Sybase Unwired Platform 1.2
Client Object API Cookbook

DOCUMENT ID: DC00836-01-0120-01

LAST REVISED: March 2009

Copyright © 2009 by Sybase, Inc. All rights reserved.

This publication pertains to Sybase software and to any subsequent release until otherwise indicated in new editions or technical notes. Information in this document is subject to change without notice. The software described herein is furnished under a license agreement, and it may be used or copied only in accordance with the terms of that agreement.

To order additional documents, U.S. and Canadian customers should call Customer Fulfillment at (800) 685-8225, fax (617) 229-9845.

Customers in other countries with a U.S. license agreement may contact Customer Fulfillment via the above fax number. All other international customers should contact their Sybase subsidiary or local distributor. Upgrades are provided only at regularly scheduled software release dates. No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of Sybase, Inc.

Sybase trademarks can be viewed at the Sybase trademarks page at http://www.sybase.com/detail?id=1011207. Sybase and the marks listed are trademarks of Sybase, Inc. ® indicates registration in the United States of America.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Unicode and the Unicode Logo are registered trademarks of Unicode, Inc.

All other company and product names mentioned may be trademarks of the respective companies with which they are associated.

Use, duplication, or disclosure by the government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of DFARS 52.227-7013 for the DOD and as set forth in FAR 52.227-19(a)-(d) for civilian agencies.

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

1. INTRODUCTION AND REQUIREMENTS 5
1.1. Introduction 5
1.2. Requirements 5
1.3. Terminology 5
1.4. Overview 6
1.5. Development Lifecycle 7
1.6. Code Generation from Unwired WorkSpace 8
1.6.1. Partial Class Option 9
1.7. Code Generation API 10
1.8. Client API Dependencies 11
1.8.1. Project Setup 12
1.9. Visual Studio Project Templates 12
2. GENERAL 14
2.1. Generated and Core (Fixed) Class Organization 14
2.2. The Managers 14
2.3. Generic List 15
2.4. API Summary 15
3. MANAGING CONNECTIONS 16
3.1. Creating a Connection 16
3.2. Retrieving a Connection 17
3.3. Database Utilities 17
3.3.1. Dynamic Publication Setting 18
4. SYNCHRONIZATION 18
4.1. Synchronization Parameters 18
4.2. Performing Mobile Business Object (MBO) Synchronization 18
4.2.1. Upload Only 19
4.2.2. Download Only 20
4.3. Performing a File Synchronization 20
4.4. Registering for SMS or HTTP Push Notifications 20
4.5. Synchronization Status Listener 21
4.6. HTTPS protocol synchronization 23
4.7. Synchronization Delays 23
4.8. File MBO (Afaria) Listener 23
5. MOBILE BUSINESS OBJECTS 25
5.1. Retrieving Data from Mobile Business Objects 25
5.1.1. Arbitrary Find 25
5.2. Retrieving Downloaded File MBO Files 27
5.3. Retrieving Relationship Data 28
5.4. Paging Data 29
5.5. MBO Operations 29
5.5.1. Insert Operations 29
5.5.2. Multi-Level Insert (MLI) 29

5.5.3. Inserting New Child MBO Object On Existing MBO Parent Object 30
5.5.4. Update Operation 31
5.5.5. Delete Operation 31
5.6. Original State 32
5.7. Personalization Usage 32
5.8. Parameters and Data Column Types and Sizes 33
5.9. Securing Data 33
5.10. Sync Results and Operation Logs 33
5.11. Date, DateTime, and Time Data Types 34
6. PUSH 35
6.1. Overview 35
6.2. Distribution 35
6.3. Single-Application Use Case (Simplest Option) 35
6.4. Multi-Application Use Case 37
6.5. Limitations of Launching Applications 37
6.6. SMS Push (unsupported) 38
6.7. HTTP Push 39
6.7.1. HttpPushListener 39
6.7.2. PushMessage 41
7. UPGRADE 41
8. SYBASE.UNWIREDPLATFORM.COMMONS ASSEMBLY (UNSUPPORTED) 41
8.1. MobileDevice 41
8.2. NetworkUtilities 42
8.3. Windows.Forms 42
8.3.1. FormsManager 42
8.3.2. StackFormsManager 44
8.3.3. FormsManagerDataObject 46
9. MISCELLANEOUS 46
9.1. SMS Push Notification Messages (unsupported) 46
9.2. Cleaning Up Client Databases in Device Applications 47

1. Introduction and Requirements

1.1. Introduction

This programming cookbook includes a collection of solutions and examples for a variety of tasks and uses of the Sybase Unwired Platform (SUP) .NET and Java Client API.

This cookbook is available from the Sybase Unwired Platform online documentation, as well as on SUP Tech Corner. Check SUP Tech Corner for more current postings:

http://www.sybase.com/developer/library/suptechcorner

Note: The samples labeled "unsupported" are not part of the product, but are useful for demonstration.

1.2. Requirements

- Microsoft Visual Studio .NET 2005 Professional or higher.
- Knowledge of Sybase Unwired Platform concepts and features.
- The Sybase Unwired Platform .NET API assemblies.
- Sybase Unwired Platform code generation API (in Java).
- .NET framework (and compact) 2.0+.

1.3. Terminology

These terms are used throughout the cookbook.

- Remote Client – refers to a user's PDA, Smartphone, or personal computer.
- Remote Database or Client Database – refers to the database on a remote client used to store the data.
- MBO – Mobile Business Objects. The fundamental unit of exchange in Sybase Unwired Platform is the MBO. An MBO corresponds to a data set from a back-end data source. The data can come from a database query, a Web service operation (including Remedy and Siebel), SAP, or a file (using Afaria). An MBO contains both concrete implementation-level details and abstract interface-level details. At the implementation-level, an MBO contains read-only columns that contain metadata about the data in the implementation, and parameters that are passed to the back-end data source. At the interface-level, an MBO contains attributes that map to columns, which correspond to client properties. An MBO may have operations, which can also contain parameters and arguments, and which may be used to create, update, or delete data.

1.4. Overview

Following is an overview of the data access stack from a mobile client.

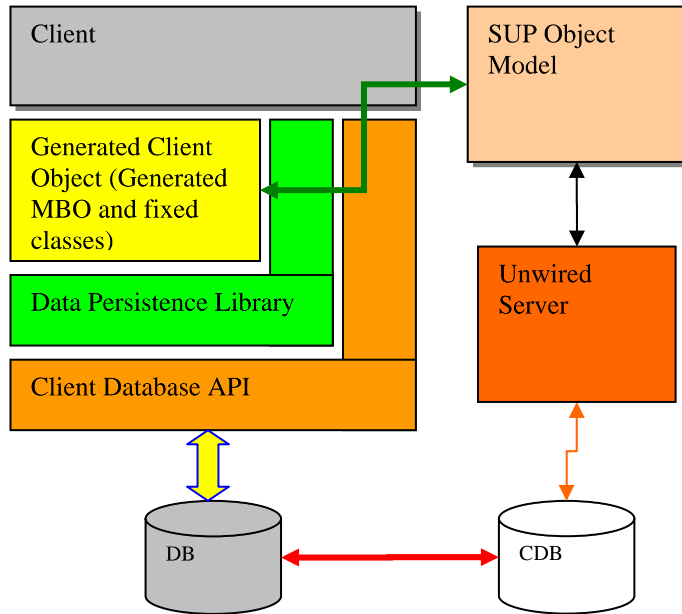


Figure 1. Mobile client data access stack. Key: (1) SUP – Sybase Unwired Platform; (2) MBO – Mobile Business Object; (3) DB – client database; (4) CDB – consolidated database.

The client application has three options as shown above to access data. However, the only supported API is the Generated Client Object API (shown in yellow). The Client Object API uses the Data Persistence Library in its implementation to access and store object data in the database on the device.

The SUP Code Generation API generates the Client Object API. Following is a high-level view of the code generation process (Unwired WorkSpace refers to the SUP development environment – either Eclipse Edition or Visual Studio Edition):

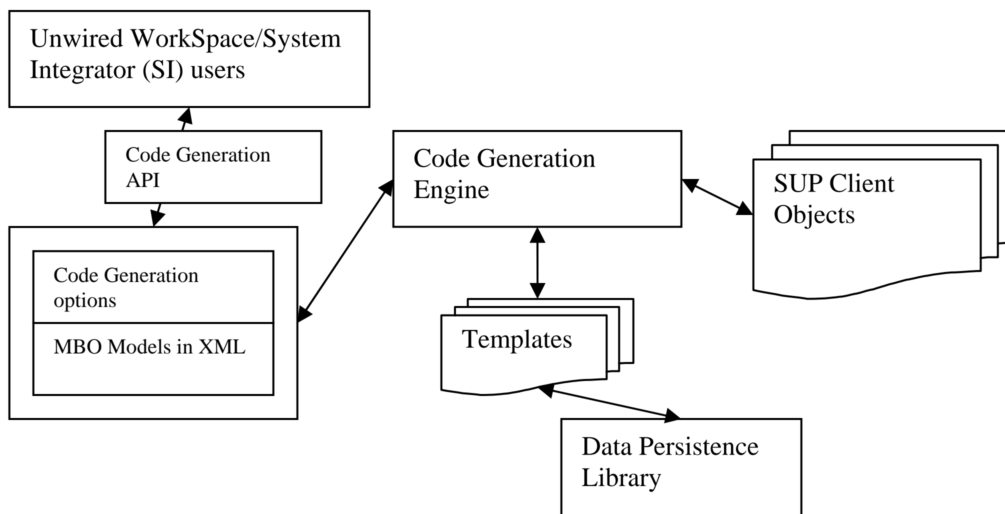
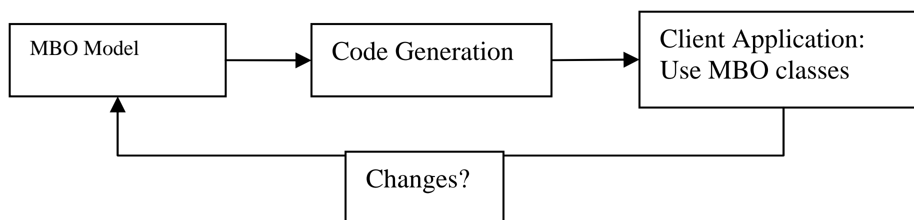


Figure 2. Client Object API is generated by SUP Code generation.

- Unwired WorkSpace in Eclipse Edition or Visual Studio Edition builds the mobile business object (MBO), or the system integrator creates scripts or manually writes the deployment unit.
- Unwired WorkSpace calls the Code Generation API and provides MBO models and code generation objects as inputs. The Code Generation API executes the SUP Code Generation Engine with the given inputs.
- SUP Code Generation Engine applies the correct templates based on the given code generation options and the MBO model.
- SUP Code Generation Engine outputs the Client Objects.

