetSynchronizationProfile().setPortNumber(2480);
GetSynchronizationProfile().setNetworkProtocol("http");
GetSynchronizationProfile().setNetworkStreamParams("trusted
certificates=;url_suffix=ias_relay_server/client/rs_client.dll/
<FarmName>");
GetSynchronizationProfile().setDomainName("default");
```

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

Authentication

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

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 last successfully authenticated credentials. There is no communication with Unwired Server in this method.

`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 down a value that the client can use when creating new records (by using the method `KeyGenerator.generateId()` to create key values that the server will accept).

The KeyGenerator value increments each time the `GenerateId` method is called. A periodic call to `SubmitPending` by the KeyGenerator 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 in a try/catch block in the client application and ensure that the client application does not attempt to send more messages to the Unwired Server if `LoginToSync` throws an exception.

```
void LoginToSync(string user, string password);
```

Connect Using a Certificate

You can set certificate information in `ConnectionProfile`.

```
ConnectionProfile profile = MyDatabase.GetSynchronizationProfile();
profile.DomainName = "default";
profile.ServerName = "host-name";
profile.PortNumber = 2481;
profile.NetworkProtocol = "https";
profile.NetworkStreamParams =
"trusted_certificates=rsa_public_cert.crt";
```

Enable End-to-End Encryption (E2EE) Using SSL

Replication synchronization streams can be encrypted end-to-end from the client to Unwired Server. You can configure the application to make these secure, encrypted connections.

This code example executes only the first time the application runs (`DataVault.exists` returns false) as part of application initialization.

The first two statements set up the UltraLite database to be encrypted (with the password "secret"), and the 4th and 5th statements set up synchronization to use E2EE. Statements 3 and 6 persist the settings. The last statement creates the encrypted database and uses E2EE for initial synchronization.

The `e2ee_public_key` is a file containing the server's PEM-encoded public key for end-to-end encryption.

The `e2ee_type` specifies the asymmetric algorithm to use for key exchange for end-to-end encryption. The value for `e2ee_type` must be either rsa or ecc, and must match the value specified on the server.

```
Sybase.Persistence.ConnectionProfile cp =
MyDB.GetConnectionProfile();
cp.SetEncryptionKey("secret");
cp.Save();
Sybase.Persistence.ConnectionProfile sp =
MyDB.GetSynchronizationProfile();
sp.NetworkStreamParams="tls_type=rsa;trusted_certificates=c:\\tmp\
\https_public_cert.crt;e2ee_type=rsa;e2ee_public_key=c:\
\e22_public.pem";
sp.Save();
MyDB.LoginToSync(...);
```

Encrypt the Database

You can use `ConnectionProfile.EncryptionKey` to set the encryption key of a local database. Set the key during application initialization, and before creating or accessing the client database.

```
ConnectionProfile profile = <PkgName>DB.GetConnectionProfile();
profile.SetEncryptionKey("Your key");
```

The encryption key must follow the rules applicable to DBKEY in UltraLite:

- Any leading or trailing spaces in parameter values are ignored.
- The value cannot include leading single quotes, leading double quotes, or semicolons.

Set Database File Property

You can use `setProperty` to specify the database file name on the device, such as the directory of the running program, a specific directory path, or a secure digital (SD) card.

```
ConnectionProfile cp = MyDatabaseClass.getConnectionProfile();
cp.setProperty("databaseFile", "databaseFile");
cp.save();
```

Examples

If you specify the *databaseFile* name only, with no path, the *databaseFile* is created in the path where the program is running:

```
/mydb.udb
```

The *databaseFile* is created in the `/Temp` directory of the Windows Mobile device:

```
/Temp/mydb.udb
```

The *databaseFile* is created on an SD card:

```
/Storage Card/mydb.udb
```

Note: For the database file path and name, the forward slash (/) is required as the path delimiter, for example `/smartcard/supprj.udb`.

Usage

- Be sure to call this API before the database is created:

  - Call this API before calling `LoginToSync()`.
- The database is UltraLite; use a database file name like `mydb.udb`.
- If the device client user changes the file name, the device user must make sure the input file name is a valid name and path on the client side.

Synchronization APIs

The client object API allows you to change synchronization parameters and perform mobile business object synchronization.

Changing Synchronization Parameters

Synchronization parameters determine the manner in which data is retrieved from the consolidated database during a synchronization session.

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

```
CustomerSynchronizationParameters sp =
Customer.SynchronizationParameters;
            sp.State = "CA";
            sp.Save();
```

Performing Mobile Business Object Synchronization

To perform mobile business object (MBO) synchronization, you must save a Connection object. Additionally, you may want to set synchronization parameters.

For replication-based synchronization, this code synchronizes an MBO package using a specified connection:

```
SampleAppDB.Synchronize (string synchronizationGroup)
```

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

Then you can call the **SubmitPendingOperations(string synchronizationGroup)** operation through the publication as this example illustrates:

```
            Product product_new = new Product();
            product_new.Color="Yellow";
            product_new.Description="";
            product_new.Id=888;
            product_new.Name = "ChildrenPants";
            product_new.Prod_size = "M";
            product_new.Quantity = 200;
            product_new.Unit_price = (decimal)188.00;
            product_new.Create();
```

```
            SampleAppDB.SubmitPendingOperations("default");
            while(SampleAppDB.HasPendingOperations())
            {
                System.Console.Write(" . ");
                System.Threading.Thread.Sleep(1000);
            }
```

You can use a publication mechanism, which allows as many as 32 simultaneous synchronizations. However, performing simultaneous synchronizations on several very large Unwired Server applications can impact server performance, and possibly affect other remote users. The following code samples demonstrate how to simultaneously synchronize multiple MBOs.

For message-based synchronization, synchronize multiple MBOs using:

```
SampleAppDB.SubmitPendingOperations();
```

Or you can use:

```
SampleAppDB.SubmitPendingOperations("my-pub");
```

where "my-pub" is the synchronization group defined.

For replication-based synchronization, synchronize multiple MBOs using:

```
SampleAppDB.Synchronize();
```

You can also use:

```
SampleAppDB.Synchronize("my-pub");
```

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 Mobile Business Objects

You can retrieve data from the local database through a variety of queries, including object queries, arbitrary find, and through filtering query result sets.

Object Queries

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

Object Query methods are generated based on the object queries defined by the modeler in Unwired WorkSpace. Object Query methods carry query 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.

This method retrieves all customers:

```
public static Sybase.Collections.GenericList<Customer> FindAll()

Sybase.Collections.GenericList<Customer> customers =
Customer.FindAll();
```

This method retrieves all customers in a certain page:

```
public static Sybase.Collections.GenericList<Customer> FindAll(int
skip, int take)

Sybase.Collections.GenericList<Customer> customers =
Customer.FindAll(10, 5);
```

Suppose the modeler defined the following Object Query for the Customer MBO in Sybase Unwired Workspace:

- **name –** findByFirstName
- **parameter –** String firstName
- **query definition –** SELECT x.* FROM Customer x WHERE x.fname = :firstName
- **return type –** Sybase.Collections.GenericList

The preceding Object Query results in two generated methods in `Customer.cs`:

```
public static Sybase.Collections.GenericList<Customer>
FindByFirstName(string firstName)
```

Query and Related Classes

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

Table 1. Query and Related Classes

Class	Description
Query	Defines arbitrary search methods and can be composed of search conditions, object/row state filter conditions, and data ordering information.
AttributeTest	Defines filter conditions for MBO attributes.
CompositeTest	Contains a method to combine test criteria using the logical operators AND, OR, and NOT to create a compound filter.
QueryResultSet	Provides for querying a result set for the dynamic query API.
SelectItem	Defines the entry of a select query. For example, "select x.attr1 from MBO x", where "X.attr1" represents one SelectItem.

Class	Description
Column	Used in a subquery to reference the outer query's attribute.

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

Arbitrary Find

The arbitrary find method lets custom device applications dynamically build queries based on user input. The `Query.DISTINCT` property has been added so you can exclude duplicate entries from the result set.

The arbitrary find method also lets the user specify a desired ordering of the results and object state criteria. A `Query` class is included in the client object API's core assembly sup-client.dll Sybase.Persistence namespace. The `Query` class is the single object passed to the arbitrary search methods and consists of search conditions, object/row state filter conditions, and data ordering information.

In MBO `Customer.cs`:

