



**Developer Guide: Windows and Windows  
Mobile Object API Applications**

---

**SAP Mobile Platform 2.3 SP03**

DOCUMENT ID: DC01925-01-0233-01

LAST REVISED: September 2013

Copyright © 2013 by Sybase, Inc. All rights reserved.

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

Upgrades are provided only at regularly scheduled software release dates. No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of Sybase, Inc.

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

SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world.

Java and all Java-based marks are trademarks or registered trademarks of Oracle and/or its affiliates in the U.S. and other countries.

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

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

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

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

# Contents

<b>Getting Started with Windows and Windows Mobile</b>	
<b>Development</b> .....	<b>1</b>
Object API Applications .....	1
Best Uses for Object API Applications .....	2
Cache Synchronization .....	2
Client Runtime Architecture .....	3
Documentation Roadmap for SAP Mobile Platform .....	4
<b>Development Task Flow for Object API Applications</b> .....	<b>5</b>
Installing the Windows Mobile Development Environment .....	6
Configuring Windows Mobile Device Center .....	6
Enabling Network Access from the Windows Mobile Device Emulator .....	6
Installing Microsoft Synchronization Software .....	6
Installing X.509 Certificates on Windows Mobile Devices and Emulators .....	7
Generating C# Object API Code .....	8
Generating C# Object API Code Using SAP Mobile WorkSpace .....	8
Generating Object API Code Using the Code Generation Utility .....	11
Generated Code Location and Contents .....	12
Validating Generated Code .....	12
Creating a Project .....	13
Downloading the Latest Afaria Libraries .....	13
Creating a Project in Visual Studio .....	13
Rebuilding the Generated Solution in Visual Studio .....	14
Generating Online Help for Visual Studio .....	15
Integrating Help into a Project .....	15
Client Application Dependencies .....	16

Adding References to a Mobile Application Project .....	19
<b>Development Task Flow for DOE-based Object API</b>	
<b>Applications .....</b>	<b>21</b>
Installing the Windows Mobile Development Environment .....	22
Configuring Windows Mobile Device Center .....	22
Enabling Network Access from the Windows Mobile Device Emulator .....	22
Installing Microsoft Synchronization Software .....	22
Installing X.509 Certificates on Windows Mobile Devices and Emulators .....	23
Generating C# Object API Code .....	24
Generated Code Location and Contents .....	24
Creating a Project .....	25
Downloading the Latest Afaria Libraries .....	25
Creating a Project in Visual Studio .....	25
Rebuilding the Generated Solution in Visual Studio .....	26
Generating Online Help for Visual Studio .....	27
Integrating Help into a Project .....	28
Client Application Dependencies .....	28
Adding References to a Mobile Application Project .....	30
<b>Developing the Application Using the Object API .....</b>	<b>33</b>
Initializing an Application .....	33
Initially Starting an Application .....	33
Subsequently Starting an Application .....	46
Accessing MBO Data .....	47
Object Queries .....	47
Dynamic Queries .....	47
MBOs with Complex Types .....	48
Relationships .....	49
Manipulating Data .....	50

Creating, Updating, and Deleting MBO Records .....	50
Other Operations .....	51
Using SubmitPending and SubmitPendingOperations .....	51
Shutting Down the Application .....	53
Closing Connections .....	53
Uninstalling the Application .....	53
Deleting the Database and Unregistering the Application .....	53
Recovering From SAP Mobile Server Failures .....	53
<b>Testing Applications .....</b>	<b>61</b>
Testing an Application Using a Simulator .....	61
Client-Side Debugging .....	61
Server-Side Debugging .....	63
Improve Synchronization Performance by Reducing the Log Record Size .....	64
Determining the Log Record Size .....	64
Reducing the Log Record Size .....	67
<b>Localizing Applications .....</b>	<b>69</b>
Generating Resource Files .....	69
Adding a Resource File Template and String Variables .....	69
Localizing the Application Code .....	70
Validating the Localization Changes .....	71
<b>Packaging Applications .....</b>	<b>73</b>
Signing .....	73
Compiling an Application in Visual Studio .....	73
<b>Client Object API Usage .....</b>	<b>75</b>
Client Object API Reference .....	75
Application APIs .....	75
Application .....	75
ConnectionProperties .....	87
ApplicationSettings .....	92
ConnectionPropertyType .....	96

Connection APIs .....	101
ConnectionProfile .....	101
Set Database File Property .....	103
Synchronization Profile .....	104
Connect the Data Synchronization Channel	
Through a Relay Server .....	105
Asynchronous Operation Replay .....	105
Authentication APIs .....	106
Logging In .....	106
Importing the Private Key into the System	
Certificate Store .....	106
Sample Code: Setting Up Login Credentials .....	106
Sample Code: Mutual Authentication .....	107
Single Sign-On With X.509 Certificate Related	
Object API .....	108
Personalization APIs .....	110
Type of Personalization Keys .....	110
Getting and Setting Personalization Key Values	
.....	110
Synchronization APIs .....	111
Managing Synchronization Parameters .....	111
Performing Mobile Business Object	
Synchronization .....	112
Message-Based Synchronization APIs .....	112
Push Synchronization Applications .....	117
Retrieving Information about Synchronization	
Groups .....	118
Resumable Synchronization .....	118
Log Record APIs .....	123
LogRecord API .....	123
Logger APIs .....	125
Change Log API .....	125
EntityType .....	125
OperationType .....	126
RootEntityType .....	126

RootSurrogateKey .....	127
SurrogateKey .....	127
Methods in the Generated Database Class .....	128
Code Samples .....	130
Security APIs .....	130
Encrypt the Database .....	130
End to End Encryption and Compression	
Support APIs .....	131
DataVault .....	132
Callback and Listener APIs .....	150
ICallbackHandler API .....	150
IApplicationCallback API .....	156
SyncStatusListener API .....	158
Query APIs .....	160
Retrieving Data from Mobile Business Objects .	160
Retrieving Relationship Data .....	169
Persistence APIs .....	169
Operations APIs .....	169
Object State APIs .....	173
Generated Package Database APIs .....	181
Large Attribute APIs .....	182
MetaData API .....	191
MetaData API .....	191
DatabaseMetaData .....	191
ClassMetaData .....	192
EntityMetaData .....	192
AttributeMetaData .....	192
Exceptions .....	192
Exception Handling .....	192
Exception Classes .....	195
Error Codes .....	196
<b>Index .....</b>	<b>199</b>

# Contents



# Getting Started with Windows and Windows Mobile Development

Use advanced SAP® Mobile Platform features to create applications for Windows and Windows Mobile devices. The audience is advanced developers who may be new to SAP Mobile 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:

- *SAP Mobile WorkSpace - Mobile Business Object Development*
- *Supported Hardware and Software*
- *Tutorial: Windows Mobile Application Development*, where you create the SMP101 sample project referenced in this guide.  
Complete the tutorials to gain a better understanding of SAP Mobile Platform components and the development process.
- *Troubleshooting*.
- A complete Client Object API reference is available in `SMP_HOME\MobileSDK23\ObjectAPI\apidoc\cs`
- *Fundamentals* contains high-level mobile computing concepts, and a description of how SAP Mobile Platform implements the concepts in your enterprise.
- *Developer Guide: Migrating to SAP Mobile SDK* contains information for developers who are migrating device applications to a newer software version, and changes to MBOs, projects, and the SAP Mobile Server.

## Object API Applications

---

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

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

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

Development involves both server-side and client-side components. SAP Mobile Server brokers data synchronization and transaction processing between the server and the client components.

- Server-side components address the interaction between the enterprise information system (EIS) data source and the data cache. EIS data subsets and business logic are encapsulated in artifacts, called mobile business object packages, that are deployed to the SAP Mobile Server.
- Client-side components are built into the mobile application and address the interaction between the data cache and the mobile device data store. This can include synchronizing data with the server, offline data access capabilities, and data change notification.

These applications:

- Allow users to connect to data from a variety of EIS systems, including SAP® systems.
- Build in more complex data handling and logic.
- Leverage data synchronization to optimize and balance device response time and need for real-time data.
- Ensure secure and reliable transport of data.

## **Best Uses for Object API Applications**

---

Synchronization applications provide operation replay between the mobile device, the middleware, and the back-end system. Custom native applications are designed and built to suit specific business scenarios from the ground up, or start with a bespoke application and be adapted with a large degree of customization.

## **Cache Synchronization**

Cache synchronization allows mapping mobile data to SAP Remote Function Calls (RFCs) using Java Connector (JCO) and to other non-SAP data sources such as databases and Web services. When SAP Mobile Platform is used in a stand-alone manner for data synchronization (without Data Orchestration Engine), it utilizes an efficient bulk transfer and data insertion technology between the middleware cache and the device database.

In an SAP Mobile Platform standalone deployment, the mobile application is designed such that the developer specifies how to load data from the back end into the cache and then filters and downloads cache data using device-supplied parameters. The mobile content model and the mapping to the back end are directly integrated.

This style of coupling between device and back-end queries implies that the back end must be able to respond to requests from the middleware based on user-supplied parameters and serve up mobile data appropriately. Normally, some mobile-specific adaptation is required within SAP Business Application Programming Interfaces (BAPI). Because of the direct nature of application parameter mapping and RBS protocol efficiencies, SAP Mobile Platform cache synchronization deployment is ideal:

- With large payloads to devices (may be due to mostly disconnected scenarios)
- Where ad hoc data downloads might be expected
- For SAP® or non-SAP back ends

Large payloads, for example, can occur in task worker (service) applications that must access large product catalogs, or where service occurs in remote locations and workers might synchronize once a day. While SAP Mobile Platform synchronization does benefit from middleware caching, direct coupling requires the back end to support an adaptation where mobile user data can be determined.

### **Client Runtime Architecture**

The goal of synchronization is to keep views (that is, the state) of data consistent among multiple tiers. The assumption is that if data changes on one tier (for example, the enterprise system of record), all other tiers interested in that data (mobile devices, intermediate staging areas/caches and so on) are eventually synchronized to have the same data/state on that system.

The SAP Mobile Server synchronizes data between the device and the back-end by maintaining records of device synchronization activity in its cache database along with any cached data that may have been retrieved from the back-end or pushed from the device. The SAP Mobile Server employs several components in the synchronization chain.

### **Mobile Channel Interfaces**

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

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

The messaging channel sends these types of communications:

- Application registration - the messaging channel is used for application registration before establishing a connection to the SAP Mobile Server.
- Change notifications - when the SAP Mobile Server detects changes in the back-end EIS, the SAP Mobile Server can send a notification to the device. By default, sending change notifications is disabled, but you can enable sending change notifications per synchronization group.

To capture change notifications, you can register an `onSynchronize` callback. The synchronization context in the callback has a status you can retrieve.

- Operation replay records - when synchronizing, these records are sent to the SAP Mobile Server and the messaging channel sends a notification of `replayFinished`. The application must call another `synchronize` method to retrieve the result.
- SAP Data Orchestration Engine (DOE) application synchronization - the messaging channel is used for synchronization for DOE applications.

- The synchronization channel sends data to keep the SAP Mobile Server and client synchronized. The synchronization is bi-directional.

### **Mobile Middleware Services**

Mobile middleware services (MMS) arbitrate and manage communications between device requests from the mobile channel interfaces in the form that is suitable for transformation to a common MBO service request and a canonical form of enterprise data supplied by the data services.

### **Data Services**

Data services is the conduit to enterprise data and operations within the firewall or hosted in the cloud. Data services and mobile middleware services together manage the cache database (CDB) where data is cached as it is synchronized with client devices.

Once a mobile application model is designed, it can be deployed to the SAP Mobile Server where it operates as part of a specialized container-managed package interfacing with the mobile middleware services and data services components. Cache data and messages persist in the databases in the data tier. Changes made on the device are passed to the mobile middleware services component as an operation replay and replayed against the data services interfaces with the EIS. Data that changes on the EIS as a result of device changes, or those originating elsewhere, are replicated to the device database.

## **Documentation Roadmap for SAP Mobile Platform**

SAP® Mobile Platform documents are available for administrative and mobile development user roles. Some administrative documents are also used in the development and test environment; some documents are used by all users.