1.5. Development Lifecycle



A typical development cycle involves:

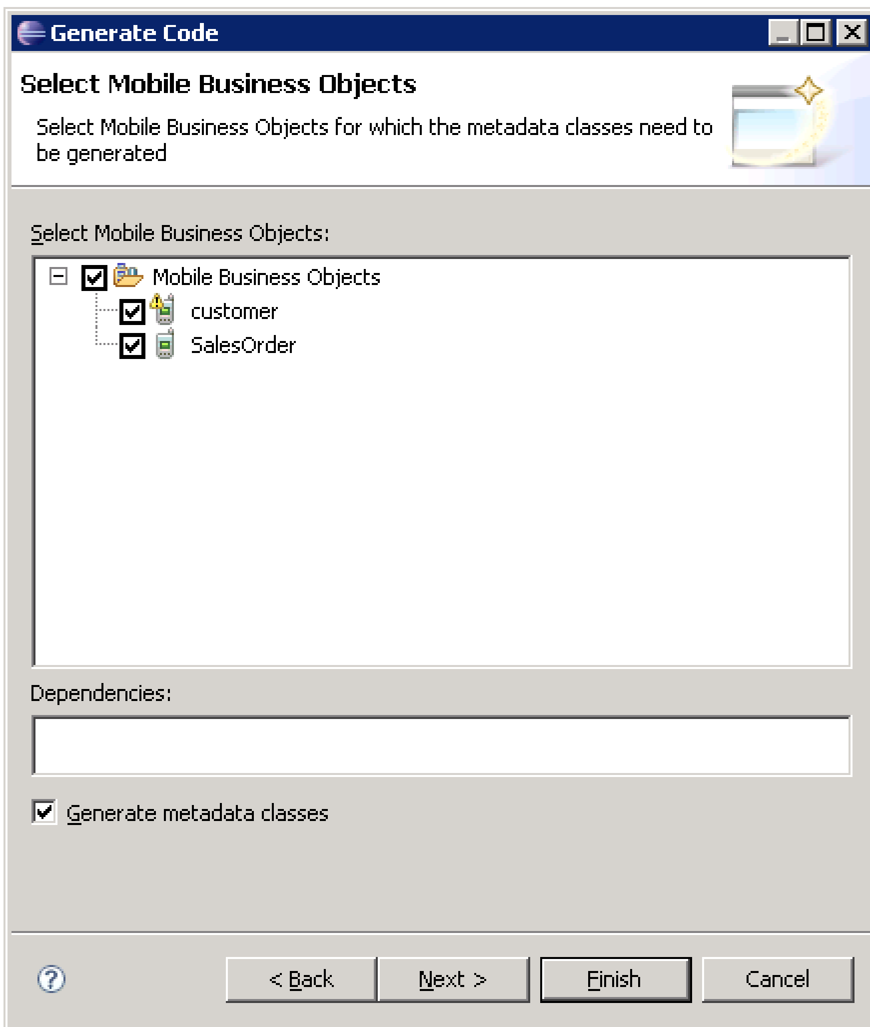
1. Design and build or modify MBO model in Unwired WorkSpace (Eclipse or Visual Studio Editions).
2. Generate MBO classes.

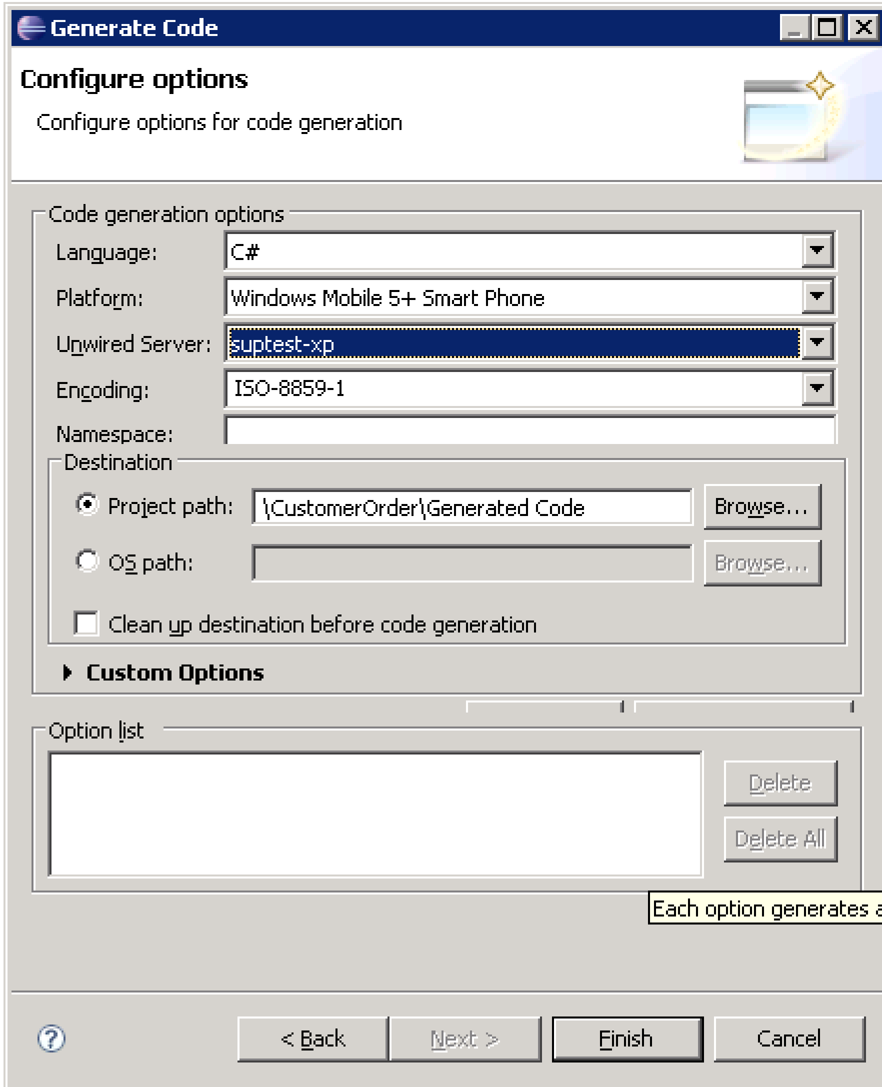
3. Apply MBO classes in client application project.
4. Repeat steps 1 to 3 on MBO changes.

Note: The best practice is to plan and design the MBO model as accurately as possible before building client applications to avoid having to redesign client applications later. The Client Object API is not affected by how the MBO binds to the enterprise information system (EIS).

1.6. Code Generation from Unwired WorkSpace

Once the Mobile Development project is completed as desired, users can right click on the project from the GUI, select Generate Code, and follow the instructions in the dialog to generate the client object code.





Note that for C#, Unwired WorkSpace generates non-UI Compact Framework 2.0 compatible code regardless of what platform is selected. The generated code works with Compact Framework 2.0+ and full .NET Framework 2.0+.

1.6.1. Partial Class Option

Partial classes mean that your class definition can be split into multiple physical files. Logically, partial classes do not make any difference to the compiler. During compile time, it simply groups all the various partial classes and treats them as a single entity. This option allows custom code to be preserved. The generated code will add a "partial" key word in the MBO classes saved in `<MBO Name>.cs`.

```
//Customer.cs
public partial class Customer
{
```

```
…
}
//CustomerCustom.cs
//custom code to preserve for Customer MBO
public partial class Customer
{
    public void foo()
    {
    }
}
```

In the example above, custom code in `CustomCustomer.cs` will be preserved no matter how many times `Customer.cs` is regenerated.

1.7. Code Generation API

The Client Object API can be generated from the Java code generation API instead of Unwired WorkSpace if MBO deployment is available. The MBO deployment unit can be extracted from the Mobile Development Project.

```
import com.sybase.sup.codegen.client.CodeGenerator;
import com.sybase.sup.codegen.client.impl.*;
import java.util.HashMap;
import java.io.*;
public class CGMain
{
    public static void main(String[] args)
    {
        CodeGenerator cg = new SUPCodeGenerator();
        HashMap <String, String>cgOptions = new HashMap <String, String>();
        cgOptions.put(CodeGenerator.OPTION_META_DATA, CodeGenerator.META_DATA_NO);
        cgOptions.put(CodeGenerator.OPTION_PACKAGE_NAME, "Sybase.Test");
        cgOptions.put(CodeGenerator.OPTION_LANGUAGE, CodeGenerator.LANGUAGE_CS);
        cgOptions.put(CodeGenerator.PLATFORM_WM_SP, CodeGenerator.PLATFORM_WM_SP);
        cgOptions.put(CodeGenerator.OPTION_META_DATA, CodeGenerator.META_DATA_YES);
        cg.generateCode (new File ("deployment_unit.xml"), "C:/tmp/SampleClientUsage",
cgOptions);
    }
}
```

1.8. Client API Dependencies

The client API assembly DLL dependencies are installed under the Sybase Unwired Platform installation directory (`<installation_directory>\UnwiredPlatform-1_2\Servers\UnwiredServer\clientAPI`).

The contents of the client API directory are:

- `Afaria` – Afaria assembly dependencies.
- `apidocs` – Code Generation API Javadocs.
- `CodeGenModel` – code generation model templates for .NET and Java.
- `dpl\dotnet\API\bin` – .NET Data Persistence Library and client database (UltraLite) assemblies.
- `fixed` – sources of the fixed classes that are used by the generated classes, and .NET documentation in XML format.
- `lib` – binaries of Code Generation API and fixed classes for .NET and Java (RIM, Java SE 5).
- `dpl\dotnet\API\src` – source code of the HTTP Push Listener included in the Sybase.UnwiredPlatform.Networking namespace. This source is included so developers can create their own custom HTTP Push Listener if needed.
- `templates` - Visual Studio Unwired WorkSpace project templates for Win32, Windows Mobile Professional, and Standard editions.

The `dpl\dotnet\API\bin` folder contains the following directory structure:

- `dpl` – the Sybase Unwired Platform Data Persistence Library DLLs as individual files.
  - `ce` – files for use on Windows CE based systems.
  - `win32` – files for use on full Windows based systems like Windows XP.
- `ultralite` – the client database assembly DLLs.
  - `ce` - files for use on Windows CE based systems like Windows Mobile 5+.
  - `win32` - files for use on full Windows based systems like Windows XP.

The assemblies listed above support Compact Framework (CF) 2.0+ on Visual Studio 2005 and 2008.

Visual Studio Version	Project Type
Visual Studio 2005 (and 2008)	Full .NET Framework 2.0+ Application
	Windows CE .NET CF 2.0+ Application
	Pocket PC .NET CF 2.0+ Application
	Smartphone .NET CF 2.0+ Application

1.8.1. Project Setup

Add these as references in the Visual Studio project:

- `Sybase.UnwiredPlatform.Data.UltraLite.dll`
- `iAnywhere.Data.UltraLite.dll`
- `iAnywhere.Data.UltraLite.resources.dll` (several languages are supported)
- `Sybase.Persistence.dll`

If File MBO is used, add:

Win32 client:

- `Interop.XeClientLib.dll`
- `Sybase.UnwiredPlatform.Data.Afaria.dll`

Afaria client needs to be installed (see the *Sybase Unwired Platform Installation Guide* for information).