```
public static  Sybase.Collections.GenericList<sample.Customer>
FindWithQuery(Sybase.Persistence.Query query)
```

In Database class `SampleAppDB.cs`:

```
public static Sybase.Persistence.QueryResultSet
ExecuteQuery(Sybase.Persistence.Query query)
```

Define these conditions by setting properties in a query:

- **TestCriteria –** criteria used to filter returned data.
- **SortCriteria –** criteria used to order returned data.
- **Skip –** an integer specifying how many rows to skip. Used for paging.
- **Take –** an integer specifying the maximum number of rows to return. Used for paging.

Set the `Query.DISTINCT` property to `true` to exclude duplicate entries from the result set. The default value is `false` for entity types, and its usage is optional for all other types.

```
Query query1 = new Query();
query1.DISTINCT = true;
```

`TestCriteria` can be an `AttributeTest` or a `CompositeTest`.

Dynamic Query

You can construct a query SQL statement to query data from a local database. This query may across multiple tables (MBOs).

```
Query query2 = new Query();
query2.Select("c.fname,c.lname,s.order_date,s.region");
query2.From("Customer", "c");
```

```
//
// Convenience method for adding a join to the query
// Detailed construction of the join criteria
query2.Join("Sales_order", "s", "c.id", "s.cust_id");
AttributeTest ts = new AttributeTest();
ts.Attribute = ("fname");
ts.TestValue = "Beth";
query2.Where(ts);
QueryResultSet resultSet = SampleAppDB.ExecuteQuery(query2);
```

Note: A wildcard is not allowed in the `Select` clause. You must use explicit column names.

SortCriteria

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

Paging Data

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

Consider using the Query object to limit the result set:

```
Query props = new Query();
props.Skip =10;
props.Take = 5;

CustomerList customers = Customer.FindWithQuery(props);
```

AttributeTest

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

- IS_NULL
- NOT_NULL
- EQUAL
- NOT_EQUAL
- LIKE
- NOT_LIKE
- LESS_THAN
- LESS_EQUAL
- GREATER_THAN
- GREATER_EQUAL
- CONTAINS
- STARTS_WITH
- ENDS_WITH
- DOES_NOT_START_WITH

- DOES_NOT_END_WITH
- DOES_NOT_CONTAIN
- IN
- NOT_IN
- EXISTS
- NOT_EXISTS

For example, the C# .NET code shown below is equivalent to this SQL query:

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

```
Query query = new Query();
  AttributeTest test = new AttributeTest();
  test.Attribute = "id";
  Sybase.Collections.ObjectList v = new
Sybase.Collections.ObjectList();
    v.Add("1");
    v.Add("2");
    v.Add("3");

  test.Value = v;
  test.SetOperator(AttributeTest.IN);
  query.Where(test);
```

When using EXISTS and NOT_EXISTS, the attribute name is not required in the `AttributeTest`. The query can reference an attribute value via its alias in the outer scope. The C# .NET code shown below is equivalent to this SQL query:

```
SELECT a.id from AllType a where exists (select b.id from AllType b
where b.id = a.id)
```

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

Aggregate Functions

You can use aggregate functions in dynamic queries.

When using the `Query.Select(String)` method, you can use any of these aggregate functions:

Aggregate Function	Supported Datatypes
`COUNT`	integer
`MAX`	string, binary, char, byte, short, int, long, integer, decimal, float, double, date, time, dateTime
`MIN`	string, binary, char, byte, short, int, long, integer, decimal, float, double, date, time, dateTime
`SUM`	byte, short, int, long, integer, decimal, float, double
`AVG`	byte, short, int, long, integer, decimal, float, double

If you use an unsupported type, a `PersistenceException` is thrown.

```
Query query1 = new Query();
query1.Select("MAX(c.id), MIN(c.name) as minName");
```

Grouping Results

Apply grouping criteria to your results.

To group your results according to specific attributes, use the `Query.GroupBy(String groupByItem)` method. For example, to group your results by ID and name, use:

```
String groupByItem = ("c.id, c.name");
Query query1 = new Query();

//other code for query1

query1.GroupBy(groupByItem);
```

Filtering Results

Specify test criteria for group queries.

You can specify how your results are filtered by using the `Query.having(com.sybase.persistence.TestCriteria)` method for queries using `GroupBy`. For example, limit your AllType MBO's results to `c.id` attribute values that are greater than or equal to 0 using:

```
Query query2 = new Query();
  query2.Select("c.id, SUM(c.id)");
  query2.From("AllType", "c");
  ts = new AttributeTest();
```

```
  ts.SetAttribute("c.id");
  ts.SetTestValue("0");
  ts.SetOperator(AttributeTest.GREATER_EQUAL);
  query2.Where(ts);
  query2.GroupBy("c.id");

  ts2 = new AttributeTest();
  ts2.SetAttribute("c.id");
  ts2.SetTestValue("0");
  ts2.SetOperator(AttributeTest.GREATER_EQUAL);
  query2.Having(ts2);
```

Concatenating Queries

Concatenate two queries having the same selected items.

The `Query` class methods for concatenating queries are:

- `Union(Query)`
- `UnionAll(Query)`
- `Except(Query)`
- `Intersect(Query)`

This example obtains the results from one query except for those results appearing in a second query:

```
Query query1 = new Query();
... ... //other code for query1

Query query2 = new Query();
... ... //other code for query 2

Query query3 = query1.Except(query2);
SampleAppDB.ExecuteQuery(query3);
```

Subqueries

Execute subqueries using clauses, selected items, and attribute test values.

You can execute subqueries using the `Query.from(Query query, String alias)` method. For example, the C# .NET code shown below is equivalent to this SQL query:

```
SELECT a.id FROM (SELECT b.id FROM AllType b) AS a WHERE a.id = 1
```

Use this C# .NET code:

```
Query query1 = new Query();
query1.Select("b.id");
query1.From("AllType", "b");
Query query2 = new Query();
query2.Select("a.id");
query2.From(query1, "a");
AttributeTest ts = new AttributeTest();
ts.Attribute = "a.id";
ts.Value = 1;
query2.Where(ts);
```

```
Sybase.Persistence.QueryResultSet qs =
DsTestDB.ExecuteQuery(query2);
```

You can use a subquery as the selected item of a query. Use the `SelectItem` to set selected items directly. For example, the C# .NET code shown below is equivalent to this SQL query:

```
SELECT (SELECT count(1) FROM AllType c WHERE c.id >= d.id) AS cn, id
FROM AllType d
```

Use this C# .NET code:

```
Query selQuery = new Query();
selQuery.Select("count(1)");
selQuery.From("AllType", "c");
AttributeTest ttt = new AttributeTest();
ttt.Attribute = "c.id";
ttt.SetOperator(AttributeTest.GREATER_EQUAL);
Column cl = new Column();
cl.Alias = "d";
cl.Attribute = "id";
ttt.Value = cl;
selQuery.Where(ttt);