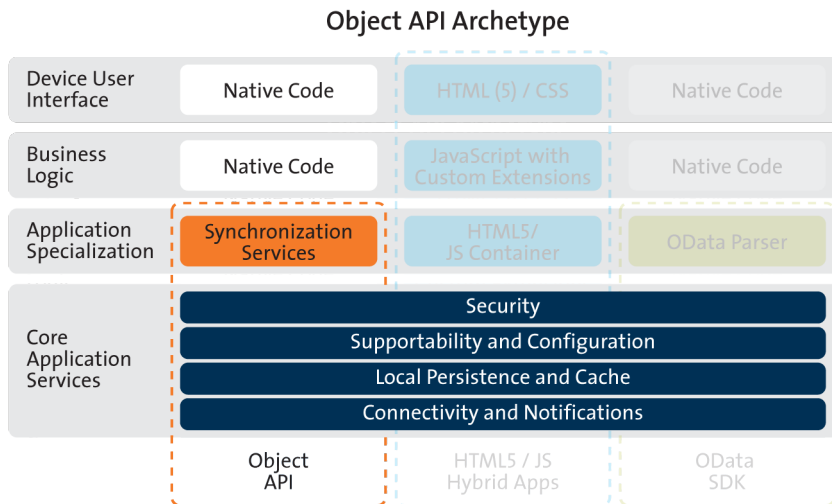
See *Documentation Roadmap* in *Fundamentals* for document descriptions by user role.

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

# Development Task Flow for Object API Applications

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

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



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

The Authentication APIs provide security by authenticating the client to the SAP Mobile Server.

The Synchronization APIs allow you to synchronize mobile business objects (MBOs) based on synchronization parameters, for individual MBOs, or as a group, based on the group's synchronization policy.

The Application and Connection APIs allow clients to register with and connect to the SAP Mobile Server. The Callback Handler and Listener APIs, and the Target Change Notification APIs provide notifications to the client on operation success or failure, or changes in data.

With non-DOE-based applications, connectivity uses the MobiLink™ channel and notifications use the Messaging channel.

## Installing the Windows Mobile Development Environment

Install and configure components required for Windows Mobile development.

### 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

This task enables the emulator's connectivity.

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. 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.

### Installing Microsoft Synchronization Software

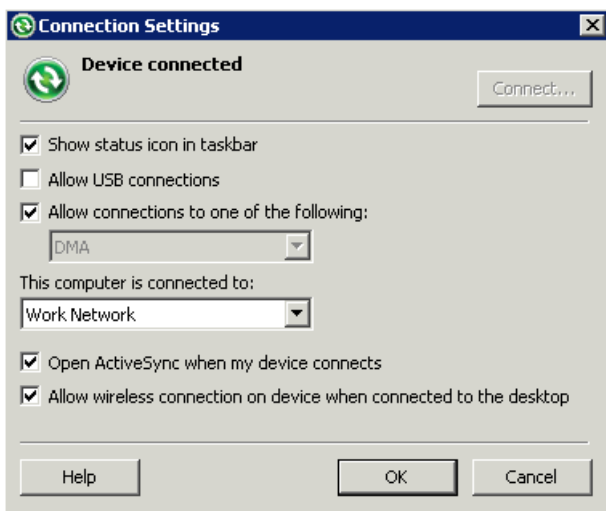
Install and configure Microsoft synchronization software so you can deploy and run a mobile application on a Windows Mobile emulator.

---

**Note:** Microsoft ActiveSync is for Windows XP. If you are using Windows Vista, Windows 7, or Windows 2008, install Virtual PC 2007 SP1 and Windows Mobile Device Center to manage synchronization settings. Download the Windows Mobile Device Center from <http://www.microsoft.com/windowsmobile/en-us/downloads/microsoft/device-center-download.msp>.

---

1. Download Microsoft ActiveSync from <http://www.microsoft.com/windowsmobile/en-us/help/synchronize/device-synch.mspx>. Save it to your local machine. Windows XP requires ActiveSync version 4.5.
2. In Windows Explorer, double-click **setup.msi**.
3. When installation is complete, restart your machine.  
ActiveSync starts automatically, and its icon appears in the Windows toolbar.
4. Double-click the **ActiveSync** icon.
5. Select **File > Connection Settings**.
6. Select **Allow Connections to One of the Following**, then select **DMA**.
7. Under This Computer is Connected to, select **Work Network**.



8. Click **OK**.

## Installing X.509 Certificates on Windows Mobile Devices and Emulators

Install the .p12 certificate on the Windows Mobile device or emulator and select it during authentication. A certificate provides an additional level of secure access to an application, and may be required by an organization's security policy.

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.

## Generating C# Object API Code

---

Generate object API code containing mobile business object (MBO) references, which allows you to use APIs to develop device applications for Windows or Windows Mobile devices. You can generate code either in SAP Mobile WorkSpace, or by using a command line utility for generating code.

Generated code can be used to leverage SAP Mobile Platform capabilities and services, and access MBO-related data: calling the mobile business object operations, object queries, and so on. This code can then be imported into an integrated development environment (IDE) of your choice to create the device application (define the user interface, application logic, and so on).

## Generating C# Object API Code Using SAP Mobile WorkSpace

---

Use SAP Mobile WorkSpace to generate object API code containing mobile business object (MBO) references.

### Prerequisites

Develop the MBOs that will be referenced in the device applications you are developing. A mobile application project must contain at least one non-online MBO. You must have an active connection to the datasources to which the MBOs are bound.

### Task

SAP Mobile Platform provides the Code Generation wizard for generating object API code. Code generation creates the business logic, attributes, and operations for your mobile business object.

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

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

2. (Optional; this page of the code generation wizard is seen only if you are using the Advanced developer profile). Enter the information for these options, then click **Next**:



Option	Description
Code generation configuration	<p>A table lists all existing named configurations plus the most recently used configuration. You can select any of these, click <b>Next</b>, and proceed. Additionally, you can:</p> <ul style="list-style-type: none"> <li>• Create new configuration – click <b>Add</b> and enter the <b>Name</b> and optional <b>Description</b> of the new configuration and click <b>OK</b> to save the configuration for future sessions. You can also select <b>Copy from</b> to copy an existing configuration which can then be modified.</li> <li>• Most recent configuration – if you click <b>Next</b> the first time you generate code without creating a configuration, the configuration is saved and displays as the chosen configuration the next time you invoke the code generation wizard. If the most recent configuration used is a named configuration, it is saved as the first item in the configuration table, and also "Most recent configuration", even though it is still listed as the original named configuration.</li> </ul>

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.

SAP Mobile WorkSpace automatically computes the default page size after you choose the MBOs based on total attribute size. If an MBO's accumulated attribute size is larger than the page size setting, a warning displays.

5. Enter the information for these configuration options:

Option	Description
Language	Select C#.
Platform	<p>Select the platform ( target device) for which the device client code is intended.</p> <ul style="list-style-type: none"> <li>• C# <ul style="list-style-type: none"> <li>• .NET Framework for Windows</li> <li>• .NET Compact Framework 3.5 for Windows Mobile</li> </ul> </li> </ul>
SAP Mobile Server	Specify a default SAP Mobile Server connection profile to which the generated code connects at runtime.

Option	Description
Server domain	<p>Choose the domain to which the generated code will connect. If you specified an SAP Mobile Server to which you previously connected successfully, the first domain in the list is chosen by default. You can enter a different domain manually.</p> <hr/> <p><b>Note:</b> This field is only enabled when an SAP Mobile Server is selected.</p>
Page size	<p>(Optional) Select the page size for the generated client code. If the page size is not set, the default page size is 4KB at runtime. The default is a proposed page size based on the selected MBO's attributes. The maximum page size is 16KB</p> <p>The page size should be larger than the sum of all attribute lengths for any MBO that is included with all the MBOs selected, and must be valid for the database. If the page size is changed, but does not meet these guidelines, object queries that use string or binary attributes with a <b>WHERE</b> clause may fail. See <i>MBO Attributes</i> in <i>Mobile Data Models: Using Mobile Business Objects</i> for more information.</p> <p>A binary length greater than 32767 is converted to a binary large object (BLOB), and is not included in the sum; a string greater than 8191 is converted to a character large object (CLOB), and is also not included). If an MBO attribute's length sum is greater than the page size, some attributes automatically convert to BLOB or CLOB, and therefore cannot be put into a <b>WHERE</b> clause.</p> <hr/> <p><b>Note:</b> This field is only enabled when an SAP Mobile Server is selected.</p>
Destination	<p>Specify the destination of the generated device client files. Enter (or <b>Browse</b>) to either a <b>Project path</b> (Mobile Application project) location or <b>File system path</b> location. Select <b>Clean up destination before code generation</b> to clean up the destination folder before generating the device client files.</p>

6. Select **Generate metadata classes** to generate metadata for the attributes and operations of each generated client object.

The **Including object manager classes** option is only available if you select **Generate metadata classes**.

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

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

8. Click **Finish**.
9. Examine the generated code location and contents.
10. Validate the generated code.

## Generating Object API Code Using the Code Generation Utility

Use the Code Generation Utility to generate object API code containing mobile business object (MBO) references. This method of generating code allows you to automate the process of code generation, for example through the use of scripts.

### Prerequisites

- Use SAP Mobile WorkSpace to develop and package your mobile business objects. See *SAP Mobile WorkSpace - Mobile Business Object Development > Develop > Developing a Mobile Business Object*.
- Deploy the package to the SAP Mobile Server, creating files required for code generation from the command line. See *SAP Mobile WorkSpace - Mobile Business Object Development > Develop > Packaging and Deploying Mobile Business Objects > Automated Deployment of SAP Mobile WorkSpace Projects*.

### Task

1. Locate `<domain name>_package.jar` in your mobile project folder. For the SMP101 example, the project is deployed to the default domain, and the deploy jar file is in the following location: `SMP101\Deployment\.pkg.profile\My_SAP_Mobile_Server\default_package.jar`.
2. Make sure that the JAR file contains this file:
  - `deployment_unit.xml`
3. Use a utility to extract the `deployment_unit.xml` file to another location.
4. From `SMP_HOME\MobileSDK23\ObjectAPI\Utils\bin`, run the `codegen.bat` utility, specifying the following parameters:

## Development Task Flow for Object API Applications

```
codegen.bat -cs -client -ul deployment_unit.xml [-output  
<output_dir>] [-doc]
```

- The `-output` parameter allows you to specify an output directory. If you omit this parameter, the output goes into the `SMP_HOME\MobileSDK23\ObjectAPI\Utils\genfiles` directory, assuming `codegen.bat` is run from the `SMP_HOME\MobileSDK23\ObjectAPI\Utils\genfiles` directory.
- The `-doc` parameter specifies that documentation is generated for the generated code.

Ignore these warnings:

```
log4j:WARN No appenders could be found for logger ...  
log4j:WARN Please initialize the log4j system properly.
```

### Generated Code Location and Contents

If you generated code in SAP Mobile WorkSpace, generated object API code is stored by default in the "Destination" location you specified during code generation. If you generated code with the Code Generation Utility, generated object API code is stored in the `SMP_HOME\MobileSDK23\ObjectAPI\Utils\genfiles` folder after you generate code.

The contents of the folder is determined by the options you selected in the Generate Code wizard in SAP Mobile WorkSpace, or specified in the Code Generation Utility. The contents include generated class (.cs) files that contain:

- MBO – class which handles persistence and operation replay of your MBOs.
- DatabaseClass – package level class that handles subscription, login, synchronization, and other operations for the package.
- Synchronization parameters – any synchronization parameters for the MBOs.
- Personalization parameters – personalization parameters used by the package.
- Metadata – Metadata class that allow you to query meta data including MBOs, their attributes, and operations, in a persistent table at runtime.

### Validating Generated Code

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

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

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

1. Select **Window > Preferences**.

2. Expand **SAP AG > Mobile Development**.
3. Select **Mobile Business Object**.
4. Add or modify the **Naming Prefix** settings as needed.

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

## **Creating a Project**

---

Build a device application project.

### **Downloading the Latest Afaria Libraries**

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

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

### **Creating a Project in Visual Studio**

---

Create a project for your Windows or Windows Mobile device application in Visual Studio.

#### **Prerequisites**

On Windows 8, you must manually enable the .NET 3.5 Framework using the Windows Features control panel applet before running any applications that require the .NET 3.5 Framework.

#### **Task**

1. From the Visual Studio main menu select **File > New > Project**.
2. Select:
  - Target platform:
    - Windows Mobile 6.0, 6.1, and 6.5 Professional
    - Windows

## Development Task Flow for Object API Applications

- 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.
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:
  - *SAP Mobile WorkSpace - Mobile Business Object Development > 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 *Rebuilding the Generated Solution in Visual Studio*.

### **Rebuilding the Generated Solution in Visual Studio**

After generating the C# Object API code for a Mobile Application project, you can modify the code in Visual Studio.

#### **Prerequisites**

Visual Studio must be installed.

#### **Task**

When you generate the Object API code for a Windows Mobile device using the Mobile Application project Code Generation wizard, the Visual Studio solution is saved to the folder you specified.

---

**Note:** If you generate 32-bit Windows code on a 64-bit machine, in the Visual Studio Configuration Manager for building and deploying the active solution, change the Active Solution Platform from "AnyCPU" to "x86" before deploying and running the application.