Windows Mobile:

- `Sybase.UnwiredPlatform.Data.Afaria.dll`
- `XeManagedClient.dll`

Add these as items in the Visual Studio project, and set “Build Action” to “Content” and “Copy to Output Director” to “Copy always”.

- `ulnet10.dll`
- `mlcrsa.dll (if HTTPS protocol is used)`

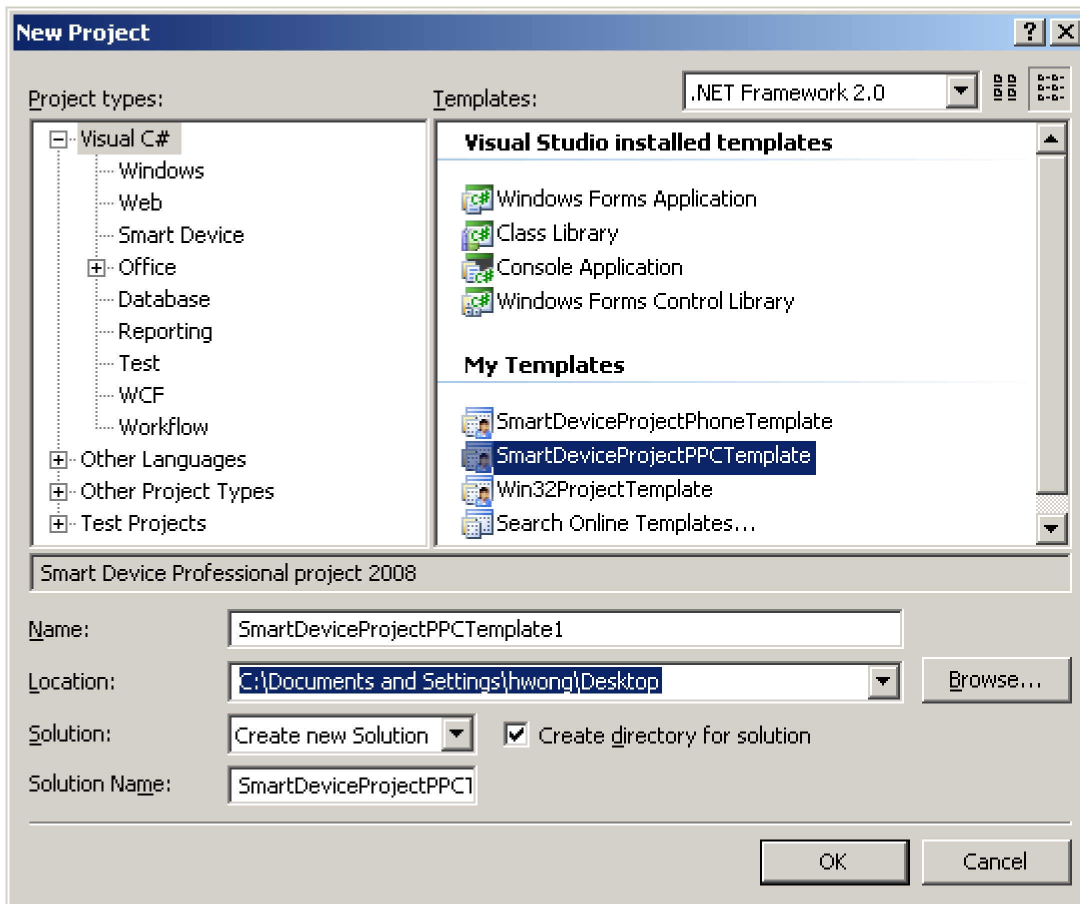
If File MBO is used, add:

Windows Mobile:

- `ClientAPIShim.dll`

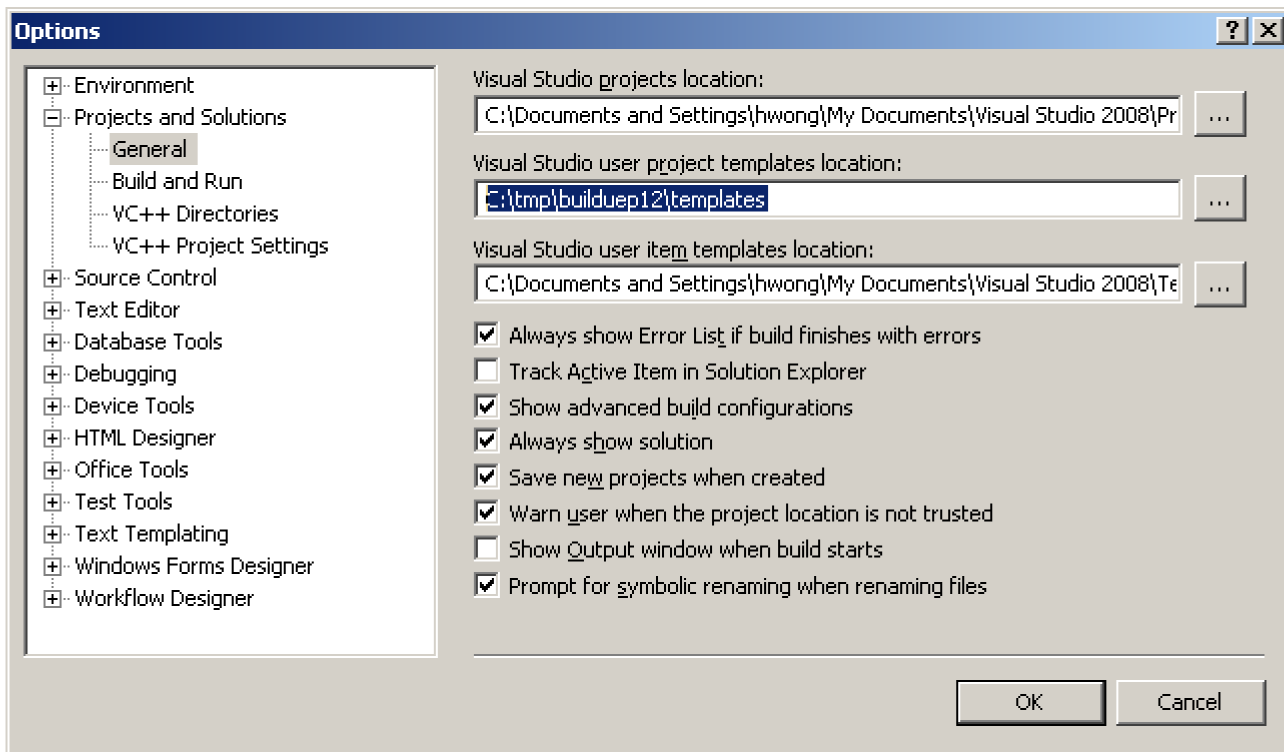
1.9. Visual Studio Project Templates

The client `templates` folder contains Visual Project 2008 project templates. Use these templates to start a project in the Visual Studio “new project” wizard with all the required assemblies referenced. Only the generated code is not included in these templates.



- SmartDeviceProjectPhoneTemplate – Windows Mobile Professional template
- SmartDeviceProjectPPCTemplate – Windows Mobile Standard template
- Win32ProjectTemplate – Windows 32 template

To see this in the new project wizard, make sure “Visual Studio user project template location” is set correctly:



2. General

2.1. Generated and Core (Fixed) Class Organization

Any classes for the client object API that are fixed and do not change are part of the "core" classes (also called fixed classes). All other classes are generated classes. Core classes are provided in a package/namespace that is consistent with a language's conventions.

- .NET – Sybase.Persistence

Note: Generated classes do not generate "using" statements due to issues with ambiguity. Fully qualified namespace are used in generated classes.

2.2. The Managers

The ConnectionManager and SynchronizationManager classes provide access to their respective portions of the Sybase.Persistence assembly. These classes follow the Singleton design pattern and are referenced using a static Instance property. Below are samples on how to acquire each of the manager classes.

```
ConnectionManager cm = ConnectionManager.Instance;
SynchronizationManager syncManager = SynchronizationManager.Instance;
```

2.3. Generic List

Any part of the API that returns a generic `List<Object>` actually returns a clone of the original collection. This is useful when there is a need to delete an item while iterating over the collection to prevent the .NET Framework from throwing an `InvalidOperationException`.

2.4. API Summary

Following is a summary of the API described in this document. See .NET documentation for all classes in Sybase.Persistence.

Object	Type	Description
Sybase.Persistence	namespace	
Connection	class	Store connection information such as connection name, host name of the Unwired Server, the Unwired Server port, the username and password.
ConnectionManager	class	Provides a single access point for storing and retrieving a number of connection profiles from the remote client database.
SynchronizationManager	class	Singleton to handle all MBO synchronization.
SynchronizationParameters	class	Sets synchronization parameters used by SynchronizationManager.
SyncResult	class	Stores the last sync result information for read, update, insert and delete operations.
SyncStatusListener	interface	The listener interface for receiving synchronization progress events
Query	Class	Used for arbitrary searches in Find method.
PushConfiguration	Class	Stores push configuration information.

Additional information is provided in the JavaDocs or system documentation provided with Sybase Unwired WorkSpace:

- .NET docs for Sybase.Persistence (CHM and online HTML) – `<installation_directory>\UnwiredPlatform-`*`x_x`*`\Servers\UnwiredServer\clientAPI\fixed\dotnet\docs`

- Code Generator (online HTML) – `<installation_directory>\UnwiredPlatform-x_x\Servers\UnwiredServer\clientAPI\\apidocs`
- Data Filter (online HTML) – `<installation _directory>\UnwiredPlatform-x_x\Servers\UnwiredServer\tomcat\webapps\onepage\docs\javadoc`

3. Managing Connections

3.1. Creating a Connection

Before synchronizing and getting MBO data, first set up a connection.

Connection related classes in the Sybase.Persistence namespace:

- Connection – a Connection is used to store information such as the host name of the Unwired Server, the Unwired Server port, the username and password. The Connection object also has a Name property that can be used to assign a convenient name to the Connection. Since the remote client database supports multiple Connections, the Name property must be unique.

- ConnectionManager – the ConnectionManager provides a single access point for storing and retrieving and number of profiles from the remote client database.

This creates a Connection to an Unwired Server:

```
Connection c = new Connection();
c.MobiLinkHost = "sup.sybase.com";
c.MobiLinkPort = 2439;
c.MobiLinkStreamType = MobiLinkStreamType.Http;
c.Name = "supAdmin";
c.Package = "Customer_1.0.0";
c.UserName = "supAdmin";
c.Password = "supAdmin";
c.Save();
//or
//ConnectionManager.Instance.SaveConnection(c);
```

Once the connection is set up and saved, it can be set as the default Connection.

```
ConnectionManager.Instance.DefaultConnection = c;
```

MBO classes use the default Connection to sync against the Unwired Server if `MBO.setConnection()` is not set.