Sybase.Collections.GenericList<Sybase.Persistence.SelectItem>
selectItems = new
Sybase.Collections.GenericList<Sybase.Persistence.SelectItem>();
SelectItem item = new SelectItem();
item.Query = selQuery;
item.AsAlias = "cn";
selectItems.Add(item);
item = new SelectItem();
item.Attribute = "id";
item.Alias = "d";
selectItems.Add(item);
Query subQuery2 = new Query();
subQuery2.SelectItems = selectItems;
subQuery2.From("AllType", "d");
Sybase.Persistence.QueryResultSet qs =
DsTestDB.ExecuteQuery(subQuery2);
```

CompositeTest

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

Complex Example

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

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

```
Query props = new Query();
            //define the attribute based conditions
            //Users can pass in a string if they know the attribute
name. R1 column name = attribute name.
            CompositeTest innerCompTest = new CompositeTest();
            innerCompTest.Operator = CompositeTest.OR;
            innerCompTest.Add(new AttributeTest("state", "CA",
AttributeTest.EQUAL));
            innerCompTest.Add(new AttributeTest("state", "NY",
AttributeTest.EQUAL));
            CompositeTest outerCompTest = new CompositeTest();
            outerCompTest.Operator = CompositeTest.OR;
            outerCompTest.Add(new AttributeTest("fname", "Jane",
AttributeTest.EQUAL));
            outerCompTest.Add(new AttributeTest("lname", "Doe",
AttributeTest.EQUAL));
            outerCompTest.Add(innerCompTest);
            //define the ordering
            SortCriteria sort = new SortCriteria();

            sort.Add("fname", SortOrder.ASCENDING);
            sort.Add("lname", SortOrder.ASCENDING);
            //set the Query object
            props.TestCriteria = outerCompTest;
            props.SortCriteria = sort;
            props.Skip = 10;
            props.Take = 5;
            Sybase.Collections.GenericList<Customer> customers2 =
Customer.FindWithQuery(props);
```

QueryResultSet

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

Example

The following example shows how to filter a result set and get values by taking data from two mobile business objects, creating a Query, filling in the criteria for the query, and filtering the query results:

```
Sybase.Persistence.Query query = new Sybase.Persistence.Query();
            query.Select("c.fname,c.lname,s.order_date,s.region");
            query.From("Customer ", "c");
            query.Join("SalesOrder ", "s", " s.cust_id ", "c.id");
            AttributeTest at = new AttributeTest();
            at.Attribute = "lname";
            at.TestValue = "Devlin";
            query.TestCriteria = at;
            QueryResultSet qrs = SampleAppDB.ExecuteQuery(query);
            while(qrs.Next())
            {
                Console.Write(qrs.GetString(1));
                Console.Write(",");
                Console.WriteLine(qrs.GetStringByName("c.fname"));
```

```
                Console.Write(qrs.GetString(2));
                Console.Write(",");
                Console.WriteLine(qrs.GetStringByName("c.lname"));

                Console.Write(qrs.GetString(3));
                Console.Write(",");

Console.WriteLine(qrs.GetStringByName("s.order_date"));

                Console.Write(qrs.GetString(4));
                Console.Write(",");
                Console.WriteLine(qrs.GetStringByName("s.region"));
            }
```

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.

```
Customer customer = Customer.FindByPrimaryKey(101);
            Sybase.Collections.GenericList<SalesOrder> orders =
customer.Orders;
```

You can also use the Query class to filter the return MBO list data.

```
Query props = new Query();
… // set query parameters
Sybase.Collections.GenericList<SalesOrder> orders =
customer.GetOrdersFilterBy(props);
```

Operations APIs

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

Any parameters in the create, update, or delete operation that are mapped to the object's attributes are handled internally by the Client Object API, and are not exposed. Any parameters not mapped to the object's attributes are left as parameters in the Generated Object API.

Note: If the Sybase Unwired Platform object model defines one instance of a create operation and one instance of an update operation, and all operation parameters are mapped to the object's attributes, then a `Save` method can be automatically generated which, when called internally, determines whether to insert or update data to the local client-side database. In

other situations, where there are multiple instances of create or update operations, it is not possible to automatically generate such a Save method.

Create Operation

To execute a create operation on an MBO, create a new MBO instance, set the MBO attributes, then call the Save() or Create() operation.

```
Customer cust = new Customer();
cust.Fname  =  "supAdmin" ;
cust.Company_name =   "Sybase";
cust.Phone = "777-8888";
cust.Create();// or cust.Save();
cust.SubmitPending();
```

Update Operation

To execute update operations on an MBO, get an instance of the MBO, set the MBO attributes, then call either the `Save()` or `Update()` operations.

```
Customer cust = Customer.FindByPrimaryKey(101);
cust.Fname = "supAdmin";
cust.Company_name = "Sybase";
cust.Phone = "777-8888";
cust.Update();// or cust.Save();
cust.SubmitPending();
```

Delete Operation

To execute delete operations on an MBO, get an instance of the MBO, set the MBO attributes, then call the `Delete()` operation.

```
Customer cust = Customer.FindByPrimaryKey(101);
cust.Delete();
```

Other Operation

Operations that are not create, update, or delete operations are called “Other” operations. An Other operation class is generated for each operation in the MBO that is not a create, update or delete operation.

Suppose the Customer MBO has an Other operation “other”, with parameters “p1” (string), “p2” (int) and “p3” (date). This results in a `CustomerOtherOperation` class being generated, with “p1”, “p2” and “p3” as its attributes.

To invoke the Other operation, create an instance of `CustomerOtherOperation`, and set the correct operation parameters for its attributes. This code provides an example:

```
CustomerOtherOperation other = new CustomerOtherOperation();
other.P1 = “somevalue”;
other.P2 = 2;
other.P3 = System.DateTime.Now;
other.Save(); // or other.Create()
other.SubmitPending();
```

Cascade Operations

Composite relationships are cascaded. Cascade operations allow a single synchronization to execute a chain of related CUD operations. Multi-level insert is a special case for cascade operations. It allows parent and children objects to be created in one round without having to synchronize multiple times.

Refer to Unwired WorkSpace documentation (Relationship Guidelines and Multi-level insert operations) for information about defining relationships that support cascading (composite) operations.

Consider creating a Customer and a new SalesOrder at the same time on the client side, where the SalesOrder has a reference to the new Customer identifier. The following example demonstrates a multilevel insert:

```
Customer customer = new Customer();
customer.Fname = “firstName”;
customer.Lname = “lastName”;
customer.Phone = “777-8888”;
customer.Save();
SalesOrder order = new SalesOrder();
order.Customer = customer;
order.Order_date = DateTime.Now;
order.Region = "Eastern";
order.Sales_rep = 102;
customer.Orders.Add(order);
//Only the parent MBO needs to call Save()
customer.Save();
//Must submit parent
customer.SubmitPending();
```

To insert an order for an existing customer, first find the customer, then create a sales order with the customer ID retrieved:

```
Customer customer = Customer.FindByPrimaryKey(102);
SalesOrder order = new SalesOrder();
order.Customer = customer;
order.Order_date = DateTime.UtcNow;
order.Region = "Eastern";
order.Sales_rep = 102;
customer.Orders.Add(order);
order.Save();
customer.SubmitPending();
```

To update MBOs in composite relationships, perform updates on every MBO to change and call `SubmitPending` on the parent MBO:

```
Customer cust = Customer.FindByPrimaryKey(101);
Sybase.Collections.GenericList<SalesOrder> orders = cust.Orders;
SalesOrder order = orders[0];
order.Order_date = DateTime.Now;
order.Save();
cust.SubmitPending();
```

To delete a single child in a composite relationship, call the child's `Delete` method, and the parent MBO's `SubmitPending`.