---

1. In Visual Studio, select **File > Open > Project/Solution**.
2. Browse to the solution file (.sln) you want to open and double-click the file.
3. In Solution Explorer, right-click the solution and select **Rebuild Solution**.
4. Select **File > Save**.

You can now open the form for which you want to modify the code.

## Generating Online Help for Visual Studio

You can generate online help for your mobile application project using the .xml file that is generated with the Object API code.

When you generate the C# Object API code for the mobile application project, an .xml file is generated for the mobile business objects. You can use the generated .xml file to build online help for the mobile application project in Visual Studio.

1. Use the Code Generation wizard to generate the C# Object API Code.
2. Build the generated solution in Visual Studio:
  - a) In Visual Studio, select **File > Open > Project/Solution**.
  - b) Browse to the solution file (.sln) you want to open and double-click the file.
  - c) In Solution Explorer, right-click the solution and select **Rebuild Solution**.
  - d) Select **File > Save**.  
The source .xml file for building online help is included with the generated project, which contains all assemblies and runtime support DLLs required to access the object API.
3. There are different tools available for building online help. This procedure uses the free tool called Sandcastle Help File Builder. You can download Sandcastle Help File Builder installer from <http://shfb.codeplex.com>.
4. Create a SandCastle Helpfile builder project (.shfb) in Sand Castle File Builder by specifying the assemblies and the generated .xml file as input.
5. Use the .shfb project file in the daily build script to build the document. For example:

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

## Integrating Help into a Project

When you generate MBOs or client applications for Windows Mobile from SAP Mobile 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:

## Development Task Flow for Object API Applications

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>
```

### Client Application Dependencies

To build device clients, some files, which are provided in the SAP Mobile 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 *SMP\_HOME* \MobileSDK<version>\ObjectAPI directory. The contents of the directory are:

- **WM and Win32** – Binaries of the framework classes for .NET.
  - WM: files for use on Windows CE based systems such as Windows Mobile 6.
  - Win32: files for use on full Windows based systems like Windows XP.
- **WM\Ultralite and Win32\Ultralite** – .NET Data Persistence Library and client database (UltraLite®) assemblies. This is used for replication-based synchronization client applications on Windows Mobile or Windows.

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	<i>SMP_HOME</i> \MobileSDK<version>\ObjectAPI\WM	<ul style="list-style-type: none"><li>• sup-client.dll</li></ul>	



## Development Task Flow for Object API Applications

Platform	Location	Files	Notes
	<ul style="list-style-type: none"> <li>• <i>SMP_HOME</i>  \ Mobi-  leSDK&lt;ver-  sion&gt;\Ob-  jectAPI\WM  \PocketPC, or</li> <li>• <i>SMP_HOME</i>  \ Mobi-  leSDK&lt;ver-  sion&gt;\Ob-  jectAPI\WM  \Smartphone</li> </ul>	CMessaging- Client.<ver- sion>.dll	
	<i>SMP_HOME</i> \Mo- bileSDK<ver- sion>\Objec- tAPI\WM\Ul- tralite	<ul style="list-style-type: none"> <li>• ulnet12.dll</li> <li>• ulnet- cli- ent12.dll</li> <li>• mlcrsa12.dl l</li> <li>• mlczlib12.d ll</li> </ul>	The mlcrsa12.dll file is needed only if you are using a secure port (HTTPS) for syn- chronization.
	<i>SMP_HOME</i> \Mo- bileSDK<ver- sion>\Objec- tAPI\WM\Ul- tralite\<lan- guage>	iAnywhere.Da- ta.Ultra- Lite.resour- ces.dll	Copy from the respec- tive locale-specific folders.

## Development Task Flow for Object API Applications

Platform	Location	Files	Notes
Windows XP, Vista, Windows 7	<i>SMP_HOME</i> \MobileSDK<version>\ObjectAPI\Win32	<ul style="list-style-type: none"> <li>• sup-client.dll</li> <li>• ECTrace.dll</li> <li>• CMessagingClient.&lt;version&gt;.dll</li> <li>• TravelerLib.dll</li> <li>• zlib1.dll</li> <li>• li-beay32.dll</li> <li>• ssleay32.dll</li> </ul>	
	<i>SMP_HOME</i> \MobileSDK<version>\ObjectAPI\Win32\Ultralite	<ul style="list-style-type: none"> <li>• iAnywhere.Data.UltraLite.dll</li> <li>• ulnet12.dll</li> <li>• mlcrsa12.dll</li> <li>• mlczlib12.dll</li> </ul>	The mlcrsa12.dll file is required only if you are using a secure port (HTTPS) for synchronization.
	<i>SMP_HOME</i> \MobileSDK<version>\ObjectAPI\Win32\Ultralite\<language>	iAnywhere.Data.UltraLite.resources.dll	Copy from the respective locale-specific folders.
Windows CE 6.0	<i>SMP_HOME</i> \MobileSDK<version>\ObjectAPI\WM	<ul style="list-style-type: none"> <li>• sup-client.dll</li> </ul>	

Platform	Location	Files	Notes
	<i>SMP_HOME</i> \MobileSDK<version>\ObjectAPI\WM\PocketPC	CMessaging-Client.<version>.dll	
	<i>SMP_HOME</i> \MobileSDK<version>\ObjectAPI\WM\Ultralite	<ul style="list-style-type: none"> <li>• ulnet12.dll</li> <li>• ulnet-client12.dll</li> <li>• mlcrsa12.dll</li> <li>• mlczlib12.dll</li> </ul>	The mlcrsa12.dll file is needed only if you are using a secure port (HTTPS) for synchronization.
	<i>SMP_HOME</i> \MobileSDK<version>\ObjectAPI\WM\Ultralite\<language>	iAnywhere.Data.UltraLite.resources.dll	Copy from the respective locale-specific folders.

## Adding References to a Mobile Application Project

This describes how to add the required libraries as references in the Visual Studio project.

You can use this method to add references to your 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 *SMP\_HOME*\MobileSDK<version>\ObjectAPI\WM.
- iAnywhere.Data.UltraLite.dll – from *SMP\_HOME*\MobileSDK<version>\ObjectAPI\WM\Ultralite.
- iAnywhere.Data.UltraLite.resources.dll (several languages are supported) – from *SMP\_HOME*\MobileSDK<version>\ObjectAPI\WM\Ultralite\<language>.

For Windows:

## Development Task Flow for Object API Applications

- `sup-client.dll` – from `SMP_HOME\MobileSDK<version>\ObjectAPI\Win32`.
  - `iAnywhere.Data.UltraLite.dll` – from `SMP_HOME\MobileSDK<version>\ObjectAPI\Win32\Ultralite`.
  - `iAnywhere.Data.UltraLite.resources.dll` (several languages are supported) – from `SMP_HOME\MobileSDK<version>\ObjectAPI\Win32\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:

- `ulnet12.dll` – from `SMP_HOME\MobileSDK<version>\ObjectAPI\WM\Ultralite`.
- `mlcrsa12.dll` (if HTTPS protocol is used) – from `SMP_HOME\MobileSDK<version>\ObjectAPI\WM\Ultralite`.
- `mlczlib12.dll` (if HTTPS protocol is used) – from `SMP_HOME\MobileSDK<version>\ObjectAPI\WM\Ultralite`.
- `CMessagingClient.<version>.dll` – from `SMP_HOME\MobileSDK<version>\ObjectAPI\WM\<DeviceType>`. `<DeviceType>` can be PocketPC or Smartphone as applicable.

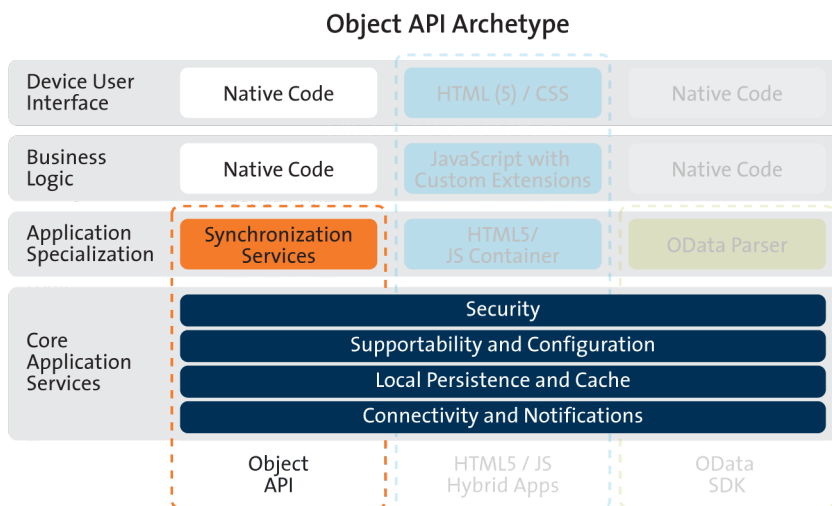
### For Windows:

- `ulnet12.dll` – from `SMP_HOME\MobileSDK<version>\ObjectAPI\Win32\Ultralite`.
- `mlcrsa12.dll` (if HTTPS protocol is used) – from `SMP_HOME\MobileSDK<version>\ObjectAPI\Win32\Ultralite`.
- `mlczlib12.dll` (if using compression) - from `SMP_HOME\MobileSDK<version>\ObjectAPI\Win32\Ultralite`.
- `CMessagingClient.<version>.dll` – from `SMP_HOME\MobileSDK<version>\ObjectAPI\Win32`.
- `ECTrace.dll` – from `SMP_HOME\MobileSDK<version>\ObjectAPI\Win32`.
- `TravelerLib.dll` – from `SMP_HOME\MobileSDK<version>\ObjectAPI\Win32`.
- `zlib1.dll` – from `SMP_HOME\MobileSDK<version>\ObjectAPI\Win32`.
- `libeay32.dll` – from `SMP_HOME\MobileSDK<version>\ObjectAPI\Win32`.
- `ssleay32.dll` – from `SMP_HOME\MobileSDK<version>\ObjectAPI\Win32`.

# Development Task Flow for DOE-based Object API Applications

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

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



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

The Authentication APIs provide security by authenticating the client to the SAP Mobile Server.

The Synchronization APIs allow you to synchronize mobile business objects (MBOs) based on synchronization parameters, for individual MBOs, or as a group, based on the group's synchronization policy.

The Application and Connection APIs allow clients to register with and connect to the SAP Mobile Server. The Callback Handler and Listener APIs, and the Target Change Notification APIs provide notifications to the client on operation success or failure, or changes in data.

With DOE-based applications, connectivity and notifications use the Messaging channel.

## Installing the Windows Mobile Development Environment

Install and configure components required for Windows Mobile development.

### 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

This task enables the emulator's connectivity.

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. 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.

### Installing Microsoft Synchronization Software

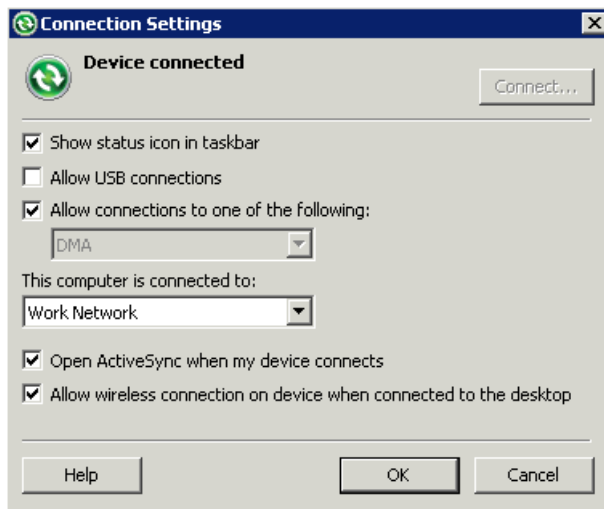
Install and configure Microsoft synchronization software so you can deploy and run a mobile application on a Windows Mobile emulator.

---

**Note:** Microsoft ActiveSync is for Windows XP. If you are using Windows Vista, Windows 7, or Windows 2008, install Virtual PC 2007 SP1 and Windows Mobile Device Center to manage synchronization settings. Download the Windows Mobile Device Center from <http://www.microsoft.com/windowsmobile/en-us/downloads/microsoft/device-center-download.msp>.

---

1. Download Microsoft ActiveSync from <http://www.microsoft.com/windowsmobile/en-us/help/synchronize/device-synch.mspx>. Save it to your local machine. Windows XP requires ActiveSync version 4.5.
2. In Windows Explorer, double-click **setup.msi**.
3. When installation is complete, restart your machine.  
ActiveSync starts automatically, and its icon appears in the Windows toolbar.
4. Double-click the **ActiveSync** icon.
5. Select **File > Connection Settings**.
6. Select **Allow Connections to One of the Following**, then select **DMA**.
7. Under This Computer is Connected to, select **Work Network**.



8. Click **OK**.

## Installing X.509 Certificates on Windows Mobile Devices and Emulators

Install the .p12 certificate on the Windows Mobile device or emulator and select it during authentication. A certificate provides an additional level of secure access to an application, and may be required by an organization's security policy.

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.

## Generating C# Object API Code

---

Use the Code Generation Utility to generate object API code, which allows you to use APIs to develop device applications for Windows or Windows Mobile devices.

### Prerequisites

- Generate and download the ESDMA bundle for you application.
- Run the ESDMA Converter utility to turn your ESDMA into an SAP Mobile Platform package.
- Deploy the package to the SAP Mobile Server.

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

### Task

1. Make sure that your `<ESDMA_dir>\META-INF` directory contains these three files:

- `afx-esdma.xml`
- `ds-doe.xml`
- `sup-db.xml`

2. From `SMP_HOME\MobileSDK23\ObjectAPI\Utils\bin`, run the `codegen.bat` utility, specifying the following parameters:

```
codegen -cs -client -doe -sqlite  
[-output <output_dir>] [-doc] <ESDMA_dir>\META-INF\sup-db.xml
```

- The `-output` parameter allows you to specify an output directory. If you omit this parameter, the output goes into the `SMP_HOME\MobileSDK23\ObjectAPI\Utils\genfiles` directory, assuming `codegen.bat` is run from the `SMP_HOME\MobileSDK23\ObjectAPI\Utils\bin` directory.
- The `-doc` parameter specifies that documentation is generated for the generated code.

Ignore these warnings:

```
log4j:WARN No appenders could be found for logger ...  
log4j:WARN Please initialize the log4j system properly.
```

## Generated Code Location and Contents

The location of the generated Object API code is the location you specified when you generated the code using `codegen.bat` at the command line.

The contents of the folder is determined by the parameters you pass to `codegen.bat` in the command line, and include generated class (.cs) files that contain:



- DatabaseClass – package level class that handles subscription, login, synchronization, and other operations for the package.
- MBO – class which handles persistence and operation replay of your MBOs.
- Personalization parameters – personalization parameters used by the package.
- Metadata – Metadata class that allows you to query meta data including MBOs, their attributes, and operations, in a persistent table at runtime.

## Creating a Project

---

Build a device application project.

### Downloading the Latest Afaria Libraries

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

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

### Creating a Project in Visual Studio

Create a project for your Windows or Windows Mobile device application in Visual Studio.

#### **Prerequisites**

On Windows 8, you must manually enable the .NET 3.5 Framework using the Windows Features control panel applet before running any applications that require the .NET 3.5 Framework.

#### **Task**

1. From the Visual Studio main menu, select **File > New > Project**.
2. Select:
  - Target platform:
    - Windows Mobile 6.0, 6.1, and 6.5 Professional
    - Windows
  - Library version – .NET version 3.5

## Development Task Flow for DOE-based Object API Applications

- 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.
4. Add all necessary dlls and assemblies:
  - a) Choose **Project > Add Reference** and add `SMP_HOME\MobileSDK<version>\ObjectAPI\DOE\WM\sup-client.dll` and `SMP_HOME\MobileSDK<version>\ObjectAPI\DOE\WM\System.Data.SQLite.dll`.
  - b) Choose **Project > Add Existing Item** and add `SMP_HOME\MobileSDK<version>\ObjectAPI\DOE\WM\SQLite.Interop.065.DLL` and `SMP_HOME\MobileSDK<version>\ObjectAPI\DOE\WM\PocketPC\CMessagingClient.<version>.dll`.
  - c) In the Solution window, open `SQLite.Interop.065.DLL` and change the property **Copy to Output Directory** to **Copy always**.
  - d) In the Solution window, open `CMessagingClient.<version>.dll` and change the property **Copy to Output Directory** to **Copy always**.
5. 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.
6. Develop the mobile business objects (MBOs) that implement the business logic. See these online help topics:
  - *SAP Mobile WorkSpace - Mobile Business Object Development > Develop > Developing a Mobile Business Object*
7. Use the Code Generation wizard to generate the C# Object API code for the mobile business object.
8. Add the generated code to the new project you created from the template. For more information, see *Rebuilding the Generated Solution in Visual Studio*.

### Rebuilding the Generated Solution in Visual Studio

After generating the C# Object API code for a Mobile Application project, you can modify the code in Visual Studio.

#### **Prerequisites**

Visual Studio must be installed.

## Task

When you generate the Object API code for a Windows Mobile device using the Mobile Application project Code Generation wizard, the Visual Studio solution is saved to the folder you specified.

---

**Note:** If you generate 32-bit Windows code on a 64-bit machine, in the Visual Studio Configuration Manager for building and deploying the active solution, change the Active Solution Platform from "AnyCPU" to "x86" before deploying and running the application.

---

1. In Visual Studio, select **File > Open > Project/Solution**.
2. Browse to the solution file (.sln) you want to open and double-click the file.
3. In Solution Explorer, right-click the solution and select **Rebuild Solution**.
4. Select **File > Save**.

You can now open the form for which you want to modify the code.

## Generating Online Help for Visual Studio

You can generate online help for your mobile application project using the .xml file that is generated with the Object API code.

When you generate the C# Object API code for the mobile application project, an .xml file is generated for the mobile business objects. You can use the generated .xml file to build online help for the mobile application project in Visual Studio.

1. Use the Code Generation wizard to generate the C# Object API Code.
2. Build the generated solution in Visual Studio:
  - a) In Visual Studio, select **File > Open > Project/Solution**.
  - b) Browse to the solution file (.sln) you want to open and double-click the file.
  - c) In Solution Explorer, right-click the solution and select **Rebuild Solution**.
  - d) Select **File > Save**.

The source .xml file for building online help is included with the generated project, which contains all assemblies and runtime support DLLs required to access the object API.

3. There are different tools available for building online help. This procedure uses the free tool called Sandcastle Help File Builder. You can download Sandcastle Help File Builder installer from <http://shfb.codeplex.com>.
4. Create a SandCastle Helpfile builder project (.shfb) in Sand Castle File Builder by specifying the assemblies and the generated .xml file as input.
5. Use the .shfb project file in the daily build script to build the document. For example:

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

### Integrating Help into a Project

When you generate MBOs or client applications for Windows Mobile from SAP Mobile 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>
```

### Client Application Dependencies

To build device clients, some files, which are provided in the SAP Mobile 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 *SMP\_HOME* \MobileSDK<version>\ObjecttAPI\DOE directory. The contents of the directory are:

- **WM and Win32** – Binaries of the framework classes for .NET.
  - WM: files for use on Windows CE based systems such as Windows Mobile 6.
  - Win32: files for use on full Windows based systems like Windows XP.

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

## Development Task Flow for DOE-based Object API Applications

- 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	<i>SMP_HOME</i> \MobileSDK\ObjectAPI\DOE\WM	<ul style="list-style-type: none"> <li>• sup-client.dll</li> <li>• SQLite.Interop.065.DLL (native DLL)</li> <li>• System.Data.SQLite.dll (assembly)</li> </ul>	
	<ul style="list-style-type: none"> <li>• <i>SMP_HOME</i>\MobileSDK&lt;version&gt;\ObjectAPI\DOE\WM\PocketPC, or</li> <li>• <i>SMP_HOME</i>\MobileSDK&lt;version&gt;\ObjectAPI\DOE\WM\Smartphone</li> </ul>	CMessagingClient.<version>.dll	<version> represents the most current version.

Platform	Location	Files	Notes
Windows XP, Vista, Windows 7	<i>SMP_HOME</i> \MobileSDK<version>\ObjectAPI\DOE\Win32	<ul style="list-style-type: none"> <li>• sup-client.dll</li> <li>• System.Data.SQLite.dll (assembly)</li> <li>• ECTrace.dll</li> <li>• CMessagingClient.&lt;version&gt;.dll</li> <li>• TravelerLib.dll</li> <li>• zlib1.dll</li> <li>• libeay32.dll</li> <li>• ssleay32.dll</li> </ul>	<version> represents the most current version.

### Adding References to a Mobile Application Project

This describes how to add the required libraries as references in the Visual Studio project.

You can use this method to add references to your client projects.

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

For Windows Mobile:

- sup-client.dll – from *SMP\_HOME*\MobileSDK<version>\ObjectAPI\WM.
- CMessagingClient.<version>.dll – from *SMP\_HOME*\MobileSDK<version>\ObjectAPI\DOE\WM\PocketPC or *SMP\_HOME*\MobileSDK<version>\ObjectAPI\DOE\WM\Smartphone.
- SQLite.Interop.065.dll – from *SMP\_HOME*\MobileSDK<version>\ObjectAPI\DOE\WM\.
- iAnywhere.Data.UltraLite.resources.dll (several languages are supported) – from *SMP\_HOME*\MobileSDK<version>\ObjectAPI\DOE\WM\Ultralite\<language>.

For Windows:

## Development Task Flow for DOE-based Object API Applications

- `sup-client.dll` – from `SMP_HOME\MobileSDK<version>\ObjectAPI\DOE\Win32`.
  - `ETrace.dll` – from `SMP_HOME\MobileSDK<version>\ObjectAPI\DOE\Win32`.
  - `CMessagingClient.<version>.dll` – from `SMP_HOME\MobileSDK<version>\ObjectAPI\DOE\Win32`.
  - `Traveler.dll` – from `SMP_HOME\MobileSDK<version>\ObjectAPI\DOE\Win32`.
  - `zlib1.dll` – from `SMP_HOME\MobileSDK<version>\ObjectAPI\DOE\Win32`.
  - `libeay32.dll` – from `SMP_HOME\MobileSDK<version>\ObjectAPI\Win32`.
  - `ssleay32.dll` – from `SMP_HOME\MobileSDK<version>\ObjectAPI\Win32`.
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:

- `ulnet12.dll` – from `SMP_HOME\MobileSDK<version>\ObjectAPI\DOE\WM\Ultralite`.
- `ulnetclient12.dll` – from `SMP_HOME\MobileSDK<version>\ObjectAPI\DOE\WM\Ultralite`.
- `m1crsa12.dll` (if HTTPS protocol is used) – from `SMP_HOME\MobileSDK<version>\ObjectAPI\DOE\WM\Ultralite`.
- `m1czlib12.dll` (if HTTPS protocol is used) – from `SMP_HOME\MobileSDK<version>\ObjectAPI\DOE\WM\Ultralite`.

For Windows:

- `ulnet12.dll` – from `SMP_HOME\MobileSDK<version>\ObjectAPI\DOE\Win32\Ultralite`.
- `m1crsa12.dll` (if HTTPS protocol is used) – from `SMP_HOME\MobileSDK<version>\ObjectAPI\DOE\Win32\Ultralite`.
- `m1czlib12.dll` (if using compression) - from `SMP_HOME\MobileSDK<version>\ObjectAPI\DOE\Win32\Ultralite`.





# Developing the Application Using the Object API

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

## Initializing an Application

---

Initialize the application when it starts the first time and subsequently.

## Initially Starting an Application

Starting an application the first time.

### Setting Up Application Properties

The Application instance contains the information and authentication credentials needed to register and connect to the SAP Mobile Server.

The following code illustrates how to set up the minimum required fields:

```
// Initialize Application settings
Application app = Application.GetInstance();

// The identifier has to match the application ID deployed to the SAP
// Mobile Server
app.ApplicationIdentifier = "SMP101";

// ConnectionProperties has the information needed to register
// and connect to SAP Mobile Server
ConnectionProperties connProps = app.ConnectionProperties;
connProps.ServerName = "server.mycompany.com";
// if you are using relay server, then use the correct port number
// for the relay server
// if connecting using http without a relay server, use the messaging
// administration port, by default 5001.
// if connecting using https without a relay server, then use a new
// port for https, for example 9001.
connProps.PortNumber = 5001;
// if connecting using https, set the network protocol
connProps.NetworkProtocol = "https";

// Set FarmId and UrlSuffix when connecting through the Relay
// Server.

// Provide user credentials
LoginCredentials loginCred = new LoginCredentials("supAdmin",
```

## Developing the Application Using the Object API

```
"supPwd");  
connProps.LoginCredentials = loginCred;  
  
// Initialize generated package database class with this Application  
instance  
SMP101DB.SetApplication(app);
```

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

```
// specify Relay Server Host  
connProps.ServerName = "relayserver.mycompany.com";  
// specify Relay Server Port (port 80 by default)  
connProps.PortNumber = 80;  
// specify the Relay Server MBS Farm, for example MBS_Farm  
connProps.FarmId = "MBS_FARM";
```

Optionally, you can specify the Relay Server URL suffix.