3.2. Retrieving a Connection

To work with a Connection that has already been saved, retrieve the Connection from the ConnectionManager.

```
Connection conn = ConnectionManager.Instance.GetConnection("MyProfile");
```

Attempting to retrieve a Connection that has not been saved or does not exist will result in a NULL value being returned. This can be combined with the Connection creation code to check for a default Connection, and, if a default Connection does not exist, then create and save it.

```
Connection c= ConnectionManager.Instance.GetConnection("MyProfile");
if(c == null)
{
    c.MobiLinkHost = "sup.sybase.com";
    c.MobiLinkPort = 2439;
    c.MobiLinkStreamType = MobiLinkStreamType.Http;
    c.Name = "supAdmin";
    c.Package = "Customer_1.0.0";
    c.UserName = "supAdmin";
    c.Password = "supAdmin";
    c.Save();
}
```

3.3. Database Utilities

The underlying storage is a database. The cache size and page size properties affect the performance of the client application. These properties can be set in the ConnectionManager.

```
// An integer specifying the page size in bytes.
// Valid values are 1024 (1K), 2048 (2K), 4096 (4K), 8192 (8K), 16384 (16K).
// The default is 4096.
ConnectionManager.Instance.PageSize = 4096;

//The database cache specifies how much memory is set aside for use as a cache for
//database operations. The cache size is specified in bytes. To specify the cache size
//in kilobytes the suffix "k" or "K" can be used. To specify the cache size in megabytes
//the suffix "m" or "M" can be used.
```

```
// If the cache size is not set or is improperly set then the default size of 64
//kilobytes is used.
ConnectionManager.Instance.CacheSize = 512k
```

3.3.1. Dynamic Publication Setting

For each syncable MBO, the client object API creates a publication for synchronization purpose. If there are more than 30 syncable MBOs in a package, dynamic publication must be turned on to avoid synchronization errors:

```
ConnectionManager.Instance.UseDynamicPublication = true;
```

This must be set before any MBO synchronization occurs.

4. Synchronization

4.1. Synchronization Parameters

Synchronization parameters let you change the parameters used to retrieve data from an MBO during a synchronization session. Their primary purpose is to partition data. By changing the synchronization parameters, you will affect the data with which you are working (this includes searches), and synchronization.

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

4.2. Performing Mobile Business Object (MBO) Synchronization

In order to perform MBO synchronization, you must save a Connection object. See Creating a Connection for more information on how to create and save a Connection. Additionally, you may want to set some synchronization parameters via SynchronizationParameters for SynchronizationManager.

```
Connection conn = ConnectionManager.GetConnection (“myconn”);
SynchronizationParameters sp = new SynchronizationParameters()
sp.Connection = conn;
sp.DoPlayback = true;
sp.MBONames = new string [] {“Customer”};
SynchronizationManager.Instance.SynchronizeApplications(sp);
```

This syncs a Customer MBO using the default Connection using the convenient Synchronize method within the MBO:

```
SynchronizationManager.Instance.SynchronizeApplication("Customer");
//or
Customer.Synchronize()
```

This syncs a Customer MBO using a specified Connection:

```
Connection conn = ConnectionManager.GetConnection ("myconn");
SynchronizationManager.Instance.SynchronizeApplication(conn, "Customer");
//or
Customer.SetConnection (conn);
Customer.Synchronize()
```

It is possible to synchronize multiple MBOs in one synchronization session. However, caution should be taken since performing simultaneous synchronizations on several very large Unwired Server applications can impact server performance; thus, possibly effecting other remote users as well. The following code sample demonstrates how to synchronize multiple MBOs in one synchronization session.

```
SynchronizationManager.Instance.SynchronizeApplications(conn, new string[] {"Customer",
"SalesOrder"});
//Or
SynchronizationParameters sp = new SynchronizationParameters()
sp.Connection = conn;
sp.DoPlayback = true;
sp.MBONames = new string [] {"Customer", "SalesOrder", "Parts"};
SynchronizationManager.Instance.SynchronizeApplications(sp);
```

4.2.1. Upload Only

Setting `SynchronizationParameters.UploadOnly = true`, only operations are submitted to the server. No data will be downloaded to the client.

Note: Use this with caution. With upload only set to true, the client API will still be able to access the operations until download occurs. As a result, the MBO instances referencing any updated, inserted, or delete operations will still maintain their states.

4.2.2. Download Only

Setting `SynchronizationParameters.DownloadOnly = true`, no data will be uploaded and client will only get data downloaded from the server.

4.3. Performing a File Synchronization

Synchronizing a file MBO is the same as synchronizing regular MBOs except that the file content is saved in a physical location on the client.

```
//download
UserPDF pdf = new UserPDF();
pdf.SetConnection (fileconn);
pdf.Synchronize(true, new String[] {"myfile.pdf"});
string path = pdf.GetFile()

//upload
UserPDF.SaveForUpload ("myfile.pdf");
UserPDF.Upload()
```

Note: The Synchronize method handles either upload or download:

- First Parameter: true => download / false => upload.
- Second Parameter: used for specifying name(s)/pattern of file(s) in case of download e.g. "UserManual.pdf", "a*.pdf" etc. This parameter can contain array of names of individual files also. This will download subset of files from available files on server. Subset query is not applicable for upload operation. Also, only one download/upload operation is allowed per MBO.

4.4. Registering for SMS or HTTP Push Notifications

During an MBO synchronization, a custom .NET client can inform the Unwired Server to send push notifications when the MBO data changes. To register for push notifications an optional `PushConfiguration` object is supplied to the SynchronizationManager Synchronize method. The `PushConfiguration` object contains information such as the phone number of the mobile device for SMS Push, or an IP address, or the hostname of the computer for HTTP Push; an optional security token; and a pass back value that is returned to the remote client when a push notification message is sent.

```
PushConfiguration push = new PushConfiguration();
push.Address = "+14157778888";
push.DeviceId = "myuniqueid";
push.Protocol = "UASMS";
push.PushRegistration = "Y";
push.SecurityToken = "1234";
push.MobileBusinessObject="myclient";
```

```
SynchronizationManager.Instance.SynchronizeApplication("Customer", push, true);
```

Valid options for the Protocol property include "Default-DeviceTracker " (default), "UASMS" (unsupported), "UAHTTP", and "UAMAIL" (unsupported).

Due to limitations of network connectivity in General Packet Radio Service (GPRS)-capable Windows Mobile devices, the use of IP number tracking and HTTP push notifications is not a viable solution. When used on Windows Mobile devices, the Client Object API only supports using the "UASMS" protocol flag. Similarly, due to limitations in Win32 systems the Client Object API only supports using the "UAHTTP" protocol flag when used on Win32 systems.

When using HTTP Push, the Address property of the PushConfiguration object must be in the following format: `http://<HOSTNAME_OR_IP>:<PORT>;mode=direct`

`<HOSTNAME_OR_IP>` is the host name or IP address of the client registering for push notifications and `<PORT>` is the port on which the push notifications will be sent. The ";mode=direct" suffix must appear at the end of the hostname/IP address and port. This additional piece of information is needed to instruct Unwired Server that the HTTP Push message is being sent directly to a remote client, and is required for HTTP Push to work.

The use of e-mail-based push and the "UAMAIL" flag is intended for the BlackBerry client.

4.5. Synchronization Status Listener

A synchronization status listener can be implemented to track the progress of running synchronization in a custom .NET client. To track synchronization status changes in a custom client, create a new class that implements the SyncStatusListener interface and pass an instance of that class to the SynchronizationManager when a new synchronization is started.

```
MySyncListener listener = new MySyncListener(); //implements SyncStatusListener
SynchronizationManager.Instance.SynchronizeApplication("Customer", listener);
```

As the application synchronization progresses, the `ApplicationSyncStatus` method defined by the SyncStatusListener interface is called and is passed an `ApplicationSyncStatusData` object. The `ApplicationSyncStatusData` object contains information about the MBO being synchronized, the Connection to which it is related, and the current state of the synchronization process. By testing the State property of the `ApplicationSyncStatusData` object and comparing it to the possible values in the `SyncStatusState` enumeration a custom client can react accordingly to the state of the synchronization. Possible uses include changing form elements on the client screen to show the progress of the sync, such as showing a green image when the synchronization is in progress, a red image if the synchronization fails, and a gray image when the synchronization has completed successfully and disconnected from the server.

Note that the `ApplicationSyncStatus` method of the SyncStatusListener are called and executed in the data synchronization thread. If a custom client runs synchronizations in a thread that is not the primary user interface thread then the custom client will not be able to update the client screen as the status changes. In this case, a custom client needs to use `Control.Invoke`, or a similar technique, to instruct the primary user interface thread to update the screen regarding the current synchronization status.

For example, create a class called SyncListener. Implement SyncStatusListener interface. In `ApplicationSyncStatus` method, create switch to catch the state during synchronization.

```
//Create a class call SyncListener
//add to your .net project

using System;
using Sybase.Persistence;

namespace YourNameSpaceHere
{
    public class SyncListener: SyncStatusListener
    {
        #region ISyncStatusListener Members

        public bool
        ApplicationSyncStatus(Sybase.UnwiredPlatform.Data.ApplicationSyncStatusData data)
        {
            switch (data.State)
            {
                case Sybase.UnwiredPlatform.Data.SyncStatusState.ApplicationSyncDone:
                    //implement your own UI indicator bar
                    break;
                case Sybase.UnwiredPlatform.Data.SyncStatusState.MetaSyncDone:
                    //implement your own UI indicator bar

                    break;
                case Sybase.UnwiredPlatform.Data.SyncStatusState.MetaSyncError:
                    //implement your own UI indicator bar
                    break;
                case Sybase.UnwiredPlatform.Data.SyncStatusState.ApplicationSyncError:
                    //implement your own UI indicator bar
                    break;
                case Sybase.UnwiredPlatform.Data.SyncStatusState.SyncDone:
                    //implement your own UI indicator bar
                    break;
                case Sybase.UnwiredPlatform.Data.SyncStatusState.SyncStarting:
                    //implement your own UI indicator bar
                    break;
```

```
            }
            return (false);
        }
        #endregion
    }
}
```

To synchronize with the listener, create a SyncListener instance.

```
SyncListener syncListener = new SyncListener();
Connection conn = ConnectionManager.GetConnection ("myconn");
SynchronizationManager.Instance.SynchronizeApplication(conn, "Customer", syncListener);
```

4.6. HTTPS protocol synchronization

If HTTPS protocol is used (`Connection.MobiLinkStreamType = MobiLinkStreamType.Https`), make sure that the Unwired Server or the Relay Server is set up to listen on HTTPS and the certificate is set up correctly. See the Unwired Server HTTPS configuration for setup instructions.

```
In the Connection object, use HTTPS as the protocol and set the stream property to
include:
 trusted_certificates=\full_path_to_yourPubliccertfile.crt
Note: make sure mlcrsa.dll is in the same location as ulnet11.dll
```

4.7. Synchronization Delays

To prevent concurrent synchronization errors in the Unwired Server because Unwired Server uses time-based synchronization technology, client applications need to introduce at least a one second delay between syncs. Even when the client completes the sync request, the Unwired Server might still be processing the end synchronization phase. For example, if the client application performs synchronization in a loop, adding a delay before the next sync request will ensure that the Unwired Server completes the previous sync request. Depending on the Unwired Server load, if concurrent access errors appear in Unwired Server logs, the client application may increase the delay.