```
Customer cust = Customer.FindByPrimaryKey(101);
Sybase.Collections.GenericList<SalesOrder> orders = cust.Orders;
SalesOrder order = orders[0];
order.Delete();
cust.SubmitPending();
```

To delete all MBOs in a composite relationship, call `Delete` and `SubmitPending` on the parent MBO:

```
Customer cust = Customer.FindByPrimaryKey(101);
cust.Delete();
cust.SubmitPending();
```

Note: For non-composite relationships, `SubmitPending` must be called on each and every MBO.

See the Sybase Unwired Platform online documentation for specific multilevel insert requirements.

Pending Operation

You can manage pending operations using these methods:

- **CancelPending –** cancels the previous create, update, or delete operations on the MBO. It cannot cancel submitted operations.
- **SubmitPending –** submits the operation so that it can be replayed on the Unwired Server. For message-based synchronization, a replay request is sent directly to the Unwired Server. For replication-based synchronization, a request is sent to the Unwired Server during a synchronization.
- **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.
- **CancelPendingOperations –** cancels all the pending records for the entity. This method internally invokes the `CancelPending` method on each of the pending records.

```
Customer customer = Customer.FindByPrimaryKey(101);
if(errorHappened)
{
  Customer.CancelPending();
}
else
{
  customer.SubmitPending();
}
```

Passing Structures to Operations

Structures hold complex datatypes (for example a string list, class or MBO object, or a list of objects) that enhance interactions with certain enterprise information systems (EIS) data

sources, such as SAP and Web services, where the mobile business object (MBO) requires complex operation parameters.

An Unwired WorkSpace project includes an example MBO that is bound to a Web service data source that includes a create operation that takes a structure as an operation parameter. MBOs differ depending on the data source, configuration, and so on, but the principles are similar.

The SimpleCaseList MBO contains a create operation that has a number of parameters, including a parameter named _HEADER_ that is a structure datatype named AuthenticationInfo, defined as:

```
AuthenticationInfo
   userName: String
   password: String
   authentication: String
   locale: String
   timeZone: String
```

Structures are implemented as classes, so the parameter _HEADER_ is an instance of the AuthenticationInfo class. The generated Java code for the create operation is:

```
public  void Create(Authentication _HEADER_,string escalated,string
hotlist,
string orig_Submitter,string pending,string workLog);
```

This example demonstrates how to initialize the AuthenticationInfo class instance and pass them, along with the other operation parameters, to the create operation:

```
AuthenticationInfo authen = new AuthenticationInfo();
           authen.UserName = "Demo";


            SimpleCaseList newCase = new SimpleCaseList();
            newCase.Case_Type = "Incident";
            newCase.Category = "Networking";
            newCase.Department = "Marketing";
            newCase.Description = "A new help desk case.";
            newCase.Item = "Configuration";
            newCase.Office = "#3 Sybase Drive";
            newCase.Submitted_By = "Demo";
            newCase.Phone_Number = "#0861023242526";
            newCase.Priority = "High";
            newCase.Region = "USA";
            newCase.Request_Urgency = "High";
            newCase.Requester_Login_Name = "Demo";
            newCase.Requester_Name = "Demo";
            newCase.Site = "25 Bay St, Mountain View, CA";
            newCase.Source = "Requester";
            newCase.Status = "Assigned";
            newCase.Summary = "MarkHellous was here Fix it.";
            newCase.Type = "Access to Files/Drives";
            newCase.Create_Time = System.DateTime.Now;

           newCase.Create (authen, “Other”, “Other”, “false”, “work
log”);
            newCase.SubmitPending();
```

Local Business Object

Defined in Unwired WorkSpace, local business objects are not bound to EIS data sources, so cannot be synchronized. Instead, they are objects that are used as local data store on device. Local business objects do not call `submitPending`, or perform a replay or import from the Unwired Server

An example of a local business object:

```
LoginStatus status= new LoginStatus ();
   status.Id = 123;
   status.Time = DateTime.Now;
   status.Success = true;
   status.Create();

   long savedId = 123;
   LoginStatus status = LoginStatus.Find(savedId);
   status.Success = false;
   status.Update();


   long savedId = 123;
   LoginStatus status = LoginStatus.Find(savedId);
   status.Delete();
```

Personalization APIs

Personalization keys allow the application to define certain input parameter values that differ (are personalized) from each mobile user. The Personalization APIs allow you to manage personalization keys, and get and set personalization key values.

Type of Personalization Keys

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

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

Get or Set Personalization Key Values

The `PersonalizationParameters` class is generated automatically for managing personalization keys.

The following code provides an example on how to set a personalization key, and pass an array of values and array of objects:

```
PersonalizationParameters pp =
SampleAppDB.GetPersonalizationParameters();
pp.MyIntPK = 10002;
pp.Save();
```

```
Sybase.Collections.IntList il = new Sybase.Collections.IntList();
il.Add(10001);
il.Add(10002);
pp.MyIntListPK = il;
pp.Save();
Sybase.Collections.GenericList<MyData> dl = new
Sybase.Collections.GenericList<MyData>(); //MyData is a structure
type defined in tooling
MyData md = new MyData();
md.IntMember = 123;
md.StringMember = "abc";
dl.Add(md);
pp.MyDataListPK = dl;
pp.Save();
```

If a synchronization parameter is personalized, you can overwrite the value of that parameter with the personalization value.

Note: For detailed description on personalization key usage, see the *Sybase Unwired Platform online help*.

Object State APIs

The object state APIs provide methods for returning information about the state of an entity.

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	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.

Name	C# Type	Description
IsUpdated	bool	Returns true if this entity has been updated or changed in the database, and one of the following is true: • The entity has not yet been submitted to the server with a replay request. • The entity has been submitted to the server, but the server has not finished processing the request. • The server rejected the replay request (replayFailure message received).
Pending	bool	Returns true for any row that represents a pending create, update, or delete operation, or a row that has cascading children with a pending operation.
PendingChange	char	If pending is true, 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'.
ReplayCounter	long	Returns a long value which 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.
ReplayPending	long	Returns a long value. When a pending row is submitted to the server, the value of `ReplayCounter` is copied to `ReplayPending`. This allows the client code to detect if a row has been changed since it was submitted to the server (that is, if the value of `ReplayCounter` is greater than `ReplayPending`).
ReplayFailure	long	Returns a long value. When the server responds with a `ReplayFailure` message for a row that was submitted to the server, the value of `ReplayCounter` is copied to `ReplayFailure`, and `ReplayPending` is set to 0.

Entity State Example

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, the requested change becomes pending. To apply the pending change, call `SubmitPending` on the entity, or `SubmitPendingOperations` on the MBO class:

```
Customer e = new Customer();
e.Name = "Fred";
e.Address = "123 Four St.";
e.Create(); // create as pending
e.SubmitPending(); // submit to server

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.

For message-based sychronization, 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.

For replication-based synchronization, the call to `SubmitPending` creates a replay record in local database. When the `DBClass.Synchronize()` method is called, the replay records are uploaded to Unwired Server. Unwired Server processes the replay records one by one and either accepts or rejects it.

At the end of the synchronization, the replay results are downloaded to client along with any created, updated or deleted rows that have changed on the Unwired Server as a result of the Replay requests. These changes are written to the client database and marked as rows that are not pending.

When the operation is successful, 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. If the Unwired Server rejects the requested change, the entity remains in the pending state, with its ReplayFailed attribute set to indicate that the change was rejected. The Unwired Server may optionally send a log record to the client.

The `LogRecord` interface for both message-based synchronization and replication-based synchronization has the following getter methods to access information about the log record:

Method Name	C# Type	Description
Component	string	Name of the MBO for the row for which this log record was written.

Method Name	C# Type	Description
EntityKey	string	String representation of the primary key of the row for which this log record was written.
Code	int	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	string	Descriptive message from the server with the reason for the log record.
Operation	string	The operation (create, update, or delete) that caused the log record to be written.
RequestId	string	The id of the replay message sent by the client that caused this log record to be written.
Timestamp	System.DateTime?	Date and time of the log record.