### *Using a Reverse Proxy for Object API applications*

The Object API application communicates with SAP Mobile Server through two ports:

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

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

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

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

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

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

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

---

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

---

### Communicating with SAP Mobile Server Through a Reverse Proxy

Connect to SAP Mobile Server through a Reverse Proxy using Relay Server or the Apache Web server.

The Object API application can communicate with SAP Mobile Server:

- Directly
- Through Relay Server by specifying:
  - Just the FarmID
  - Just the URL suffix
  - Both the FarmID and URL suffix
- Through a Reverse Proxy by specifying only the URL suffix - you can configure the reverse proxy in such a way that the device communicates to `http://server:port/customcontext/tm` or `/tm2` when communicating with SAP Mobile Server. "FarmID" has no meaning when using a Reverse Proxy. In this case use only the URL suffix and leave "FarmID" empty.

---

**Note:** You can also use the custom context with the Relay Server when using Apache.

---

### Connecting to SAP Mobile Server through a Reverse Proxy using Relay Server:

```
//Register:
Application.GetInstance().ConnectionProperties.UrlSuffix =
"/ias_relay_server/client/rs_client.dll/$messaging-farmId$"
// (e.g. /ias_relay_server/client/rs_client.dll/mega-vm008.msg)

//Synchronize:
TestDB.GetSynchronizationProfile().StreamParams.Url_Suffix =
"/ias_relay_server/client/rs_client.dll/$replication-farmId$"
// (e.g. /ias_relay_server/client/rs_client.dll/mega-vm008.rep)
```

### Connecting to SAP Mobile Server through a Reverse Proxy using Apache Web server as proxy server:

```
Add to the httpd.conf file:
Listen 80 <VirtualHost *:80>ServerName proxy-server
<Location /app1/>
ProxyPass http://sup-server:5001/ ProxyPassReverse
http://sup-server:5001/
</Location> <Location /app2/>
ProxyPass http://sup-server:2480/
ProxyPassReverse http://sup-server:2480/
</Location>
</VirtualHost>

//Register:
Application.GetInstance().ConnectionProperties.UrlSuffix = "/app1";

//Synchronize:
TestDB.GetSynchronizationProfile().StreamParams.Url_Suffix = "/
app2"
```

### **Registering an Application**

Each device must register with the server before establishing a connection.

To register the device with the server during the initial application startup, use the `RegisterApplication` method in the `Sybase.Mobile.Application` class. You do not need to use the `RegisterApplication` method for subsequent application start-ups. The `RegisterApplication` method automatically starts the connection to complete the registration process.

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

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

```
// Initialize Application settings
Application app = Application.GetInstance();

// The identifier has to match the
// application ID deployed to the SAP Mobile Server
app.ApplicationIdentifier = "SMP101";
IApplicationCallback appCallback = new MyApplicationCallback();
app.ApplicationCallback = appCallback;

// set connection properties, login credentials, etc
...

SMP101DB.SetApplication(app);

if (app.RegistrationStatus != RegistrationStatus.REGISTERED)
{
    // If the application has not been registered to the server,
    // register now
    app.RegisterApplication(<timeout_value>);
}
else
{
    // start the connection to server
    app.StartConnection(<timeout_value>);
}
```

### **Setting Up the Connection Profile**

The Connection Profile stores information detailing where and how the local database is stored, including location and page size. The connection profile also contains UltraLite®J runtime tuning values.

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

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

Use the `Sybase.Persistence.ConnectionProfile` class to set up the locally generated database. Retrieve the connection profile object using the SAP Mobile Platform database's `GetConnectionProfile` method.

```
// Initialize the device database connection profile (if needed)
ConnectionProfile connProfile = SMP101DB.GetConnectionProfile();

// encrypt the database
connProfile.SetEncryptionKey("your encryption key"); //Encryption
key can be of arbitrary length, but generally the longer, the better.
// You can also automatically generate a encryption key and store it
inside a data vault.

// use 100K for cache size
connProfile.CacheSize = 102400;

// Store the database in a location other than the default location
connProfile.SetProperty("databaseFile", "SMP1011_0.udb");
```

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

```
SMP101DB.GetConnectionProfile().SetProperty("allowConcurrentWrite",
"true");
```

---

**Note:** Multiple threads are allowed to write to the database at the same time. However, there will be errors when multiple threads write to the same row of one MBO. Avoid writing to the same MBO row in your application.

---

### **Setting Up Connectivity**

Store connection information to the SAP Mobile Server data synchronization channel.

### **Setting Up the Synchronization Profile**

You can set SAP Mobile Server synchronization channel information by calling the synchronization profile's setter method. By default, this information includes the server host, port, domain name, certificate and public key that are pushed by the message channel during the registration process.

Settings are automatically provisioned from the SAP Mobile Server. The values of the settings are inherited from the application connection template used for the registration of the application connection (automatic or manual). You must make use of the connection and security settings that are automatically used by the Object API.

Typically, the application uses the settings as sent from the SAP Mobile Server to connect to the SAP Mobile Server for synchronization so that the administrator can set those at the application deployment time based on their deployment topology (for example, using Relay Server, using e2ee security, or a certificate used for the intermediary, such as a Relay Server Web server). See the *Applications* and *Application Connection Templates* topics in *System Administration*.

Set up a secured connection using the `ConnectionProfile` object.

1. Retrieve the synchronization profile object using the SAP Mobile Platform database's `GetSynchronizationProfile` method.

```
ConnectionProfile cp = SMP101DB.GetSynchronizationProfile();
```

2. Set the connection fields in the `ConnectionProfile` object.

```
cp.ServerName = "SAP_Mobile_Platform_Host";  
cp.PortNumber = 2481;  
cp.StreamParams.Trusted_Certificates = "rsa_public_cert.crt";  
cp.NetworkProtocol = "https";
```

### **Creating and Deleting a Device's Local Database**

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

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

```
SMP101DB.OpenConnection();
```

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

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

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

```
if (!SMP101DB.DatabaseExists())  
{  
    SMP101DB.CreateDatabase();  
}
```

3. When the local database is no longer needed, delete it by calling the generated database instance's `DeleteDatabase` method.

```
SMP101DB.DeleteDatabase();
```

### **Logging In**

Use online authentication with the server.

Authenticate the user for data synchronization by calling the generated database API `OnlineLogin` method.

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

```
ConnectionProfile syncProfile =
SMP101DB.GetSynchronizationProfile();
syncProfile.UserName = "user";
syncProfile.Password = "password";
SMP101DB.OnlineLogin();
```

### **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.

### **Turn Off API Logger**

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

```
SMP101DB.GetLogger().SetLogLevel(LogLevel.OFF);
```

### **Setting Up Callbacks**

When your application starts, it can register database and MBO callback listeners, as well as synchronization listeners.

Callback handler and listener interfaces are provided so your application can monitor changes and notifications from SAP Mobile Platform:

- The `Sybase.Mobile.IApplicationCallback` class is used for monitoring changes to application settings, messaging connection status, and application registration status.
- The `Sybase.Persistence.ICallbackHandler` interface is used to monitor notifications and changes related to the database. To register callback handlers at the package level, use the `RegisterCallbackHandler` method in the generated database class. To register for a particular MBO, use the `RegisterCallbackHandler` method in the generated MBO class.
- The `Sybase.Persistence.SyncStatusListener` class is used for debugging and performance measures when monitoring stages of a synchronization session, and can be used in the user interface to indicate synchronization progress.

### Setting Up Callback Handlers

Use the callback handlers for event notifications.

Use the `Sybase.Persistence.ICallbackHandler` API for event notifications including login for synchronization and replay. If you do not register your own implementation of the `Sybase.Persistence.ICallbackHandler` interface, the generated code will register a new default callback handler.

1. The generated database class contains a method called `RegisterCallbackHandler`. Use this method to install your implementation of `ICallbackHandler`.

For example:

```
SMP101DB.RegisterCallbackHandler(new MyCallbackHandler());
```

2. Each generated MBO class also has the same method to register your implementation of the `ICallbackHandler` for that particular type. For example, if `Customer` is a generated MBO class, you can use the following code:

```
Customer.RegisterCallbackHandler(new  
MyCustomerMBOCallbackHandler());
```

### Asynchronous Operation Replay

Upload operation replay records asynchronously.

When an application calls `SubmitPending` on an MBO on which a create, update, or delete operation is performed, an operation replay record is created on the device local database.

When `Synchronize` is called, the operation replay records are uploaded to the server. The method returns without waiting for the backend to replay those records. The `Synchronize` method downloads all the latest data changes and the results of the previously uploaded operation replay records that the backend has finished replaying in the background. If you choose to disable asynchronous operation replay, each `Synchronize` call will wait for the backend to finish replaying all the current uploaded operation replay records.

When SAP Mobile Platform does an update operation replay, if the primary key or foreign key of the MBO is generated by the EIS and the MBO's content coming from the device has no primary key or foreign key, the SAP Mobile Server loads the primary key or foreign key from the CDB to merge the incoming values with the CDB content so that a full row (graph) can be communicated to the EIS.

```
oneMBO mbo = new oneMBO();  
mbo.SetXX(xx);  
....  
mbo.Create();  
mbo.SubmitPending();  
mbo.SetXX(yy);  
....  
mbo.Update();  
mbo.SubmitPending();  
DBClass.Synchronize();
```



This feature is enabled by default. You can enable or disable the feature by setting the `asyncReplay` property in the synchronization profile. The following code shows how to disable asynchronous replay:

```
SMP101DB.GetSynchronizationProfile().AsyncReplay = false;
```

**When the application is connected**

(by `Application.StartConnection()` or `Application.RegisterApplication()`), it may receive background notifications and trigger a synchronize or other database operation. If you try to delete the database, you may receive database exceptions.

**Before deleting the database, stop the application connection**

(`Application.StopConnection()`).

You can specify an upload-only synchronization where the client sends its changes to the server, but does not download other changes from the server. This type of synchronization conserves device resources when receiving changes from the server.

```
public static void
BeginSynchronize(Sybase.Collections.GenericList<Sybase.Persistence.
ISynchronizationGroup> sgs, object context, bool uploadOnly)
```

**When asynchronous replay is enabled and the replay is finished, the `OnSynchronize` callback method is invoked with a `SynchronizationStatus` value of**

`SynchronizationStatus.ASYNC_REPLAY_COMPLETED`. Use this callback method to invoke a synchronize call to pull in the results, as shown in the following callback handler.

```
public virtual SynchronizationAction OnSynchronize(
    Sybase.Collections.GenericList<ISynchronizationGroup> groups,
    SynchronizationContext context)
{
    switch(context.Status)
    {
        case SynchronizationStatus.ASYNC_REPLAY_UPLOADED:
            LogMessage("AsyncReplay uploaded");
            break;
        case SynchronizationStatus.ASYNC_REPLAY_COMPLETED:
            // operation replay finished
            if (busy)
            {
                // if busy, don't do synchronize now
                return SynchronizationAction.CANCEL;
            }
            break;
        default:
            break;
    }
    return SynchronizationAction.CONTINUE;
}
```

### Synchronize Status Listener

Retrieve the synchronization status.

Synchronize Status Listener is mainly for debugging and performance measuring purposes to monitor stages of a synchronize session. It could also be used in UI for synchronization progress status. Below is a sample Synchronize Status Listener.

```
public class MySyncStatusListener : SyncStatusListener
{
    long start;

    public MySyncStatusListener()
    {
        start = DateTime.Now.Ticks;
    }

    public virtual bool ObjectSyncStatus(ObjectSyncStatusData
statusData)
    {
        long now = DateTime.Now.Ticks;
        long interval = now - start;
        start = now;
        String infoMessage;

        SyncStatusState syncState = statusData.State;

        switch (syncState) {
            case SyncStatusState.SyncStarting:
                infoMessage = "START [" + interval + "]";
                break;
            case SyncStatusState.ApplicationDataUploading:
                infoMessage = "DATA UPLOADING [" + interval + "] " +
                    statusData.CurrentMBO + ": (S>" +
                    statusData.SentBytes + " R<" +
                    statusData.ReceivedBytes + ")";
                break;
            case
SyncStatusState.ApplicationDataUploadingAcknowledgementReceived:
                infoMessage = "RECEIVING UPLOAD ACK [" + interval + "]";
                break;
            case SyncStatusState.ApplicationDataUploadingDone:
                infoMessage = "UPLOAD DONE [" + interval + "] " +
                    statusData.CurrentMBO + ": (S>" +
                    statusData.SentBytes + " R<" +
                    statusData.ReceivedBytes + ")";
                break;
            case SyncStatusState.ApplicationDataDownloading:
                infoMessage = "DATA DOWNLOADING[" + interval + "] " +
                    statusData.CurrentMBO + ": (S>" +
                    statusData.SentBytes + " R<" +
                    statusData.ReceivedBytes + ")";
                break;
            case SyncStatusState.ApplicationSyncDisconnecting:
```

```

        infoMessage = "DISCONNECTING [" + interval + "];
        break;
    case SyncStatusState.ApplicationSyncCanceled:
        infoMessage = "SYNC CANCELED [" + interval + "];
        break;
    case SyncStatusState.ApplicationDataDownloadingDone:
        infoMessage = "DATA DOWNLOADING DONE [" + interval + "];
        break;
    case SyncStatusState.ApplicationSyncDone:
        infoMessage = "DONE [" + interval + "];
        break;
    default:
        infoMessage = "STATE" + syncState + "[" + interval + "];
        break;
    }
    LogMessage(infoMessage);
    return false;
}
}

```

### **Connecting to the Device Database**

Establish a connection to the database on the device.