4.8. File MBO (Afaria) Listener

The File MBO listener can be used to track the progress of File MBO syncs and log specific activities that happen during sync. For example, implement the `AfariaConnectionListener` interface to track the events:

```
public class MyAfariaConnListener : AfariaConnectionListener
```

```
    {
        private Message Msg = Message.Instance;
        public void MessageReceived(MessageEventArgs args)
        {
            Msg.Log("MessageReceived(): " + args.TimeStamp + " " + args.Message + " " +
args.MessageString);
        }
        public void ProgressReceived(int progress)
        {
            Msg.Log("ProgressReceived(): " + progress);
        }
        public void StatusReceived(StatusEventArgs args)
        {
            Msg.Log("StatusReceived(): " + args.TimeStamp + " " + args.Status + " " +
args.StatusString);
        }
    }
```

Then set the listener in the AfariaSynchronizer instance. This requires accessing the Data Persistence Library profile to accomplish this:

```
Connection conn = ConnectionManager.DefaultConnection;
Sybase.UnwiredPlatform.Data.Profile profile =
Sybase.UnwiredPlatform.Data.ProfileManager.Instance.GetProfile(conn.Name);
Sybase.UnwiredPlatform.Data.Afaria.AfariaSynchronizer afc;
afc = Sybase.UnwiredPlatform.Data.Afaria.AfariaSynchronizer.Instance;
afc.Profile = profile;
afc.setConnectionListener(new MyAfariaConnListener());
```

Any File MBO syncs using the above connection will be tracked.

5. Mobile Business Objects

5.1. Retrieving Data from Mobile Business Objects

To retrieve data from a Mobile Business Object (MBO), use one of the static `FindBy` methods in the MBO class. Named `FindBy` methods are defined in the Unwired WorkSpace developer environment by the modeler and result in the generation of `FindBy` methods with whatever parameters and return type were defined in Unwired WorkSpace. `FindBy` methods return a collection of objects that match the specified search criteria defined in the named query.

This retrieves a Customer by Id (Unwired WorkSpace defines Id as the **only** `FindBy` which is similar to a primary key in a database).

```
Customer customer = Customer.FindById(101);
```

If Unwired WorkSpace defines more than one `FindBy` attribute and we are using the `FindById` to get the customers, the Customer MBO returns a list.

```
List<Customer> customers = Customer.FindById(101);
```

Also in this case where there are more than one `FindBy` attributes defined, a single Customer object is returned when calling the `FindBy`.

```
Customer customers = Customer.FindBy(101, "Sybase");
```

Note that the `FindBy` method is always generated and returns a List of the MBO object regardless of the number of `FindBy` attributes defined.

To get all the Customers, call the `FindAll` method.

```
List<Customer> customers = Customer.FindAll();
```

Note: After an MBO is synchronized, it is recommended to refresh the MBO instance via one of the Find methods.

5.1.1. Arbitrary Find

The Arbitrary search is meant for custom device applications that need the ability to build queries on the fly based on user input. In addition to allowing for arbitrary search criteria, the arbitrary find method also lets the user specify a desired ordering of the results and object state criteria.

For simplicity and future proofing, a Query class is included in the client object API's core classes. The Query class is the single object that is passed to the arbitrary search methods and is composed of search conditions, object/row state filter conditions, and data ordering information.

For example, consider you wish to locate all Customer objects that meet the following 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
//assume the variable "camd" is a CustomerAttributeMetaData retrieved earlier
//Note that camd is not required for arbitrary find operation, but just
//for example purpose. Users can
//pass in a string if they know the attribute name. R1 column name = attribute name.

CompositeTest innerCompTest = new CompositeTest();
innerCompTest.CompositionType = TestType.OR;
innerCompTest.add (new AttributeTest ("state", "CA", TestType.EQUALS);
innerCompTest.add (new AttributeTest ("state", "NY", TestType.EQUALS);

CompositeTest outerCompTest = new CompositeTest();
outerCompTest.CompositionType(TestType.AND);
outerCompTest.addFilter(new AttributeTest("FirstName", "John", TestType.EQUALS));
outerCompTest.addFilter(new AttributeTest("LastName", "Doe", TestType.EQUALS));
outerCompTest.addFilter(innerCompTest);

//define the ordering
SortOrderCollection sort = new SortOrderCollection();
sort.add ("LastName", SortOrderType.Ascending);
sort.add(new SortOrder(camd.getFirstName().getName(), SortOrderType.Ascending));
sort.add(new SortOrder(camd.getCredit().getName(), SortOrderType.Descending));

//define the ordering
SortOrderCollection sort = new SortOrderCollection();
sort.add(new SortOrder(camd.getLastName().getName(), SortOrderType.Ascending));

//set the Query object
props.TestCriteria = outerCompTest;
props.ObjectState = ObjectState.INSERTED | ObjectState.UPDATED;
props.SortOrderCollection = sort;
props.Skip =10;
props.Take = 5;
```

```
List<Customer> customers = Customer.Find(props);
```

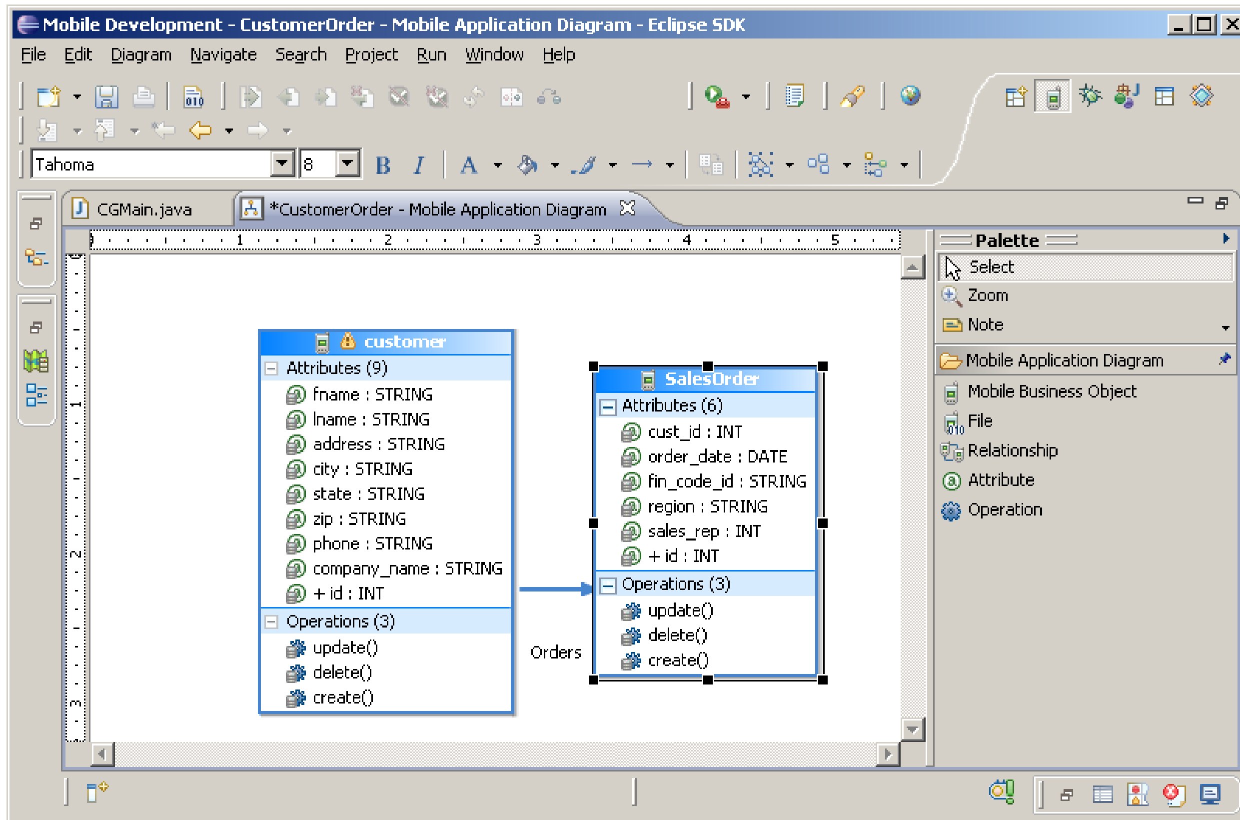
5.2. Retrieving Downloaded File MBO Files

After a File MBO download synchronization, the files are stored in the client device under the download location.

```
UserPDF pdf = new UserPDF;
pdf.SetConnection (fileconn);
pdf.Synchronize (true, "*.pdf");

string downloadpath = pdf.DownloadPath;

//this gets the list of files in the download path
string [] files = pdf.GetFiles();

//or a file filtered for this File MBO
string file = pdf.GetFile();
```

5.3. Retrieving Relationship Data

If there is a relationship between two MBOs, the parent MBO can access the associated MBO.



To illustrate, 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 from Customers to Orders on the customer id column. The Orders application is parameterized to return order information for the customer id.

```
Customer customer = Customer.FindBy (101);
List<SalesOrder> orders = customer.Orders;
```

To access filtered associated data, use the MBO <relationship name>FilterBy (Query query).