If a rejection is received, the application can use the entity method `GetLogRecords` or the database class method `SampleDB.GetLogRecords(query)` to access the log records and get the reason:

```
Sybase.Collections.GenericList<Sybase.Persistence.ILogRecord> logs =
e.GetLogRecords();
for(int i=0; i<logs.Size(); i++)
{
Console.WriteLine("Entity has a log record:");
Console.WriteLine("Code = {0}",logs[i].Code);
Console.WriteLine("Component = {0}",logs[i].Component);
Console.WriteLine("EntityKey = {0}",logs[i].EntityKey);
Console.WriteLine("Level = {0}",logs[i].Level);
Console.WriteLine("Message = {0}",logs[i].Message);
Console.WriteLine("Operation = {0}",logs[i].Operation);
Console.WriteLine("RequestId = {0}",logs[i].RequestId);
```

```
Console.WriteLine("Timestamp = {0}",logs[i].Timestamp);
}
```

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

Mobile Business Object States

A mobile business object can be in one of three states:

- Original state, the state before any create, update, or delete operation.
- Downloaded state, the state downloaded from the Unwired Server.
- Current state, the state after any create, update, or delete operation.

The mobile business object class provides properties or methods for querying the original state and the downloaded state:

```
public sample.Customer GetOriginalState()
public Customer DownloadState;
```

The original state is valid only before the application synchronizes with the Unwired Server. After synchronization has completed successfully, the original state is cleared and set to null.

```
Customer cust = Customer.FindByPrimaryKey(101); // state 1
cust.Fname = "supAdmin";
cust.Company_name = "Sybase";
cust.Phone = "777-8888";
cust.Save(); // state 2
Customer org = cust.GetOriginalState(); // state 1
//suppose there is new download for Customer 101 here
Customer download = cust.DownloadState; // state 3
cust.CancelPending(); // state 3
```

Using all three states, the application can resolve most conflicts that may occur.

Refresh Operation

The refresh operation of an MBO allows you to refresh the MBO state from the client database.

The following code provides an example:

```
Customer cust = Customer.FindByPrimaryKey(101);
cust.Fname = “newName”;
cust.Refresh();// newName is discarded
```

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.

ClearRelationshipObjects

Security APIs

Unwired Server supports encryption of client data and the database.

DataVault

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

You can use the `DataVault` class for on-device persistent storage of certificates, database encryption keys, passwords, and other sensitive items. Use this class to:

- Create a vault
- Set a vault's properties
- Store objects in a vault
- Retrieve objects from a vault
- Change the password used to access a vault

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

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

CreateVault

Creates a new secure store.

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

Syntax

```
public static DataVault CreateVault(
   string sDataVaultID,
   string sPassword,
   string sSalt
)
```

Parameters

- **sDataVaultID –** The vault name.
- **sPassword –** The password.
- **sSalt –** The encryption salt value.

Returns

CreateVault creates a `DataVault` instance.

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

Examples

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

```
DataVault vault = null;
if (!DataVault.VaultExists("myVault"))
{
   vault = DataVault.CreateVault("myVault", "password", "salt");
}
else
{
   vault = DataVault.GetVault("myVault");
}
```

VaultExists

Tests whether the specified vault exists.

Syntax

```
public static bool VaultExists(string sDataVaultID)
```

Parameters

- **sDataVaultID –** The vault name.

Returns

VaultExists can return the following values:

Returns	Indicates
true	The vault exists.
false	The vault does not exist.

Examples

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

```
if (DataVault.VaultExists("myVault"))
{
   DataVault.DeleteVault("myVault");
}
```

GetVault
Retrieves a vault.

Syntax

```
public static DataVault GetVault(string sDataVaultID)
```

Parameters

- **sDataVaultID –** The vault name.

Returns

GetVault returns a `DataVault` instance.

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

DeleteVault
Deletes the specified vault from on-device storage.

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

Syntax

```
public static void DeleteVault(string sDataVaultID)
```

Parameters

- **sDataVaultID –** The vault name.

Examples

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

```
if (DataVault.VaultExists("myVault"))
{
   DataVault.DeleteVault("myVault");
}
```

Lock
Locks the vault.

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

Syntax

```
public void Lock()
```

Examples

- **Locks the data vault. –** Prevents changing the vaults properties or stored content.
  ```
  vault.Lock();
  ```

IsLocked
Tests whether the vault is locked.

Syntax

```
public bool IsLocked()
```

Returns

IsLocked can return the following values:

Returns	Indicates
true	The vault is locked.
false	The vault is unlocked.

Unlock
Unlocks the vault.

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

Syntax

```
public void Unlock(string sPassword, string sSalt)
```

Parameters

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

Returns

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

Examples

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

```
if (vault.IsLocked())
{
   vault.Unlock("password", "salt");
}
```

SetLockTimeout

Determines how long a vault remains unlocked.

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

Syntax

```
public void SetLockTimeout(int iPeriod)
```

Parameters

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

Examples

- **Set the Lock Timeout –** Sets the lock timeout to 1 hour.

```
vault.SetLockTimeout( 3600 );
```

GetLockTimeout

Retrieves the configured lock timeout period.

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

Syntax

```
public int GetLockTimeout()
```

Returns

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

Examples

- **Set the Lock Timeout –** Retrieves the lock timeout in seconds.
```
int timeout = vault.GetLockTimeout();
```

SetRetryLimit

Sets the retry limit value for the vault.

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

Syntax
```
public void SetRetryLimit(int iLimit)
```

Parameters

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

Examples

- **Set the Retry Limit –** Sets the retry limit to 5 attempts.
```
vault.SetRetryLimit( 5 );
```

GetRetryLimit

Retrieves the retry limit value for the vault.

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

Syntax
```
public int GetRetryLimit()
```

Returns

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

Examples

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

```
int retrylimit = vault.GetRetryLimit();
```

SetString

Stores a string object in the vault.

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

Syntax

```
public void SetString(
   string sName,
   string sValue
)
```

Parameters

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

Examples

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

```
string teststring = "ABCDEFabcdef";
try
{
   vault.Unlock("password", "salt");
   vault.SetString("testString", teststring);
}
catch (DataVaultException e)
{
   Console.WriteLine("Exception: " + e.ToString());
}
finally
{
   vault.Lock();
}
```

GetString

Retrieves a string value from the vault.

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

Syntax

```
public string GetString(string sName)
```

Parameters

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

Returns

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

Examples

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

```
try
{
   vault.Unlock("password", "salt");
   String retrievedstring = vault.GetString("testString");
}
catch (DataVaultException e)
{
   Console.WriteLine("Exception: " + e.ToString());
}
finally
{
   vault.Lock();
}
```

SetValue

Stores a binary object in the vault.

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

Syntax

```
public void SetValue(
   string sName,
   byte[] baValue
)
```

Parameters

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

Examples

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

```
try
{
   vault.Unlock("password", "salt");
   vault.SetValue("testValue", new byte[] { 1, 2, 3, 4, 5});
}
catch (DataVaultException e)
{
   Console.WriteLine("Exception: " + e.ToString());
}
finally
{
   vault.Lock();
}
```

GetValue

Retrieves a binary object from the vault.

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

Syntax

```
public byte[] GetValue(string sName)
```

Parameters

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

Returns

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

Examples

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

```
try
{
   vault.Unlock("password", "salt");
   byte[] retrievedvalue = vault.GetValue("testValue");
}
catch (DataVaultException e)
```

```
{
   Console.WriteLine("Exception: " + e.ToString());
}
finally
{
   vault.Lock();
}
```

ChangePassword

Changes the password for the vault.

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

Syntax

```
public void ChangePassword(
   string sPassword,
   string sSalt
)
```

Parameters

- **sPassword –** The new password.
- **sSalt –** The new encryption salt value.