After completing the device registration, call the generated database's `OpenConnection` method to connect to the UltraLite/UltraLiteJ database on the device. If no device database exists, the `OpenConnection` method creates one.

### **Synchronizing Applications**

Synchronize package data between the device and the server.

The generated database provides you with synchronization methods that apply to either all synchronization groups in the package or a specified list of groups.

For information on synchronizing DOE-based applications, see *Message-Based Synchronization APIs*.

### **Configuring Data Synchronization Using SSL Encryption**

Enable SSL encryption by configuring the synchronization HTTPS port.

1. In the left navigation pane of SAP Control Center for SAP Mobile Platform, expand the **Servers** node and click the server name.
2. Click **Server Configuration**.
3. In the right administration pane, click the **Replication** tab.
4. Select **Secure synchronization port 2481** as the protocol used for synchronization, and configure the certificate properties. In the optional properties section, specify the security certificate file, the public security certificate file using the fully qualified path to the file, along with the password you entered during certificate creation.

### Nonblocking Synchronization

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

Subscribe to the package using synchronization APIs in the generated database class, specify the groups to be synchronized, and invoke the asynchronous synchronization method (BeginSynchronize).

1. Set the synchronization parameters if there are any.
2. Make a blocking synchronize call to SAP Mobile Server to pull in all MBO data:

```
SMP101DB.Synchronize();
```

3. List all customer MBO instances from the local database using an object query, such as FindAll, which is a predefined object query.

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

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

```
Customer cust = Customer.FindByPrimaryKey(100);  
cust.Address = "1 Sybase Dr.";  
cust.Phone = "9252360000";  
cust.Save(); //or cust.Update();
```

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

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

6. Use non-blocking synchronize call to upload the pending changes to the SAP Mobile Server. The previous replay results and new changes are downloaded to the client device in the download phase of the synchronization session.

```
ISynchronizationGroup sg = SMP101DB.GetSynchronizationGroup("my-  
synchronization-group");  
GenericList<ISynchronizationGroup> syncGroups = new  
GenericList<ISynchronizationGroup>();  
syncGroups.Add(sg);  
SMP101DB.BeginSynchronize(syncGroups, "my-context");
```

### Enabling Change Notifications

A synchronization group can enable or disable its change notifications.

By default, change notifications are disabled for synchronization groups. To enable change notifications, you must synchronize, then call the SynchronizationGroup object's SetEnableSIS method.

```
Sybase.Persistence.ISynchronizationGroup sg =
SMP101DB.GetSynchronizationGroup("PushEnabled");

if (!sg.EnableSIS)
{
    sg.EnableSIS = true;
    sg.Interval = 2; // 2 minutes
    sg.Save();
    SMP101DB.Synchronize("PushEnabled");
}
```

### **Specifying Personalization Parameters**

Use personalization parameters to provide default values used with synchronization, connections with back-end systems, MBO attributes, or EIS arguments. The `PersonalizationParameters` class is within the generated code for your project.

1. To instantiate a `PersonalizationParameters` object, call the generated database instance's `GetPersonalizationParameters` method:

```
PersonalizationParameters pp =
SMP101DB.GetPersonalizationParameters();
```

2. Assign values to the `PersonalizationParameters` object:

```
pp.PKCity = "New York";
```

3. Save the `PersonalizationParameters` value to the local database:

```
pp.Save();
```

---

**Note:** If you define a default value for a personalization key that value will not take effect, unless you call `pp.Save()`.

---

4. Synchronize the `PersonalizationParameters` value to the SAP Mobile Server:

```
SMP101DB.Synchronize();
```

### **Specifying Synchronization Parameters**

Use synchronization parameters within the mobile application to download filtered MBO data.

---

**Note:** The `getSynchronizationParameters` method has been deprecated.

---

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

1. List all the synchronization parameters.

```
Sybase.Collections.GenericList<CustomerSubscription> r =
Customer.GetSubscriptions();
```

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

```
CustomerSubscription sp = new CustomerSubscription();  
sp.Name = "example";  
Customer.AddSubscription(sp);
```

### 3. Synchronize to download the data:

```
SMP101DB.Synchronize();
```

## Subsequently Starting an Application

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

Starting an application on subsequent occasions:

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

---

**Note:** Once the application is registered, changes to any of the application connection properties do not take effect. To modify the connection properties, unregister the application, change the connection properties and then register again. Unregistering the application also removes the user from the server.

---

2. Set up the connection profile properties if needed for database location and tuning parameters.
3. Set up the synchronization profile properties if needed for SSL or a relay server.
4. Start the application connection to the server using the existing connection parameters and registration information.
5. Open the database connection.

You can open the database connection in parallel with starting the application connection to the server.

```
// Calls non-blocking StartConnection  
// This call will return immediately.  
application.StartConnection();  
  
// Open the device database connection while establishing  
// the messaging channel connection in the background  
SMP101DB.OpenConnection();  
  
// Once the device database connection has been opened, check  
// whether the messaging channel is connected using the  
// ApplicationCallback interface or the  
Application.GetConnectionStatus() API
```

## Accessing MBO Data

---

Use MBO object queries to retrieve lists of MBO instances, or use dynamic queries that return results sets or object lists.

### Object Queries

Use the generated static methods in the MBO classes to retrieve MBO instances.

1. To find all instances of an MBO, invoke the static `findAll` method contained in that MBO. For example, an MBO named `Customer` contains a method such as `public static Sybase.Collections.GenericList<MBO> FindAll()`.
2. To find a particular instance of an MBO using the primary key, invoke `MBO.FindByPrimaryKey(...)`. For example, if a `Customer` has the primary key "id" as int, the `Customer` MBO would contain the `public static Customer FindByPrimaryKey(int id)` method, which performs the equivalent of `Select x.* from Customer x where x.id = :id`.

If the return type is a list, additional methods are generated for you to further process the result, for example, to use paging. For example, consider this method, which returns a list of MBOs containing the specified city name:

```
Sybase.Collections.GenericList<SMP101.Customer>
FindByCity(string city, int skip, int take);
```

The `skip` parameter specifies the number of rows to skip, and the `take` parameter specifies the maximum number of rows to return.

### Dynamic Queries

Build queries based on user input.

Use the `Sybase.Persistence.Query` class to retrieve a list of MBOs.

1. Specify the **where** condition used in the dynamic query.

```
Query query = new Query();

AttributeTest aTest = new AttributeTest();

aTest.Attribute = "state";
aTest.Value = "NY";
aTest.Operator = AttributeTest.EQUAL;
query.TestCriteria = aTest;

SortCriteria sort = new SortCriteria();
sort.Add("lname", SortOrder.ASCENDING);
```

```
sort.Add("fname", SortOrder.ASCENDING);  
query.SetSortCriteria(sort);
```

2. Use the `FindWithQuery` method in the MBO to dynamically retrieve a list of MBOs according to the specified attributes.

```
GenericList<MyMBO> mbos = MyMBO.FindWithQuery(query);
```

3. Use the generated database's `executeQuery` method to query multiple MBOs through the use of joins.

```
Query query = new Query();  
query.Select("c.fname,c.lname,s.order_date,s.id");  
query.From("Customer", "c");  
query.Join("Sales_order", "s", "s.cust_id", "c.id");
```

```
AttributeTest ts = new AttributeTest();  
ts.Attribute = "lname";  
ts.Value = "Smith";  
ts.Operator = AttributeTest.EQUAL;
```

```
query.TestCriteria = ts;
```

```
QueryResultSet qrs = SMP101DB.ExecuteQuery(query);
```

```
while (qrs.Next())  
{  
    string fname = qrs.GetString(1);  
    string lname = qrs.GetString(2);  
    int orderId = qrs.GetInt(4);  
    // ...  
}
```

## MBOs with Complex Types

Mobile business objects are mapped to classes containing data and methods that support synchronization and data manipulation. You can develop complex types that support interactions with backend data sources such as SAP® and Web services. When you define an MBO with complex types, SAP Mobile Platform generates one class for each complex type.

Using a complex type to create an MBO instance.

1. Suppose you have an MBO named `SimpleCaseList` and want to use a complex data type called `AuthenticationInfo` to its `Create` method's parameter. Begin by creating the complex datatype:

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

2. Instantiate the MBO object:

```
SimpleCaseList newCase = new SimpleCaseList();  
newCase.Case_Type = "Incident";  
newCase.Category = "Networking";  
newCase.Create_Time = System.DateTime.Now;
```



3. Call the `create` method of the `SimpleCaseList` MBO with the complex type parameter as well as other parameters, and call `submitPending()` to submit the `create` operation to the operation replay record. Subsequent synchronizations upload the operation replay record to the SAP Mobile Server and get replayed.

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

## Relationships

The Object API supports one-to-one, one-to-many, and many-to-one relationships.

Navigate between MBOs using relationships.

1. Suppose you have one MBO named `Customer` and another MBO named `SalesOrder`. This code illustrates how to navigate from the `Customer` object to its child `SalesOrder` objects:

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

2. To filter the returned child MBO's list data, use the `Query` class:

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

3. For composite relationship, you can call the parent's `SubmitPending` method to submit the entire object tree of the parent and its children. Submitting the child MBO also submits the parent and the entire object tree. (If you have only one child instance, it would not make any difference. To be efficient and get one transaction for all child operations, it is recommended to submit the parent MBO once, instead of submitting every child).

If the primary key for a parent is assigned by the EIS, you can use a multilevel insert cascade operation to create the parent and child objects in a single operation without synchronizing multiple times. The returned primary key for the parent's `create` operation populates the children prior to their own creation.

The following example illustrates how to submit the parent MBO which also submits the child's operation:

```
Customer cust = Customer.FindById(101);
Sales_order order = new Sales_order();
order.Id = 1001;
order.Customer = cust;
order.Order_date = Date.ValueOf("1996-03-14");
order.Fin_code_id = "r1";
order.Region = "Eastern";
order.Sales_rep = 101;
order.Save(); // or order.create();
cust.Save();
cust.SubmitPending();
```

## Manipulating Data

---

Create, update, and delete instances of generated MBO classes.

You can create a new instance of a generated MBO class, fill in the attributes, and call the `Create` method for that MBO instance.

You can modify an object loaded from the database by calling the `Update` method for that MBO instance.

You can load an MBO from the database and call the `Delete` method for that instance.

## Creating, Updating, and Deleting MBO Records

---

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

You can call the `Create`, `Update`, and `Delete` methods for MBO instances.

---

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

---

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

```
Customer newcustomer = new Customer();
//Set the required fields for the customer
// ...

newcustomer.Create();
newcustomer.SubmitPending();
SMP101DB.Synchronize();
```

2. To update an existing MBO instance, retrieve the object instance through a query, update its attributes, and invoke its `Update` method, which causes the object to enter a pending state. Then call the MBO instance's `SubmitPending` method. Finally, synchronize with the generated database:

```
Customer customer = Customer.FindByPrimaryKey(myCustomerId) //find
by primary key
customer.City = "Dublin"; //update any field to a new value
customer.Update();
customer.SubmitPending();
SMP101DB.Synchronize();
```

3. To delete an existing MBO instance, retrieve the object instance through a query and invoke its `Delete` method, which causes the object to enter a pending state. Then call the MBO instance's `SubmitPending` method. Finally, synchronize with the generated database:

```
Customer newcustomer = new Customer();
Customer customer = Customer.FindByPrimary(myCustomerId) //find
by primary key
customer.Delete();
customer.SubmitPending();
SMP101DB.Synchronize();
```

For an object tree with MBOs in a composite (cascading) relationship, `SubmitPending` submits changes found in the entire hierarchy. If each MBO in the hierarchy has its own CRUD operations, the submitted object tree replays in this order:

- Create and Update: a preorder traversal, i.e. parent -> left child -> right child. That is, create the parent before the children.
- Delete: a postorder traversal, i.e. left child ->right child->parent.