```
Customer customer = Customer.FindBy (101);
Query query = new Query();
…
List<SalesOrder> orders = customer.OrdersFindBy (query);
```

5.4. Paging Data

On low memory devices retrieving 30,000 records from the database will likely cause the custom client to fail and throw an `OutOfMemoryException`. Consider using the Query object to limit the result set.

5.5. MBO Operations

Mobile Business Object operations are performed on the 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 CUD 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 convenience `Save` method can be automatically generated which, when called internally, determines if it should insert or update data to the local client side database. This generated save method is colloquially referred to as the "magic save" method. Under other situations, where there are multiple instances of create and/or update operations, it is not possible to automatically and smartly generate a magic `Save` method.

5.5.1. Insert Operations

To execute insert operations on an MBO, create a new instance of the MBO, set the MBO attributes and then either call `save()` or insert the operation name.

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

If a custom client needs to create a second customer record the custom client must create a new instance of the MBO and call its save method.

5.5.2. Multi-Level Insert (MLI)

Multi-level insert allows a single synchronization to execute a chain of related insert operations. Consider creating a Customer and a new Customer order at the same time on the client side, where the SalesOrder has a reference to the new Customer identifier. The following example demonstrates a multi-level insert:

```
Customer customer = new Customer();
customer.Fname = "supAdmin";
customer.Lname = "s3pAdmin";
```

```
customer.Phone = "777-8888";
SalesOrder order = new SalesOrder();
order.Cust_id = customer.Id;
order.Order_date = DateTime.Now;
order.Region = "Eastern";
order.Sales_rep = 102;
//now add the new SalesOrder to the new Customer
//Note: If there are more than 2 levels, you can add to the Order's relationship list
customer.Orders.Add (order);
//Only the parent MBO needs to call Save()
customer.Save();
```

In the above example, MBO primary keys do not need to be set because they are generated by the MBO operation. This assumes that the child MBO has only one parent relationship. Multi-parent relationships are not supported. See the Sybase Unwired Platform online documentation for specific multi-level insert requirements.

5.5.3. Inserting New Child MBO Object On Existing MBO Parent Object

Any MBO can be in a relationship and can have multiple parents. The generated MBO constructor doesn't allow a parameter to identify the parent. To support this use case, use the generated `Add<relationship name><child mbo operation name>(<child MBO> mbo)` method:

```
Customer customer = Customer.FindBy (101);
SalesOrder order = new SalesOrder();
order.Cust_id = customer.Id;
order.Order_date = DateTime.UtcNow;
order.Region = "Eastern";
order.Sales_rep = 102;
order.Id = 1001;
//don't call order.Save();
customer.AddSalesOrderCreate(order);
Customer.Synchronize();
```

Another option is to find the parent MBO Id, assign it to the new child MBO object, and then sync the child MBO:

```
Customer customer = Customer.FindBy (101);
SalesOrder order = new SalesOrder();
order.Cust_id = customer.Id;
order.Order_date = DateTime.UtcNow;
order.Region = "Eastern";
order.Sales_rep = 102;
```

```
order.Id = 1001;
order.Save();

SalesOrder.Synchronize();
//to get the latest customers and orders
Customer.Synchronize();
```

In this case, the SalesOrder MBO needs to be syncable, and MBO primary key attributes need to be set manually.

5.5.4. Update Operation

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

```
Customer cust = Customer.FindBy (101);
cust.Fname = "supAdmin";
cust.Company_name = "Sybase";
cust.Phone = "777-8888";
cust.Save();
```

From a relationship, an update can be done like this:

```
Customer cust = Customer.FindBy (101);
List<SalesOrder> orders = cust.Orders;
SalesOrder order = orders[0];
order.Order_date = DateTime.Now;
order.Save();
```

Note: If the MBO attributes and update operation parameters are not modified and `Save()` or the update method is called, a `SUPException` is thrown. Only modified attributes and operation parameters are allowed to execute the update operation.

5.5.5. Delete Operation

To execute delete operations on an MBO, get an instance of the MBO, set the MBO attributes, and then call the delete operation name.

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

From a relationship, a delete can be done like this:

```
Customer cust = Customer.FindBy (101);
List<SalesOrder> orders = cust.Orders;
SalesOrder order = orders[0];
order.Delete();
```

5.6. Original State

Before a client synchronizes a called instance operation, the original data values may be retrieved by calling a special property that returns the object as it was before the change(s). After synchronization is completed successfully, the special original value property is cleared and set to NULL.

```
Customer cust = Customer.FindBy (101);
cust.Fname = "supAdmin";
cust.Company_name = "Sybase";
cust.Phone = "777-8888";
cust.Save();
Customer org = cust.OriginalState;
```

5.7. Personalization Usage

The Personalization class is generated automatically with the rest of the MBOs in the package. This MBO is saved under the "system" package; therefore, a Connection instance to the system package is required to use the Personalization MBO.

Note: Personalization MBO is used for setting personalization values for all personalization keys used in any package.

```
Personalization.SetConnection (systemConn);
Personalization.Synchronize();
List<Personalization> per = Personalization.FindAll();
for (int i = 0; i < per.Count; i++)
{
    if (per[i].KeyName == ''state'')
    {
        per[i].KeyValue = ''NY'';
        per[i].Save();
        break;
    }
}
Personalization.Synchronize();
```

If a synchronization parameter is personalized, this can be overwritten.

```
CustomerSynchronizationParameters.SetOverridePersonalization(new string[] { "region" });
syncparam.Region = "West";
```

5.8. Parameters and Data Column Types and Sizes

By default, the Unwired Server stores all columns as `char(300)` columns if the max length is not specified in the MBO model. This default can be customized by changing the Sybase Unwired Platform property. Edit `sup.properties` and change the `string.maxlen=300` to a size that works with your application.

Each attribute and parameter string data type can be adjusted to the proper length. This is recommended to reduce the size of the client database. This can be set in the Unwired WorkSpace development environment before deploying the MBOs to Unwired Server.

5.9. Securing Data

By default, client databases are unencrypted on disk and in permanent memory when they are created. Text and binary columns are plainly readable within the database file when using a viewing tool such as a hex editor. All client database files created by the API can be encrypted to provide security against skilled and determined attempts to gain access to the data, but has a significant performance impact. Note that once the encryption key is set, you cannot change the encryption key unless the UDB file is deleted and recreated. Also, the API does not store the encryption key anywhere. It is up to the client application to provide the encryption key to the API when needed.

The `localdb.udb` stores Connection information. To encrypt this database, set the encryption key using `ConnectionManager.LocalEncryptionKey`.

Note: The `ConnectionManager.LocalEncryptionKey` is not persisted. Once this is set, client applications needs to set this again or the client API will not be able to connect to the `localdb.udb`.

The API creates UDB files based on a connection's MobiLink host, MobiLink port number, and package values (`<MobiLinkHost.MobiLinkPort.Package>.udb`). Other connections with the same host name, port number, and package share the same UDB file. To encrypt these UDB files, set an encryption key using the Connection object's `EncyptionKey` property before the UDB file is created. UDB files are created when a connection is opened to the database for the first time.

Note: If the Connection instance is retrieved from ConnectionManager, the Connection.EncryptionKey property needs to be set again.

5.10. Sync Results and Operation Logs

Sybase.Persistence.SyncResult holds sync results for read, update, insert and delete operations. This can be accessed from each generated MBO. The data in SyncResult gets updated after each MBO sync.

This example gets the Sybase.Persistence.SyncResult and gets the read sync information:

```
Sybase.Persistence.SyncResult result =  Customer.SyncResult();
```

```
//get the read playback information
result.ErrorMessage;
result.LastGoodPlayback;
result.LastPlayback;
result.BadPlaybackCount;
```

This example gets the OperationLog for Customer MBO:

```
List<Sybase.Persistence.OperationLog> logs = result.GetOperationLog();
foreach (Sybase.Persistence.OperationLog log in logs)
{
   log.Message;
   log.OperationName;
   log.Status;
}
```

The OperationLog.Status properties have two possible values:

- `MSGS_MAPP_SYNC_<UPDATE|INSERT|DELETE>_OK` – operation executed successfully
- `MSGS_MAPP_SYNC_<UPDATE|INSERT|DELETE>_FAILED` – operation execution has an error. In this case, check the OperationLog.Message to get the details.

5.11. Date, DateTime, and Time Data Types

Datetime, time, and date data types are set in UTC time zone in Unwired Server; however, the date and time fields are maintained when they are taken from enterprise information system (EIS), such as DateTime in databases, with no time zone information. For EIS, such as Web Services, with time zone information in DateTime data types, the Unwired Server converts them to UTC time zone. The Generated Object API returns DateTime object with the date time fields maintained in `DateTimeKind.Unspecified` kind. It is up to the client application to make sense of the DateTime object returned by the API.

```
//show a date from Database EIS
List<SalesOrder> orders = customer.Orders;
foreach (SalesOrder order in orders)
{
    DateTime orderDate = order.Order_date;
    MessageBox.Show(orderDate.ToString());
}

//update date information
List<SalesOrder> orders = customer.Orders;
foreach (SalesOrder order in orders)
{
```

```
    DateTime orderDate = DateTime.Parse ("9/12/2008 5:30 PM");
}

//Convert a date to local time assuming the Date we get back is in UTC from Web Services
CaseList mycase = CaseList.FindBy("HD10001");
DateTime caseDate = mycase.CaseDate;
caseDate = new DateTime(caseDate.Ticks, DateTimeKind.Utc);
caseDate = caseDate.ToLocalTime();

//set a UTC date for EIS after parsing it in local time
CaseList newcase = new CaseList();
newcase.CaseDate = DateTime.Parse("10/1/2008 3:00 PM").ToUniversalTime();
```

6. Push

6.1. Overview

Prior to version 1.2, Sybase Unwired Platform supported the HTTP listener method of push sync on Windows platforms, which requires a client to listen for HTTP POST requests containing any push notifications. This method cannot function very well (if at all) in a sometimes-connected, highly-managed environment normally found with mobile devices.

Unwired Platform 1.2 introduces a new push sync alternative for both Windows and Windows CE/Mobile platforms and .NET clients. This is based on the SQL Anywhere `dblsn` tool, which interfaces directly with the Unwired Server and uses directed UDP for push notifications.

6.2. Distribution

The components needed for `dblsn`-based push sync are packaged in the .NET `API-bin.zip` generated with the .NET client APIs. These are currently laid out in the `<installation_directory>\Sybase\UnwiredPlatform-1_2\Servers\UnwiredServer\clientAPI`.

- The Win32 components are in: `<installation_directory>\UnwiredPlatform-1_2\Servers\UnwiredServer\clientAPI\dpl\dotnet\API\bin\dpl\win32\v2`.
- The CE/WindowsMobile components are in: `<installation_directoyr>\UnwiredPlatform-1_2\Servers\UnwiredServer\clientAPI\dpl\dotnet\API\bin\dpl\ce\v2`.

6.3. Single-Application Use Case (Simplest Option)

The simplest use case is for a single application on a device or workstation receiving push notification. Follow these instructions:

1. Portions, or all, of this step, may be handled automatically by the client-code generator. If so, please skip this step and go to step #2.

   In your application code, make sure the PushConfiguration element of the SynchronizationParameters is set up to request push notification:

```
p.PushConfiguration = new PushConfiguration();
p.PushConfiguration.PushRegistration = "Y";
p.PushConfiguration.ApplicationName = "My Application";
```

   The application name above should be unique on the device (for example, there should be no other applications on the same device with the same name) and can optionally be a full-path to the executable that should be launched when a push request is received.

   A recommended but optional step is to specify this field, which automates the registration of Unwired Server connection details to the `SUPSyncConfig.xml` configuration file.