Examples

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

```
try
{
   vault.Unlock("password", "salt");
   vault.ChangePassword("newPassword", "newSalt");
}
catch (DataVaultException e)
{
   Console.WriteLine("Exception: " + e.ToString());
}
finally
{
   vault.Lock();
}
```

Utility APIs

The Utility APIs allow you to customize aspects of logging, callback handling, and generated code.

Using the Logger and LogRecord APIs

`LogRecord` is used to store two types of logs.

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

`DBClass.GetLogger` – gets the log API. The client can write its own records using the log API. For example:

```
ILogger logger = SampleAppDB.GetLogger();
    logger.Debug(“Write this string to the log records table");
    SampleAppDB. SubmitLogRecords();
```

`DBClass.GetLogRecords` – gets the log records received from the server. For example:

```
Query query = new Query();
query.TestCriteria =
Sybase.Persistence.AttributeTest.Equal("component", “Customer”);
Sybase.Persistence.SortCriteria sortCriteria = new
Sybase.Persistence.SortCriteria();
sortCriteria.Add("requestId",
Sybase.Persistence.SortOrder.DESCENDING);
query.SortCriteria = sortCriteria;

GenericList<ILogRecord> loglist = SampleAppDB.GetLogRecords(query);
```

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}
```

GenerateId

You can use the `GenerateId` method in the `LocalKeyGenerator` or `KeyGenerator` classes to generate an ID when creating a new object for which you require a primary key or surrogate key.

This method in the `LocalKeyGenerator` class generates a unique ID for the package on the local device:

```
public static long GenerateId()
```

This method in the `KeyGenerator` class generates a unique ID for the same package across all devices:

```
public static long GenerateId()
```

Callback Handlers

To receive callbacks, you must register a `CallBackHandler` with the generated database class, the entity class, or both. You can create a handler by extending the `DefaultCallbackHandler` class, or by implementing the interface, ICallbackHandler.

In your handler, override the particular callback you want to use (for example, `OnReplaySuccess`). The callback is executed in the thread that is performing the action (for example, replay). When you receive the callback, the particular activity is already complete.

Callbacks in the CallbackHandler interface include:

```
namespace Sybase.Persistence
{
  // Summary:
  //     A default implementation for the
Sybase.Persistence.ICallbackHandler interface.
  //      Application programmers should implement their own
CallbackHandler.
  //
  // Remarks:
  //    This class contains dummy implements, application programmer
should not use
  //     this default implementation.
  public class DefaultCallbackHandler : ICallbackHandler
```

```
  {
    public DefaultCallbackHandler();

    public virtual void BeforeImport(object o);
    //
    // Summary:
    //     This method will be invoked when device connection status
is changing When
    //     there is connection error, there is a non-zero error code
and the errorMessage
    //     will not be null
    //
    // Parameters:
    //   connStatus:
    //     Connection status
    //
    //   connType:
    //     Connection type
    //
    //   errorCode:
    //     Error code
    //
    //   errorMessage:
    //     Error message
   public virtual void OnConnectionStatusChange(int connStatus, int
connType, int errorCode, string errorMessage);
    public virtual void OnImport(object o);
    public virtual void OnImportSuccess();
    public virtual void OnLoginFailure();
    public virtual void OnLoginSuccess();
    public virtual void OnMessageException(Exception ex);
    public virtual void OnRecoverFailure();
    public virtual void OnRecoverSuccess();
    public virtual void OnReplayFailure(object o);
    public virtual void OnReplaySuccess(object o);
    public virtual void OnResetSuccess();
    public virtual void OnResumeSubscriptionFailure();
    public virtual void OnResumeSubscriptionSuccess();
    public virtual void OnSearchFailure(object o);
    public virtual void OnSearchSuccess(object o);
    public virtual void OnStorageSpaceLow();
    public virtual void OnStorageSpaceRecovered();
    public virtual void OnSubscribeFailure();
    public virtual void OnSubscribeSuccess();
    public virtual void OnSubscriptionEnd();
    public virtual void OnSuspendSubscriptionFailure();
    public virtual void OnSuspendSubscriptionSuccess();
    public virtual SynchronizationAction
OnSynchronize(GenericList<ISynchronizationGroup> groups,
SynchronizationContext context);
    public virtual void
OnSynchronizeFailure(GenericList<ISynchronizationGroup> groups);
    public virtual void
OnSynchronizeSuccess(GenericList<ISynchronizationGroup> groups);
    public virtual void OnTransactionCommit();
    public virtual void OnTransactionRollback();
```

```
    public virtual void OnUnsubscribeFailure();
    public virtual void OnUnsubscribeSuccess();
  }
}
```

This code example shows how to create and register a handler to receive callbacks:

```
public class MyCallbackHandler : DefaultCallbackHandler
{
    // implementation
}

CallbackHandler handler = new MyCallbackHandler();
MyPackageDB.RegisterCallbackHandler(handler);
//or Customer.RegisterCallbackHandler(handler);
```

Client Database APIs

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

```
public static void CreateDatabase()
public static void DeleteDatabase()
```

Typically, `CreateDatabase` does not need to be called since it will be called internally when necessary. An application may use `DeleteDatabase` when the client database contains corrupted data and needs to be cleared.

Installing X.509 Certificates on Windows Mobile Devices and Emulators

Install the *.p12 certificate on a Windows Mobile device or simulator and select it during authentication.

1. Launch the simulator or device.
2. Start the Windows synchronization software and cradle the device.
3. Use File Explorer to copy the `*.p12` certificate to the simulator or device.
4. Navigate to and double-click the certificate.
5. Enter the password at the prompt and click **Done**.
   An informational window indicates the certificate installed successfully.

Windows Mobile Sample Code

This sample code illustrates importing the certificate and setting up login credentials, as well as other APIs related to certificate handling:

```
/// End2EndDB is a generated RBS class
///First install certificates on your emulator, for example
"Sybase101.p12"

//Test getting certificate from certificate store
CertificateStore myStore =
CertificateStore.GetDefault();
```

```
string filter1 = "Sybase";
StringList labels = myStore.CertificateLabels(filter1, null);
string aLabel = labels.Item(0);
LoginCertificate lc = myStore.GetSignedCertificate(aLabel,
"password");

// Save the login certificate to your synchronization profile
End2EndDB.GetSynchronizationProfile().Certificate = lc;

// Login to Unwired Server without username/password
End2EndDB.LoginToSync();

//Perform synchronization
End2EndDB.Synchronize();

// Save the login certificate to your data vault
DataVault vault = null;
if (!DataVault.VaultExists("myVault"))
{
    vault = DataVault.CreateVault("myVault", "password", "salt");
}
else
{
    vault = DataVault.GetVault("myVault");
}
vault.Unlock("password", "salt");
lc.Save("myLabel", vault);

// Get certificate that was previously loaded from the data vault
LoginCertificate newLc = LoginCertificate.Load("myLabel", vault);
Debug.Assert(newLc.SubjectDN.Equals(lc.SubjectDN));

// Delete the certificate from the data vault
LoginCertificate.Delete("myLabel", vault);
```

Single Sign-On With X.509 Certificate Related Object API

Use these classes and attributes when developing mobile applications that require X.509 certificate authentication.

- CertificateStore class - wraps platform-specific key/certificate store class, or file directory
- LoginCertificate class - wraps platform-specific X.509 distinguished name and signed certificate
- ConnectionProfile class - includes the certificate attribute used for Unwired Server synchronization.

Refer to the Javadocs that describe implementation details.

Importing a Certificate Into the Data Vault

Obtain a certificate reference and store it in a password protected data vault to use for X.509 certificate authentication.

The difference between importing a certificate from a system store or a file directory is determined by how you obtain the `CertificateStore` object. In either case, only a label and password are required to import a certificate.