Left and right in this context means from the first child in the children list to the last child. For a tree with multiple operation types, for example root (update) and two children (one create and one update) and each child has two children, the order of the operation is: root (update), child one(create), children of child one(create), children of child two (delete), child two (delete).

### Other Operations

Use operations other than create, update, or delete.

In this example, a customized operator is used to perform a sum operation.

1. Suppose you have an MBO that has an operator that generates a customized sum. Begin by creating an object instance and assigning values to its attributes, specifying the "Add" operation:

```
SMP101AddOperation op = new SMP101AddOperation(); //MBOName +
OperationName + "Operation"

op.Operand1 = 12;
op.Operand2 = 23;
op.Operator = "Add";
op.Save();
```

2. Call the MBO instance's `SubmitPending` method and synchronize with the generated database:

```
op.SubmitPending();
SMP101DB.Synchronize();
```

### Using SubmitPending and SubmitPendingOperations

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

Note that **SubmitPendingOperations** APIs are expensive. SAP recommends using the **SubmitPending** API with the MBO instance whenever possible.

### Database Classes

Submit pending operations for all entities in the package or synchronization group, cancel all pending operations that have not been submitted to the server, and check if there are pending operations for all entities in the package.

1. To submit pending operations for all pending entities in the package, invoke the generated database's `SubmitPendingOperations` method.

Note that **SubmitPendingOperations** APIs are expensive. SAP recommends using the **SubmitPending** API with the MBO instance whenever possible.

2. To submit pending operations for all pending entities in the specified synchronization group, invoke the generated database's `SubmitPendingOperations (string synchronizationGroup)` method.
3. To cancel all pending operations that have not been submitted to the server, invoke the generated database's `CancelPendingOperations` method.

### Generated MBOs

Submit pending operations for all entities for a given MBO type or a single instance, and cancel all pending operations that have not been submitted to the server for the MBO type or a single entity.

1. To submit pending operations for all pending entities for a given MBO type, invoke the MBO class' static `SubmitPendingOperations` method.

Note that **SubmitPendingOperations** APIs are expensive. SAP recommends using the **SubmitPending** API with the MBO instance whenever possible.

2. To submit pending operations for a single MBO instance, invoke the MBO object's `SubmitPending` method.
3. To cancel all pending operations that have not been submitted to the server for the MBO type, invoke the MBO class' static `CancelPendingOperations` method.
4. To cancel all pending operations for a single MBO instance, invoke the MBO object's `CancelPending` method.
5. For a single MBO, you must call the `refresh()` method of the MBO instance before you use this instance again.

```
customer.Create();
customer.submitPending();
// must call Refresh() here
customer.Refresh();
customer.Update();
customer.SubmitPending();
```

6. For related MBOs, you must call the `refresh()` method of the MBO instance before you use this instance again, even if the MBO's child or parent has called `SubmitPending`.

## Shutting Down the Application

---

Shut down an application and clean up connections.

### Closing Connections

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

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

```
app.StopConnection(<timeout_value>);
```
2. Use the `CloseConnection` method to close all database connections for this package and release all resources allocated for those connections. This is recommended to be part of the application shutdown process.

## Uninstalling the Application

---

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

### Deleting the Database and Unregistering the Application

Delete the package database, and unregister the application.

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

```
app.UnregisterApplication(<time out value>);
```
2. To delete the package database, call the generated database's `DeleteDatabase` method.  

```
SMP101DB.DeleteDatabase();
```

## Recovering From SAP Mobile Server Failures

---

Add application code to check for and recover from from SAP Mobile Server failures.

It is highly recommended that you add a catch call to all synchronize methods (`synchronize()`, `beginSynchronize()`, and so on) within your applications to allow the application to recover if SAP Mobile Server fails and needs to be restored from an older database. If not, you may have to reinstall the application manually for all users so they can resynchronize with SAP Mobile Server.

See *Restoring from an Older Backup Database File (Data Loss)* in the *System Administration Guide* for information about SAP Mobile Server recovery.

As a best practice, and not included in these examples, application developers should include code that informs mobile application users about:

- What is going to happen (for example, reregistering, recreating the local database, and so on). And,
- The reason for the action (for example, lost registration, server is restored, and so on).

And prompt them for confirmation before executing the code.

1. The server is restored to a state prior to client registration, the Application Callback class reregisters:

- a) Windows Mobile, Win32 RBS client:

```
public class MyApplicationCallback :
Sybase.Mobile.DefaultApplicationCallback
{
    bool callFlag = false;
    public override void OnConnectionStatusChanged(int
connectionStatus,
    int errorCode, string errorMessage)
    {
        //Error 580 indicates that the application appears
registered to the device but
// not in the server. If true, the application reregisters
with the server
        if (errorCode == 580 && callFlag)
        {
            callFlag = true;
            Thread registerThread = new Thread(new
ThreadStart(reregister));
            registerThread.Name = "reregister";
            registerThread.Start();
        }
    }
    private void reregister()
    {
        System.Console.WriteLine("RegisterApplication...");
        Application.GetInstance().RegisterApplication();
        System.Console.WriteLine("RegisterApplication done");
    }
}
```

- b) Windows MBS/DOEC client:

```
public class MyApplicationCallback :
Sybase.Mobile.DefaultApplicationCallback
{
    bool callFlag = false;
    public override void OnConnectionStatusChanged(int
connectionStatus,
    int er rorCode, string errorMessage)
    {
        //Error 580 indicates that the application appears registered
```

```

to the device but
    // not in the server. If true, the application reregisters
with the server

        if (errorCode == 580 && !callFlag)
        {
            System.Console.WriteLine("OnConnectionStatusChanged:
got errorCode580");
            callFlag = true;
            Thread registerThread = new Thread(new
ThreadStart(reregister));
            registerThread.Name = "reregister";
            registerThread.Start();
        }
    }
private void reregister()
{
    System.Console.WriteLine("RegisterApplication...");
    Application APP1 = Application.GetInstance();
    APP1.ApplicationCallback = new MyApplicationCallback();

    ConnectionProfile cp =
MyDatabase.GetSynchronizationProfile();
    cp.AsyncReplay = false;
    cp.EnableTrace(false);
    cp.Save();

    MyDatabase.SetApplication(APP1);
    Console.WriteLine("####SetApplication OK...");
    ConnectionProperties props = APP1.ConnectionProperties;
    props.ServerName = Host;
    props.PortNumber = Port;

    LoginCredentials login = new LoginCredentials(Username,
Credential);
    props.LoginCredentials = login;
    Console.WriteLine("####RegisterApplication...");

    Application.GetInstance().RegisterApplication();

    while (APP1.ConnectionStatus !=
ConnectionStatus.CONNECTED)
    {
        Thread.Sleep(1000);
        Console.WriteLine("Wait for connected, sleep 1
second");
    }
    System.Console.WriteLine("RegisterApplication done");
}
}
    
```

2. The server is restored to a state to which the client is registered and synchronized, but the server's state is earlier than the client's current synchronized state. When the client synchronizes (either synchronous call, `synchronize()`, or asynchronous call, `beginSynchronize()`), the client gets an error

Sybase.Persistence.SynchronizeException.SQLE\_UPLOAD\_FAILED\_AT\_SERVER. The client application then needs to catch this exception in order to recover on the client side.

a) Windows Mobile, Win32 RBS client needs to start connection again and subscribe:

---

**Note:** When the Windows Mobile Win32 RBS client calls `DB.Synchronize()` with the callback handler, the "SQLE\_UPLOAD\_FAILED\_AT\_SERVER" error may appear in both callback's `OnSynchronize()` method, and an exception caught in the `DB.Synchronize()` call. When handling the client side recovery from `OnSynchronize()`, be very careful when coding (threading), otherwise the exception nested synchronize call is detected because of synchronize method is invoked in `onSynchronize` callback may be encountered. This client side recovery example catches the exception in `DB.Synchronize()`.

---

```
catch (Exception ex)
{
    if (ex is Sybase.Persistence.SynchronizeException)
    {
        Sybase.Persistence.SynchronizeException syncException
        =
            (Sybase.Persistence.SynchronizeException)ex;
        //If an error during synchronization is encountered close the
        connection,
        //address the exception, delete the client database,
        reregister/subscribe/resync
        if (syncException.ErrorCode ==

Sybase.Persistence.SynchronizeException.SQLE_UPLOAD_FAILED_AT_
SERVER)
        {
            TestDB.CloseConnection();
            TestDB.DeleteDatabase();
            System.Console.WriteLine("Sync again");
            TestDB.RegisterCallbackHandler(_ch);
            TestDB.Subscribe();
            TestDB.Synchronize();
        }
    }
}
```

b) Windows Mobile/Win32 asynchronous client:

```
public void Synchronize()
{
    try
    {
        Console.WriteLine("-- asynchronize --");
        AsyncCallbackHandlerHARA callback = new
AsyncCallbackHandlerHARA();
        GenericList<ISynchronizationGroup> syncGroups = new
GenericList<ISynchronizationGroup>();
    }
}
```



```

syncGroups.Add(AsyncOpReplayDB.GetSynchronizationGroup("default"));
        callback.userContext = Environment.TickCount + "";
        AsyncOpReplayDB.RegisterCallbackHandler(callback);
        AsyncOpReplayDB.BeginSynchronize(syncGroups,
callback.userContext);
        Console.WriteLine("BeginSynchronize");
        int waitCount = 0;
        while (!callback.asyncDone())
        {
            if (waitCount++ > maxWaitTime)
            {
                break;
            }
            try
            {
                Thread.Sleep(1000);
            }
            catch (Exception e)
            {
                Console.WriteLine("exception=" +
e.ToString());
            }
            if (callback.errorMessage != null)
            {
                Console.WriteLine("callback errorMessage=" +
callback.errorMessage);
                throw new Exception(callback.errorMessage);
            }
            callback.userContext = null;
            Console.WriteLine("-- asynchronize done --");
        }
        catch (Exception ex)
        {
            Console.WriteLine("Exception message is:
{0}", ex.Message);
        }
    }
}
class AsyncCallbackHandlerHARA :
DefaultCallbackHandler
{
    private volatile bool asyncCompleted = false;
    private volatile bool asyncUploaded = false;
    public volatile String userContext = null;
    public volatile String errorMessage = null;
    public bool asyncDone()
    {
        return asyncCompleted;
    }
    public void RecoverClientDatabase()
    {
        System.Console.WriteLine("RecoverClientDatabase start");
    }
}

```

```

        AsyncOpReplayDB.CloseConnection();
        AsyncOpReplayDB.DeleteDatabase();
        AsyncOpReplayDB.CreateDatabase();
        AsyncCallbackHandlerHARA callback = new
AsyncCallbackHandlerHARA();
        GenericList<ISynchronizationGroup>
syncGroups = new
        GenericList<ISynchronizationGroup>();
syncGroups.Add(AsyncOpReplayDB.GetSynchronizationGroup("default"));
        callback.userContext =
Environment.TickCount + "";
AsyncOpReplayDB.RegisterCallbackHandler(callback);

AsyncOpReplayDB.BeginSynchronize(syncGroups,
callback.userContext);

System.Console.WriteLine("RecoverClientDatabase done");
    }
    public override SynchronizationAction
OnSynchronize (GenericList<ISynchronizationGroup> groups,
Sybase.Persistence.SynchronizationContext
context)
    {
        Console.WriteLine("--
AsyncCallbackHandlerHARA OnSynchronize --");
        Exception ex = context.Exception;
        if (ex is
Sybase.Persistence.SynchronizeException)
        {
            Sybase.Persistence.SynchronizeException syncException =
(Sybase.Persistence.SynchronizeException)ex;
            if (syncException.ErrorCode ==
Sybase.Persistence.SynchronizeException.SQLE_UPLOAD_FAILED_AT_
SERVER)
                {
                    RecoverClientDatabase();
                }
            }
            ...
        return SynchronizationAction.CONTINUE;
    }
}

```

3. The following example illustrates a MBS/DOEC client where the client waits long enough before syncing to allow the server to recover to a restored state. The client makes calls according to the state instead of using error handling code. The server is restored to a state where the client is registered but not subscribed. The client needs to unsubscribe and resubscribe.

```

Public void sync() {
...