```
p.PushConfiguration.RegisterListenerConfiguration = true;
```

   In addition, you must set up a push-sync listener in your application. This defines what will be done in your application when a push request is inbound and it also sets up a thread which will monitor for push requests.

   Create a class that implements the ServerSyncListener interface and register an instance of the class with DPL using a call to:

```
mgr.RegisterServerSyncListener(listener, "My Application");
```

The second argument should match the application name you used when you registered for push configuration in the first part of this step.

2. Deploy the external sync components (`dblsn.exe`, `supsis.exe`, `supsis_sync.exe`, `supsis_listener.exe` and required DLL files) to your application deployment directory (not the Sybase installation directory like early versions).
3. Launch your application and perform a sync.
4. Launch the `supsis.exe` tool from your application directory and verify that the Unwired Server hostname, username are set correctly.

   Make sure the “Application Directory” is correctly set to the directory in which `dblsn.exe`, and `supsis_sync.exe` are located.

You can optionally enable "Launch Client Application" from the UI, which will cause an inbound sync request to launch a new instance of the client application.

5. Finally, launch the push-sync listener. There are three ways to do this:
   - From your client application, call the following API:

```
SynchronizationManager.LaunchServerSyncHelper();
```

   - From the `supsis.exe` UI, click the "Launch Listener" button.
   - Launch the `supsis_listener.exe`.

After performing these steps, your application is ready to receive push sync requests. Monitor your application directory for a file named `SUPSyncNotify.da`, which will include push requests received and processed by `dblsn.exe` and `supsis_sync.exe`.

Any errors will be logged to the `SUPSyncNotify.log` file.

6.4. Multi-Application Use Case

This use case is for a situation where multiple applications on the same device need sync notification. In this case, you must use a single installation of dblsn.exe, supsis.exe, et al. Perform the following steps:

1. If necessary, perform step #1 from the preceding Single-Application Use Case.
2. Deploy `dblsn.exe`, `supsis.exe`, et al to a common location such as `%PROGRAM FILES%\Sybase`.
3. Run `supsis.exe` from this location and configure all settings appropriately. All applications on the device must share a common Unwired Server username and server. Verify that the application directory correctly points to the shared location.
4. Copy the `SUPSyncConfig.xml` from the common directory to each application directory. This tells each application where to find `dblsn.exe`, et al along with the common sync notification data file.
5. Perform step #5 from the single-application use case. Only one listener can/should be started for the device.

6.5. Limitations of Launching Applications

On Windows Mobile platform, most applications are created as single-instance applications and attempting to launch a second instance will simply activate the first instance. Our launching logic takes advantage of this and simply launches a new instance each time. If your application does not behave like this then you will see multiple application instances as a result of this launching behavior and you should not use the automatic launching capability.

On the Windows (Win32) platform, most applications are created as multi-instance applications and attempting to launch a second instance will actually create a second instance. The launching logic will attempt to locate an existing instance of the application by calculating the process name of the application based on the executable (For example, application name of `C:\Program Files\MyApp\MyApplication.exe` will normally have a process name of `MyApplication`). If a process already exists with the specified name, then the launcher will not launch a second copy. If the process does not exist, we will launch the application. This logic is simplistic and may not work under all circumstances such as when multiple executables have the same name but are in different directories.

In order to maintain consistency and because there is some potential for problems, particularly on Win32, application launching is disabled by default but can be enabled using the `supsis.exe` UI or a simple change to the `SUPSyncConfig.xml`.

6.6. SMS Push (unsupported)

Custom clients can implement an SMS listener to receive SMS push messages from Unwired Server to synchronize MBOs.

For example:

```
//Your form must include namespace
using Microsoft.WindowsMobile.PocketOutlook.MessageInterception;

//Your reference must reference
Microsoft.WindowsMobile
Microsoft.WindowsMobile.PocketOutlook dll


//Register for sms reciever on your Form's Load method
private void FormXYZ_Load(object sender, EventArgs e)
        {
            Program.NetworkUtilities.MessageInterceptor = new
MessageInterceptor(InterceptionAction.NotifyAndDelete, true);
            Microsoft.WindowsMobile.PocketOutlook.MessageInterceptor.MessageCondition =
new MessageCondition(MessageProperty.Body, MessagePropertyComparisonType.Contains,
"mbo=customer";
            Microsoft.WindowsMobile.PocketOutlook.MessageInterception.MessageReceived +=
new MessageInterceptorEventHandler(SmsPushMessageReceived);
        }

        public void SmsPushMessageReceived(object sender, MessageInterceptorEventArgs e)
        {
            //this method will be called if any sms push happen
            Sybase.Persistence.Connection conn =
Sybase.Persistence.ConnectionManager.Instance.DefaultConnection();
            if (profile != null)
```

```
            {
                //Create your own auto login
                this.AutoLogin();
            }
        }
```

6.7. HTTP Push

HTTP Push is only supported on remote clients with network connections that allow inbound traffic and have a consistent IP address or hostname. Cellular providers typically do not allow inbound traffic to devices on their networks. Additionally, IP addresses for GPRS connections are usually pooled and change frequently as GPRS connections are established and dropped. Therefore, HTTP Push is neither a practical nor viable push method for cellular based network connections. This leaves HTTP Push best suited for 802.X LAN or WLAN connected clients.

6.7.1. HttpPushListener

A simple HTTP Push listener class is included in the Sybase.UnwiredPlatform.Networking namespace. The source code for the listener is included in the client installation directory for customization purposes. When an HTTP Push message is received, a PushMessage object is passed to the client application hosting the HTTP Push listener.

The following code demonstrates how to register for HTTP Push.

```
//Register for HTTP Push notifications.
PushConfiguration push = new PushConfiguration();
push.Address = "http://my_pc_hostname.mydomain.com:25000;mode=direct";
push.DeviceId = "myuniqueid";
push.Protocol = "UAHTTP";
push.PushRegistration = "Y";
push.SecurityToken = "1234";
push.MobileBusinessObject="myclient";

SynchronizationManager.Instance.SynchronizeApplication("Customer", push, true);
```

The following code shows how to instantiate and start an HttpPushListener.

```
//Create an HttpPushListener instance, listen on all network connections on port 25000
HttpPushListener httpPushListener = new HttpPushListener(System.Net.IPAddress.Any, 25000);
//assign an event handler for the HttpPushMessageReceived event
httpPushListener.HttpPushMessageReceived += new
HttpPushMessageReceivedEventHandler(HttpPushMessageReceivedHandlerProc);
//assign an event handler for the HttpPushMessageError event
```

```
httpPushListener.HttpPushMessageError += new
HttpPushMessageErrorEventHandler(HttpPushMessageErrorHandlerProc);
//start the HttpPushListener
httpPushListener.Start();
```

Here is a sample implementation of the HttpPushListener event handlers. Both of these event handler methods are called by the HttpPushListener's own processing thread. Therefore, these event handler methods are not called in the main user interface thread of a client application. If user interface changes need to take place in these event handler methods the client application may hang or throw an exception. Therefore, be sure to use proper invocation techniques, such as the `Control.Invoke` method, so the user interface change takes place in the main user interface thread.

```
public void HttpPushMessageReceivedHandlerProc(object sender,
HttpPushMessageReceivedEventArgs e)
{
    //push notification properties can be accessed via e.PushMessage
    //make sure the push message is not empty, this means the entire push
    //message was received and make sure it is meant for this client application
    if(!e.PushMessage.IsEmpty && e.PushMessage.ApplicationName == "MyApp")
    {
        //this is a valid push message, initiate a synchronization
    }
}

public void HttpPushMessageErrorHandlerProc(object sender, HttpPushMessageErrorEventArgs
e)
{
    //There was an error in processing the received push message
    //Any exception information can be accessed via e.Exception
}
```

The following code demonstrates how to stop and tear down the HttpPushListener.

```
//unassign the HttpPushMessageReceived handler method
httpPushListener.HttpPushMessageReceived -= new
HttpPushMessageReceivedEventHandler(HttpPushMessageReceivedHandlerProc);
//unassign the HttpPushMessageError handler method
httpPushListener.HttpPushMessageError -= new
HttpPushMessageErrorEventHandler(HttpPushMessageErrorHandlerProc);
//stop the HttpPushListener thread
httpPushListener.Stop();
```

6.7.2. *PushMessage*

The `PushMessage` class contains the following properties.

```
req_id=<autoincrement integer>;op=sync;wid=<windowID>;profile=<profileID>;app=<client
device application name>;mbo=<MBO name>
```

- `ApplicationName` – the "app" section of the push message.
- `Operation` – the "op" section of the push message.
- `ProfileId` – the "profile" section of the push message.
- `RequestId` – the "req_id" section of the push message.
- `SecurityToken` – the "token" section of the push message (not used by Unwired Server).
- `WindowId` – the "wid" section of the push message.

For more details on the meanings of the sections of a push message refer to SMS Push Notification Messages.

7. Upgrade

To upgrade clients to a new version of software:

1. Update all SUP client assemblies to 1.2, and recompile the client application. See Client API Dependencies for the list of assemblies to update.
2. Delete the old client database files. See Cleaning Up Client Databases in Device Applications.

8. Sybase.UnwiredPlatform.Commons Assembly (unsupported)

8.1. MobileDevice

The `MobileDevice` class is provided as is and is only intended for use on physical Windows Mobile PocketPCs and Windows Mobile Smartphones. The behavior and results on emulators is not consistent and varies on different emulator versions. Using the `MobileDevice` class on anything other then actual Windows Mobile PocketPCs or Windows Mobile Smartphones is not supported.

```
//Get the unique device ID for the Windows Mobile device.
//MobileDevice.DeviceId can be used on non-telephone Windows Mobile PocketPCs as well.
```

```
string deviceId = MobileDevice.DeviceId;

//Get the device phone number, the device must be a phone.
string phone = MobileDevice.PhoneNumber;

//Get the device IMEI, the device must be a phone.
string imei = MobileDevice.Imei
```

8.2. NetworkUtilities

The `NetworkUtilities.IsNetworkConnectionActive` property returns a Boolean true if there is an IP address other then the loopback 127.0.0.1 address assigned to the device. Having an IP address other then 127.0.0.1 means the device has an open network connection. However, having a network connection does not necessarily mean the internet or a particular server is currently accessible. This is similar to having a desktop PC connected to an internal company network but having the external gateway disabled. While the desktop PC has an IP address the internet and external servers can not be accessed. The `IsNetworkConnectionActive` property is intended to serve as a quick test to see if a network connection has been established. It is not intended to determine if a particular remote computer is accessible. Only an attempt to connect to a remote computer can determine if the remote computer is accessible or not.

8.3. Windows.Forms

The Sybase.UnwiredPlatform.Windows.Forms namespace provides a set of utility classes for managing inter-form navigation and communication. Two different form manager classes are included. They are named FormsManager and StackFormsManager. Both form manager classes support caching forms to minimize the amount of device memory used by multiple screen applications.

8.3.1. FormsManager

The FormsManager can be thought of as a forward-only form navigation/communication manager, since it does not track the history of form navigation steps. The forward-only nature of the FormsManager class makes it ideal for use in large multi-form applications that do not have a structured form navigation hierarchy. In order for a form that is managed by the FormsManager class to receive cross-form communication messages as a form navigation step takes place, the form must implement the `IFormsManagerForm` interface. By implementing the `IFormsManagerForm` interface the forms class will implement a method named `PassDataForward`. The `PassDataForward` method is called when the form that is being navigated to by the FormsManager implements the `IFormsManagerForm` interface.