```
//Obtain a reference to the certificate store
CertificateStore myStore = CertificateStore.GetDefault();

//List all certificate labels from the certificate store
StringList labels = myStore.CertificateLabels();

//List the certificate labels filtered by subject
String filter1 = "Sybase";
labels = myStore.CertificateLabels(filter1, null);

//Get a LoginCertificate from the certificate store
string aLabel = ... //ask user to select a label
string password = ... //prompt user for password
LoginCertificate lc = myStore.GetSignedCertificate(aLabel,
password);

//Save/Load/Delete...LoginCertificate
//Create or lookup a data vault
DataVault vault = null;
if (!DataVault.VaultExists("myVault"))
{
    vault = DataVault.CreateVault("myVault", "password", "salt");
}
else
{
    vault = DataVault.GetVault("myVault");
}
```

Selecting a Certificate for Unwired Server Connections

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

```
//Unlock the vault before using it
vault.Unlock("password", "salt");
//Save the certificate with specified label
lc.Save("myLabel", vault);
//load the certificate from data vault by label
LoginCertificate newLc = LoginCertificate.Load("myLabel", vault);
//Delete the certificate from the data vault
LoginCertificate.Delete("myLabel", vault);
```

Connecting to Unwired Server With a Certificate

Once the certificate property is set, use the `LoginToSync` or `OnlineLogin` API with no parameters to connect to Unwired Server with the login certificate.

```
//connect to Unwired Server with the login certificate
MyPackageDB.GetSynchronizationProfile().Certificate = lc;
MyPackageDB.LoginToSync();
```

Exceptions

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

Handling Exceptions

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

Server-Side Exceptions

Exceptions thrown on the Unwired Server are logged in both the server log and in `LogRecord`. For `LogRecord`, the exception gets downloaded to the device automatically during synchronization (replication-based synchronization) or when importing a message (message-based synchronization).

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 2. Recoverable Error Codes

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

Table 3. Non-recoverable Error Codes

Error Code	Probable Cause	Manual Recovery Action
401	Backend EIS credentials wrong.	Change the connection information, or backend user password.

Error Code	Probable Cause	Manual Recovery Action
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
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 4. 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

Constant	Description	HTTP Error Code
JCO_ERROR_RESOURCE	Indicates that JCO has run out of resources such as connections in a connection pool	503
JCO_ERROR_STATE_BUSY	The remote SAP system is busy. Try again later	503

Client-Side Exceptions

Device applications are responsible to catch and handle exceptions thrown by the client object API.

For message-based synchronization, you can catch exceptions for background thread message processing by creating a callback handler and implementing `OnMessageException` methods.

Note: Refer to *Callback Handlers* on page 61 for more information.

Exception Classes

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

- **SynchronizeException –** this exception is thrown when an error occurs during synchronization.
- **PersistenceException –** this exception is thrown when trying to load an MBO that is inside the local database.
- **SystemException –** this exception is thrown for uncategorized errors, and is typically unrecoverable.
- **ObjectNotFoundException –** this exception is thrown when trying to load an MBO that is not inside the local database.
- **NoSuchOperationException –** this exception is thrown when trying to call a method (using the Object Manager API) but the method is not defined for the MBO.
- **NoSuchAttributeException –** this exception is thrown when trying to access an attribute (using the Object Manager API) but the attribute is not defined for the 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.

MetaData and Object Manager API

Some applications or frameworks may wish to operate against MBOs in a generic manner by invoking MBO operations without prior knowledge of MBO classes. This can be achieved by using the MetaData and Object Manager APIs.

These APIs allow retrieving the metadata of packages, MBOs, attributes, operations and parameters during runtime. The APIs are especially useful for a runtime environment without a reflection mechanism such as J2ME.

You can generate metadata classes using the `-md` code generation option. You can use the `-rm` option to generate the object manager class.

You can also generate metadata classes by selecting the option "Generate metadata classes" or "Generate metadata and object manager classes" option in the code generation wizard in the mobile application project.

ObjectManager

The `ObjectManager` class allows an application to call the Object API in a reflection style.

```
IObjectManager rm = new MyDatabase_RM();
ClassMetaData customer = MyDatabase.Metadata.GetClass(“Customer”);
AttributeMetaData lname = customer.GetAttribute(“lname”);
OperationMetaData save = customer.GetOperation(“save”);
object myMBO = rm.NewObject(customer);
rm.SetValue(myMBO, lname, “Steve”);
rm.Invoke(myMBO, save, new ObjectList());
```

DatabaseMetaData

The `DatabaseMetaData` class holds package level metadata. You can use it to retrieve data such as synchronization groups, default database file, and MBO metadata .

```
DatabaseMetaData dmd = SampleAppDB.Metadata;
    foreach (String syncGroup in dmd.SynchronizationGroups)
    {
        Console.WriteLine(syncGroup);
    }
```

EntityMetaData

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

```
EntityMetaData customerMetaData = Customer.GetMetaData();
    AttributeMetaData lname =
customerMetaData.GetAttribute("lname");
    OperationMetaData save = customerMetaData.GetOperation("save");
```

AttributeMetaData

The AttributeMetaData class holds metadata for an attribute such as attribute name, column name, type, and maxlength.

```
Console.WriteLine(lname.Name);
Console.WriteLine(lname.Column);
Console.WriteLine(lname.MaxLength);
```

Replication-Based Synchronization APIs

Replication-based synchronization (RBS) clients receive device notifications when a data change is detected for any of the MBOs in the synchronization group to which they are subscribed.

The following operations are available when performing replication-based synchronization.

IsSynchronized() and GetLastSynchronizationTime

For replication-based synchronization applications, the package database class provides the following two methods for querying the synchronized state and the last synchronization time of a certain synchronization group:

```
/// Returns if the synchronizationGroup was synchronized
public static bool IsSynchronized(string synchronizationGroup)

/// Returns the last synchronization time of the synchronizationGroup
public static System.DateTime GetLastSynchronizationTime(string
synchronizationGroup)
```

Push Configuration APIs

The push configuration APIs provide methods for configuring push through lightweight polling (LWP).

Note: To use the push notification API in the Object API, the Sybase Server Sync Tool must be installed on the device. You can get the installer from `\<UnwiredPlatform_InstallDir>\ClientAPI\RBS\WM\ServerSync` *.CAB.

LWPPush

The following APIs support push notification in the generated database class.

The `RegisterServerSyncConfiguration()` method registers a synchronization configuration from a connection profile and generates a configuration file for the `SybaseServerSync` application, which is required for SIS push notification.

```
//Register server sync configuration based on the connection profile
properties.
 //This generates a configuration file for SybaseServerSync
application under \Application Data\
MyDatabase.RegisterServerSyncConfiguration();
```

The `LaunchServerSyncHelper()` method starts the `SybaseServerSync` application that provides the lightweight polling used for SIS push notification.

```
// Starts LWP used for SIS push notification
MyDatabase.LaunchServerSyncHelper()
```

The client sets the SIS push configuration parameters using `SynchronizationGroup`.

```
Sybase.Persistence.ISynchronizationGroup
GetSynchronizationGroup(string syncGroup)

ISynchronizationGroup sg =
MyDatabase.GetSynchronizationGroup("ofs");
sg.EnableSIS = true;
// sg.EnableSIS = false
sg.Interval = 0;
sg.Save();
```

The following method registers a callback handler to configure push notifications through lightweight polling.

```
// Install a user-defined callback handler
MyDatabase.RegisterCallbackHandler(new MyCallbackHandler());
```

The following method starts a background thread to do synchronization of push notifications.

```
MyDatabase.StartBackgroundSynchronization();
```

The following method stops the background thread performing synchronization on push notifications.

```
MyDatabase.StopBackgroundSynchronization();
```

The `ShutdownServerSyncHelper()` method stops the `SybaseServerSync` application.

```
// Stops LWP used for SIS push notification
MyDatabase.ShutdownServerSyncHelper()
```

Creating a Replication-based Push Application

Create a single device application using the Push Synchronization APIs described in this section.

Develop the push application directly from generated mobile business object (MBO) code.

1. Properly configure and deploy the mobile business objects (MBOs).
   a) Create a Cache Group (or use the default) and set the cache policy to **Scheduled** and set some value for the **Cache interval**, 30 seconds for example.
   b) Create a Synchronization Group and set some value for the **Change detection level**, one minute for example.
   c) Place all Mobile Application project MBOs in the same Cache Group and Synchronization Group.
   d) Deploy the Mobile Application Project as **Replication-based** in the Deployment wizard.

2. Develop the push application.
   a) Generate the Object API code.
   b) Create a new device application in Visual Studio.
   c) Install `SybaseServerSync` on your Windows Mobile device. The cab file can be found in the `<UnwiredPlatform_InstallDir>\ClientAPI\RBS\WM\ServerSync\` folder.
   d) If you have only a single push-enabled application running on the device, write application code that calls the push APIs.

   Note: To run multiple push-enabled applications on a device, see *Developer Guide for Windows and Windows Mobile > Reference < Replication-Based Synchronization APIs > Push Configuration APIs > Running Multiple Push-Enabled Applications on a Device*.

```
private void sampleEnableSIS_Click(object sender, EventArgs e)
  {
    //Set synchronizaton profile properties
    SISsampleDB.GetSynchronizationProfile().ServerName =
"example-xp2";
    SISsampleDB.GetSynchronizationProfile().NetworkProtocol =
"http";
    SISsampleDB.GetSynchronizationProfile().PortNumber = 2480;

    //Set poll interval to 180 seconds.
    SISsampleDB.GetSynchronizationProfile().SISIntervalMS =
180000;

   //Register server sync configuration based on the connection
profile properties.
    //This generates a configuration file for SybaseServerSync
application under \Application Data\
    SISsampleDB.RegisterServerSyncConfiguration();
   //Start SybaseServerSync application. you can also start it
manually before running the SIS sample.
    SISsampleDB.LaunchServerSyncHelper();

    // Login to Unwired Server
    SISsampleDB.LoginToSync("test", "test123");

    //Synchronize the syncgroup.
    SISsampleDB.Synchronize("ofs");

    //Enable SIS on the "ofs" synchronization group.
    ISynchronizationGroup sg =
SISsampleDB.GetSynchronizationGroup("ofs");
    sg.EnableSIS = true;
    sg.Interval = 0;
    sg.Save();

    // Register your callback handler if you want to handle
notifications.
    SISsampleDB.RegisterCallbackHandler(new
MyCallbackHandler());
```

```
    // Start a background thread to do synchronization on
notifications.
    SISsampleDB.StartBackgroundSynchronization();

    //Synchronize the synchronization group to enable SIS on
server.
    SISsampleDB.Synchronize("ofs");
    sg = SISsampleDB.GetSynchronizationGroup("ofs");
    System.Diagnostics.Debug.Assert(sg.EnabledSIS,
"SISSubscription not created");

  }

private void sampleExit(object sender, EventArgs e)
  {
    SISsampleDB.StopBackgroundSynchronization();
    SISsampleDB.ShutdownServerSyncHelper();
  }
```

Running Multiple Push-Enabled Applications on a Device

If you need to run multiple push-enabled applications on a device, you cannot call certain push configurations APIs in your push-enabled applications. Instead you must manually configure and launch the server synchronization tool, `SybaseServerSync`.

You can run multiple push-enabled applications on a single device if the following requirements are met.

- The push-enabled applications must have same connection information (host, port, synchronization protocol, stream parameters, login user, password, and domain), because of the device-level dependency of the `SybaseServerSync` application.
- The push-enabled applications cannot start, stop, or configure the `SybaseServerSync` application in their client code. You must start the `SybaseServerSync` application manually before running the push-enabled applications.

To run multiple applications on a device:

1. Develop the applications as described in *Developer Guide for Windows and Windows Mobile* > *Reference* < *Replication-Based Synchronization APIs* > *Push Configuration APIs* > *Creating a Replication-based Push Application*.
2. Install `SybaseServerSync` on your Windows Mobile device. The cab file can be found in the `<UnwiredPlatform_InstallDir>\ClientAPI\RBS\WM \ServerSync\` folder.
3. Configure `SybaseServerSync` with the host, port, user, and stream parameters.

4. Select the Launch tab and click the start Listener button to launch the `SybaseServerSync` tool.
5. Create your applications by writing the SIS sample code. Refer to the sample code in *Developer Guide for Windows and Windows Mobile* > *Reference* < *Replication-Based Synchronization APIs* > *Push Configuration APIs* > *Creating a Replication-based Push Application*, but ensure that your push-enabled application does not call any of the following APIs from the generated database class:
   - `RegisterServerSyncConfiguration`

- `LaunchServerSyncHelper`
- `ShutdownServerSyncHelper`

Best Practices for Developing Applications

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

- Set up your development environment and develop your application using the procedures in the *Developer Guide for Windows and Windows Mobile*.

Check Network Connection Before Login

If a device does not establish a network connection, the login process does not return a result until after a long timeout occurs. To avoid this delay, check the network connection before performing a login.

Search for *Detect and Verify a Network Connection* on the .NET Framework Developer Center at *http://msdn.microsoft.com/en-us/netframework* for information on verifying connections to network resources required by an application.

Check Connection before Synchronization

Before starting a synchronization, check the connection to the Unwired Server.

```
public static void Synchronize()
{
  if(CheckConnectionStates())
  {
    XXXDB.Synchronize();
  }

   // else notify user or show error message.
}
```

Start a New Thread to Handle Replication-based Synchronization

The synchronization process in replication-based synchronization may take a long time. To avoid blocking the user interface, start a new thread to do background work.

```
public void Synchronize()
{
  if(CheckConnectionStates())
  {
    Thread thread = new Thread(new ThreadStart(this.
SynchronizationCore));
    thread.Start();
  }
   // else notify user or show error message.
}

Private void SynchronizationCore()
```

```
{
  XXXDB.Synchronization(new MYSyncStatusListener);
}
Private class MYSyncStatusListener: SyncStatusListener
{
  // get the status and update the UI
}
```

Constructing Synchronization Parameters

When constructing synchronization parameters to filter rows to be download to a device, if the SQL statement involves two mobile business objects, you must use an "in" clause rather than a "join" clause. Otherwise, when there is a joined SQL statement, all rows of the subsequent mobile business object are filtered out.

For example, you would change this statement:

```
SELECT x.* FROM So_company x ,So_user y where x.company_id =
y.company_id and y.uname='test'
```

To:

```
SELECT x.* FROM So_company x where x.company_id in (select
y.company_id from So_user y where y.uname='test')
```

Clear Synchronization Parameters

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

```
<MBO>SynchronizationParameters params =
<MBO>.SynchronizationParameters;
params.Delete();
params = <MBO>.SynchronizationParameters; //must re-get the sync
parameter instance
params.Param1 = value1; //set new sync parameter value
params.Param2 = value2; //set new sync parameter value
params.Save();
```

Clear the Local Database

Each time you redeploy a package on the Unwired Server, the client application should clear the local database. After clearing the database, login again so that the local database is reconstructed.

```
XXDB.DeleteDatabase();
XXDB.LoginToSync(); //Don't forget to login again so that the local
database will be re-constructed.
```

Turn Off API Logger

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

```
XXDB.GetLogger().SetLogLevel(LogLevel.OFF);
```

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W