```
//Create a new FormsManager, this should be stored in a globally accessible variable
//since it is needed by all forms in the client application.
FormsManager formsManager = new FormsManager();
//Set the first form the FormsManager will manage
formsManager.FirstForm = formsManager.GetForm(typeof(FormCustomers));
```

```
//Use the first form with the Application.Run method in the Main function of the client.
Application.Run(formsManager.FirstForm);
//Because the first time that FormCustomers is loaded and shown must be handled by the
//Aplication.Run method the PassDataForward method of FormCustomers will not be called.
//Instead, the FormLoad event should be handled and used to signal that the first form has
//been loaded and shown. Subsequent navigations to FormCustomers from another form in the
//client will trigger a call to FormCustomer's PassDataForward function.
```

After a globally accessible FormsManager has been created and the first form is shown a custom client can navigate to other forms using the FormsManager `ShowForm` method. See the API documentation for information regarding the several overloaded versions of the `ShowForm` method.

```
//Create a new FormsManagerDataObject to pass a collection of data to the next form.
FormsManagerDataObject data = new FormsManagerDataObject();
//Set the key VALUE1 to be "ABC"
data["VALUE1"] = "ABC";
//Set the key "VALUE2" to be 42
data["VALUE2"] = 42;
//Tell the FormsManager to navigate to a form class named Form2 and pass along the
//FormsManagerDataObject data to the next form.
//The form specified in the second parameter is the form that the FormsManager will hide
//as the navigation to the next form is made. Since we are leaving the current form the
//second parameter is set to the 'this' keyword.
formsManager.ShowForm(typeof(Form2), this, data);
```

During form navigation, the FormsManager hides the form specified in the second parameter of the `ShowForm` method. This prevents end users from using the Running Programs section of the Memory Settings screen to incorrectly return to a previous form before the application logic allows.

In addition to the `PassDataForward` method, the `IFormsManagerForm` interface requires that a `CloseForm` method be implemented as well. The `CloseForm` method is intended to be called by the FormsManager when the form is being closed and removed from the FormsManager cache. The `CloseForm` method can also be called manually from custom client code if a custom client uses a specialized shutdown and clean up method.

8.3.2. StackFormsManager

The StackFormsManager form manager class is similar to the `FormsManager` class in that it manages cross-form navigation/communication and caches previously instantiated forms. The difference between StackFormsManager and FormsManager is that StackFormsManager maintains a form navigation history by keeping a stack containing information about the previously visited forms. By using the stack information, the StackFormsManager allows a custom .NET client to navigate backward through the history of previously visited forms. The StackFormsManager is ideal for use in large multi-form applications that do have a structured form navigation hierarchy tree.

In order for a form that is managed by the StackFormsManager class to receive cross-form communication messages as a form navigation step takes place, the form must implement the `IStackFormsManagerForm` interface. The `IStackFormsManagerForm` class implements the `PassDataForward` and `PassDataBackward` methods. The `PassDataForward` method is called when forward form navigation uses the StackFormsManager. Conversely, the `PassDataBackward` method is called when backward form navigation is made using StackFormsManager history.

```
//Create a new StackFormsManager, this should be stored in a globally accessible variable
//since it is needed by all forms in the client application.
StackFormsManager stackFormsManager = new StackFormsManager();
//Set the first form the StackFormsManager will manage
stackFormsManager.FirstForm = stackFormsManager.GetForm(typeof(FormCustomers));
//Use the first form with the Application.Run method in the Main function of the client.
Application.Run(stackFormsManager.FirstForm);
//Because the first time that FormCustomers is loaded and shown must be handled by the
//Aplication.Run method the PassDataForward method of FormCustomers will not be called.
//Instead, the FormLoad event should be handled and used to signal that the first form has
//been loaded and shown. Subsequent forward navigations to FormCustomers from another form
//in the client will trigger a call to FormCustomer's PassDataForward method and
//backward navigations to FormCustomers from another form will trigger a call to
//FormCustomer's PassDataBackward method.
```

After a globally accessible StackFormsManager has been created and the first form is shown, a custom client can navigate to other forms using the StackFormsManager `ShowForm` method. See the API documentation for information regarding the several overloaded versions of the `ShowForm` method.

```
//Create a new FormsManagerDataObject to pass a collection of data to the next form.
FormsManagerDataObject data = new FormsManagerDataObject();
//Set the key VALUE1 to be "ABC"
data["VALUE1"] = "ABC";
//Set the key "VALUE2" to be 42
data["VALUE2"] = 42;
//Tell the StackFormsManager to navigate to a form class named Form2 and pass along the
//FormsManagerDataObject data to the next form.
//The form specified in the second parameter is the form that the FormsManager will hide
```

```
//as the navigation to the next form is made. Since we are leaving the current form the
//second parameter is set to the 'this' keyword.
stackFormsManager.ShowForm(typeof(Form2), this, data);
```

From the above sample the custom client can add code to the `Form2` class to navigate backward to the `FormCustomers` class by using the following code.

```
//Hide Form2 (do not unload it from the StackFormsManager's cache of forms or from memory)
//and navigate back to the parent form in the StackFormsManager's history. This will
//result in returning to FormCustomers.
//And optional FormsManagerDataObject can be sent to FormCustomers by supplying one to
//the HideForm method.
stackFormsManager.HideForm();
```

Using `HideForm` leaves `Form2` in the StackFormsManager form cache. This speeds up subsequent navigations to `Form2` because the form is already instantiated and in the cache.

If a custom client needs to navigate back a form and wants to close the current form and remove it from the cache in order free up some memory, then the following alternative code can be used.

```
//Close Form2 (unload it from the StackFormsManager's cache of forms and from memory)
//and navigate back to the parent form in the StackFormsManager's history. This will
//result in returning to FormCustomers.
//And optional FormsManagerDataObject can be sent to FormCustomers by supplying one to
//the HideForm method.
stackFormsManager.CloseForm();
```

Calling either `HideForm` or `CloseForm` will result in a call to the `PassDataBackward` method of the previous form in the StackFormsManager history if the previous form implements the `IStackFormsManagerForm` interface.

During forward form navigation, the StackFormsManager hides the form specified in the second parameter of the `ShowForm` method. This prevents end users from using the Running Programs section of the Memory Settings screen to incorrectly return to a previous form before the application's logic allows them to.

In addition to the `PassDataForward` and `PassDataBackward` methods, the `IStackFormsManagerForm` interface requires that a `CloseForm` method be implemented as well. The `CloseForm` method is intended to be called by the StackFormsManager when the form is being closed and removed from the StackFormsManager cache. The `CloseForm` method can also be called manually from a custom client's code if a custom client uses a specialized shutdown and clean up method.

8.3.3. FormsManagerDataObject

The FormsManagerDataObject is a collection class that contains a set of key/value pairs. The FormsManagerDataObject provides a quick and simple way to pass any number of values of any type from one form to another during form navigation when using the FormsManager or StackFormsManager classes.

To test if a `FormsManagerDataObject` contains a value for a specific key the following code can be used.

```
//Assume the FormsManagerDataObject is a variable named 'data'.
//Check for a key named CompanyID, if it is not found it will return 'null'.
if(data["CompanyID"] != null)
{
    MessageBox.Show(data["CompanyID"].ToString());
}
else
{
    MessageBox.Show("Company ID not provided.");
}
```

The key of the key/value pairs is a string object and the value of the key/value is returned as an object. Therefore, any object passed from one form to another using the `FormsManagerDataObject` must be cast to its true type after it is extracted from the `FormsManagerDataObject` before it can be used.

```
//Cast the ArrayList that is extracted from the FormsManagerDataObject before it
//can be used
ArrayList list = (ArrayList) data["CustomerList"];
```

The keys of the `FormsManagerDataObject` must be unique. If a custom client sets a specific key/value pair twice then the first key/value pair is replaced with the second pair.

9. Miscellaneous

9.1. SMS Push Notification Messages (unsupported)

Unwired Platform SMS push notification messages are structured in the following format:

```
req_id=<autoincrement integer>;op=sync;wid=<windowID>;profile=<profileID>;app=<client
device application name>;mbo=<MBO name>
```

The SMS push notification messages are designed to not exceed the length limitation of SMS messages. The individual fields of the SMS message are as follows:

- `req_id` – a unique identifier integer value for the push notification that was sent from Unwired Server.
- `op` – the operation to perform. Potential values for this field are "sync" and "delete". An `op` value of "sync" means the client should synchronize. An op value of "delete" means a registration for push notifications regarding a particular application has been deleted from the Unwired Server. See the `wid` field for additional information. How and if a custom client supports reacting to the "delete" operation often depends on the feature specifications of the custom client and is left to developer of the custom client to process.
- `profile` – a unique identifier for the profile associated with the application involved with the push notification message. For custom .NET clients, the profile field is a GUID for the Profile associated with the application involved with the push notification. A custom .NET client can iterate over all available Profile objects, and test the Profile object's GUID Id property against the profile field to locate the specific Profile.
- `wid` – an identifier integer that represents the application ID of the MBO involved with the push notification message. If the `op` field is "sync" then `wid` is the application ID of the application the push notification is for. If the `op` field is "delete" then the `wid` is the application ID of the application for which the custom client is no longer registered to receive push notifications.
- `token` – the optional security token that was passed to the server when a custom client registers for push notifications via the Data API.
- `app` – the flag keyword that was specified and passed to the server when a custom client registers for push notifications via the Data API. If a custom client submits the value "MyPush" in the `ApplicationName` property of the `PushSyncInfo` object, then the app field will be "MyPush" when a push notification for the associated application is sent.
- `mbo` – the MBO name that can be used when calling the SynchronizationManager.

9.2. Cleaning Up Client Databases in Device Applications

It is possible to clean up a custom client and return it to a freshly installed state without having to uninstall and reinstall the entire client application. This is also useful for SUP client API upgrades. To do this, delete all `<MobiLinkHost.MobiLinkPort.Package>.udb` files from the client's installation folder on the device. Once the custom client is started and any necessary Profile is saved, the client will be ready to begin synchronizing with the Unwired Server again. No data downloaded will be lost if the `*.udb` files are deleted because it will be downloaded again when the freshly cleaned client is synchronized once again. The only data loss that can occur is for pending Operations. Pending Operations that are not synchronized with the consolidated database before cleaning and resetting the client application will be lost.

The Sybase.UnwiredPlatform.Data.DatabaseUtilities has a handle method to cleanup all databases:

```
Sybase.UnwiredPlatform.Data.DatabaseUtilities.DeleteAllDatabases();
```