```

```

_rh = new TextResponseHandler();
MyDatabase.RegisterCallbackHandler(_rh);
oqe.Create();

MyDatabase.SubmitPendingOperations();

//at this point, server is restored to the state client registered
and did not subscribe

//client wake up and new operations
if (APP.ConnectionStatus != ConnectionStatus.CONNECTED)
{
    APP.StartConnection(100);
    Thread.Sleep(2000);
}
if(MyDatabase.IsSubscribed())
{
    MyDatabase.Unsubscribe();
    // waiting for dowload data complete
    _rh.WaitForMessage("UnsubscribeFailure",
"UnsubscribeSuccess");
    System.Console.WriteLine("Unsubscribe is called");
    MyDatabase.Subscribe();
    System.Console.WriteLine("Subscribe is called");
    _rh.WaitForMessage("SubscribeSuccess");
}
oqe = new ASADataTypesOQE();
...
}

```

- 4.** The following example illustrates a MBS/DOEC client where the client waits long enough before synching to allow the server to recover to a restored state. The client makes calls according to the state, instead of using error handling code. The MBS/DOEC Client needs to subscribe if the server is restored to a state where the client is registered and subscribed.

```

oqe.Create();
MyDatabase.SubmitPendingOperations();
lst = ASADataTypesOQE.FindAll();
Console.WriteLine("ASADataTypesOQE has records: " + lst.Size());
foreach (ASADataTypesOQE itm in lst)
{
    Console.WriteLine("ID: " + itm.Id + " | Name: " +
itm.Ctext);
}
//at this point, server is restored to the state client registered
and subscribed

//client wake up and new operations
if (!MyDatabase.IsSubscribed())
{
    MyDatabase.Subscribe();
    //waiting for dowload data complete
    _rh.WaitForMessage("ImportSuccess");
}

```



# Testing Applications

Test native applications on a device or simulator.

For additional information about testing applications, see these topics in the Mobile Application Life Cycle collection:

- *Recommended Test Methodologies*
- *Best Practices for Testing Applications on a Physical Device*

## Testing an Application Using a Simulator

---

Run and test the application on a simulator and verify that the application automatically registers to the SAP Mobile Server using the default application connection template.

1. In Visual Studio, choose **Tools > Device Emulator Manager**.
2. Scroll down to the entry for **Windows Mobile 6 Professional**, right-click that entry and select **Connect**.
3. Right-click the entry for **Windows Mobile 6 Professional** again and select **Cradle**.
4. In the Solution Explorer of Visual Studio, right-click the C# project and select **Deploy**.
5. In the Windows Mobile emulator go to **Start > Programs > File Explorer**.
6. Navigate to the installed path by going in **Program Files > > <Application>**.
7. Start the application.
8. In SAP Control Center, verify that the application connection was created in **Applications > Application Connections**.  
When the application has successfully registered, the application connection displays a value of zero in the Pending Items column. The Pending Items column is used only for messaging applications.
9. Test the functionality of the application. Use debug tools as necessary, setting breakpoints at appropriate places in the application.

## Client-Side Debugging

---

Identify and resolve client-side issues while debugging the application.

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

- SAP Mobile Server connection failed - use your device browser to check the connectivity of your device to the server.

## Testing Applications

- Data does not appear on the client device - check if your synchronization and personalization parameters are set correctly. If you are using queries, check if your query conditions are correctly constructed and if the device data match your query conditions.
- Physical device problems, such as low memory - implement `ApplicationCallback.onDeviceConditionChanged` to be notified if device storage gets too low, or recovers from an error.

To find out more information on the device client side:

- If you have implemented debugging in your generated or custom code (which SAP recommends), turn on debugging and review the debugging information. See the API Reference information about using the `Logger` class to add logs to the client log record and synchronize them to the server (viewable in SAP Control Center).
- Check the log record on the device. Use the `DatabaseClass.GetLogRecords` (`Sybase.Persistence.Query`) or `Entity.GetLogRecords` methods.

This is the log format

```
level,code,eisCode,message,component,entityKey,operation,requestId,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` – SAP Mobile Server administration codes.
  - Synchronization codes:
    - 200 – success.
    - 500 – failure.
- `eisCode` – maps to HTTP error codes. If no mapping exists, defaults to error code 500 (an unexpected server failure).
- `message` – the message content.
- `component` – MBO name.
- `entityKey` – MBO surrogate key, used to identify and track MBO instances and data.
- `operation` – operation name.
- `requestId` – operation replay request ID or messaging-based synchronization message request ID.
- `timestamp` – message logged time, or operation execution time.

- If you have implemented `ApplicationCallback.OnConnectionStatusChanged` for synchronization in the `CallbackHandler`, the connection status between the SAP Mobile Server and the device is reported on the device. See the `CallbackHandler` API reference information. The device connection status, device connection type, and connection error message are reported on the device:
  - 1 – current device connection status.
  - 2 – current device connection type.
  - 3 – connection error message.
- For other issues, you can turn on SQLTrace trace on the device side to trace Client Object API activity. To enable SQLTrace using the `ConnectionProfile`'s `enableTrace` API:

```
/// To enable SQL trace with values also displayed
SMP101DB.GetConnectionProfile().EnableTrace(true, true);
```

## Server-Side Debugging

---

Identify and resolve server-side issues while debugging the application.

Problems on the SAP Mobile Server side may cause device client problems:

- The domain or package does not exist. If you create a new domain, with a default status of disabled, it is unavailable until enabled.
- Authentication failed for the application user credentials.
- The operation role check failed for the synchronizing user.
- Back-end authentication failed.
- An operation failed on the remote, replication database back end, for example, a table or foreign key does not exist.
- An operation failed on the Web Service, REST, or SAP® back end.

To find out more information on the SAP Mobile Server side:

- Check the SAP Mobile Server log files.
- For message-based synchronization mode, you can set the log level to DEBUG to obtain detailed information in the log files:
  1. Set the log level using SAP Control Center. See *SAP Control Center for SAP Mobile Platform > Administer > SAP Mobile Server > Server Log > SAP Mobile Server Runtime Logging > Configuring SAP Mobile Server Log Settings*.

---

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

---

- Obtain DEBUG information for a specific device:
  - In the SCC administration console:
    1. Set the DEBUG level to a higher value for a specified device:

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

---

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

---

## Improve Synchronization Performance by Reducing the Log Record Size

---

Improve synchronization performance and free SAP Mobile Server resources by deleting log records from SAP Mobile Server and the client when no longer needed.

A large log record table can negatively impact client synchronization performance. Each package contains a single log record table that consists of:

- **SAP Mobile Server operation replay logs** – downloaded to the device when the application synchronizes. SAP Mobile Server generates a log record if the operation replay fails, or succeeds but results in a warning.
- **Client logs generated by the application** – uploaded from the device to SAP Mobile Server for audit and logging purposes.

If the application and SAP Mobile Server do not delete these log records, the log record table continues to grow.

Unrestricted growth of the log record table eventually affects synchronization performance. You can view client log records from SAP Control Center; however, this displays only active log records (that is, those that have not been logically deleted). A logically deleted log record is marked for deletion but retained until the application downloads the delete record and deletes the copy from the device. Once SAP Mobile Server confirms that the application has downloaded the delete, the inactive log record can be physically removed from SAP Mobile Server.

## Determining the Log Record Size

---

Use Sybase® Central to query the database of a given SAP Mobile Server to determine the size of the log record.

### Prerequisites

SAP Mobile Platform services must be running and at least one Mobile Application project deployed to SAP Mobile Server.



## Task

1. Launch Sybase Central (`scjview.exe`) to manage SQL Anywhere and Ultralite databases.

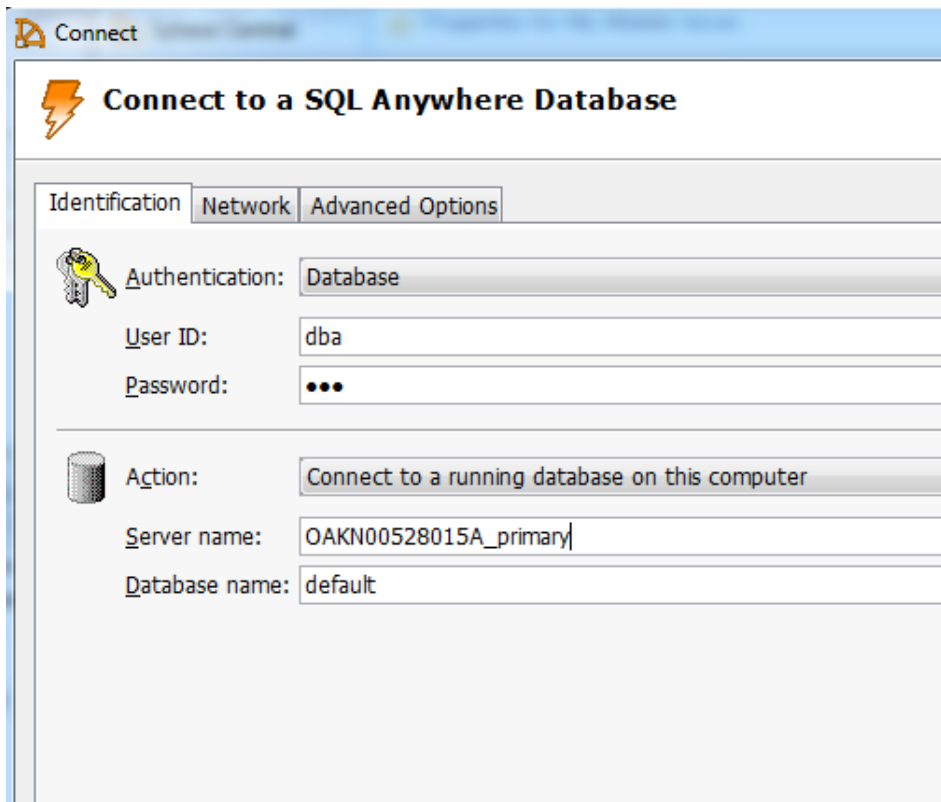
The default installation location of the Sybase Central executable is `SMP_HOME \Servers\SQLAnywhere12\BIN32\scjview.exe`.

2. From Sybase Central connect to the database server by selecting **Connections > Connect with SQL Anywhere 12**.

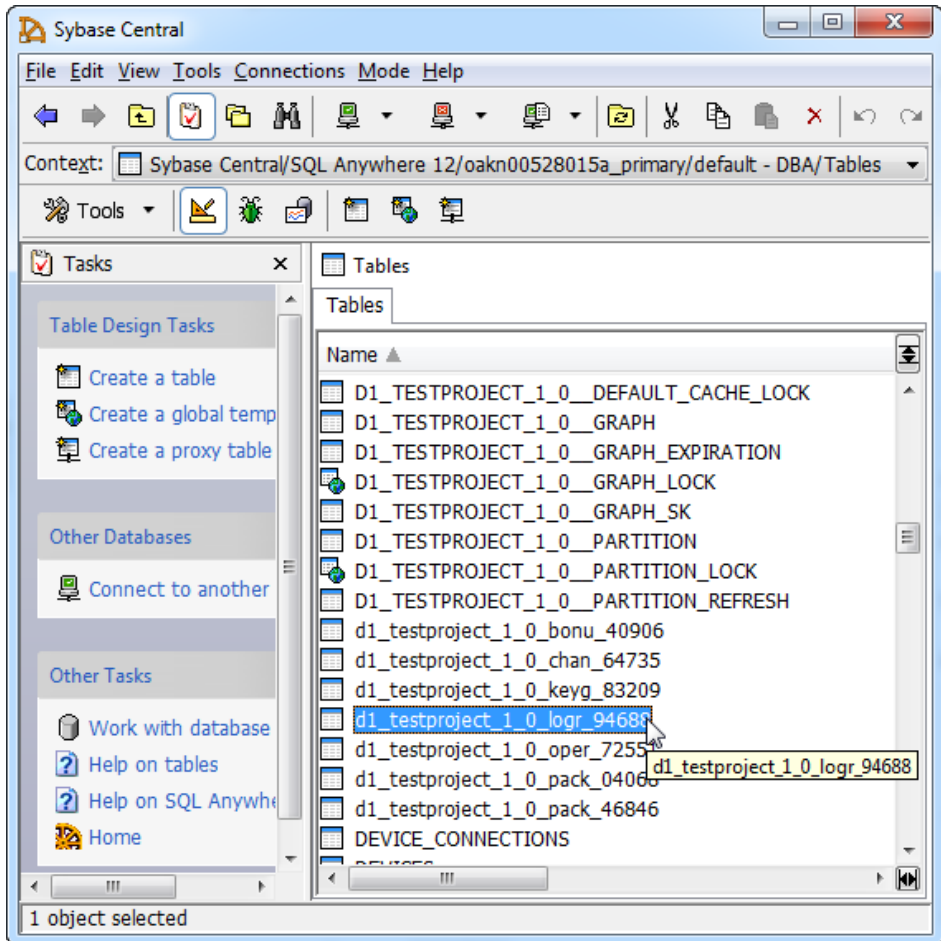
3. Provide connection details and click **Connect**.

For example, select **Connect to a running database on this computer** and enter:

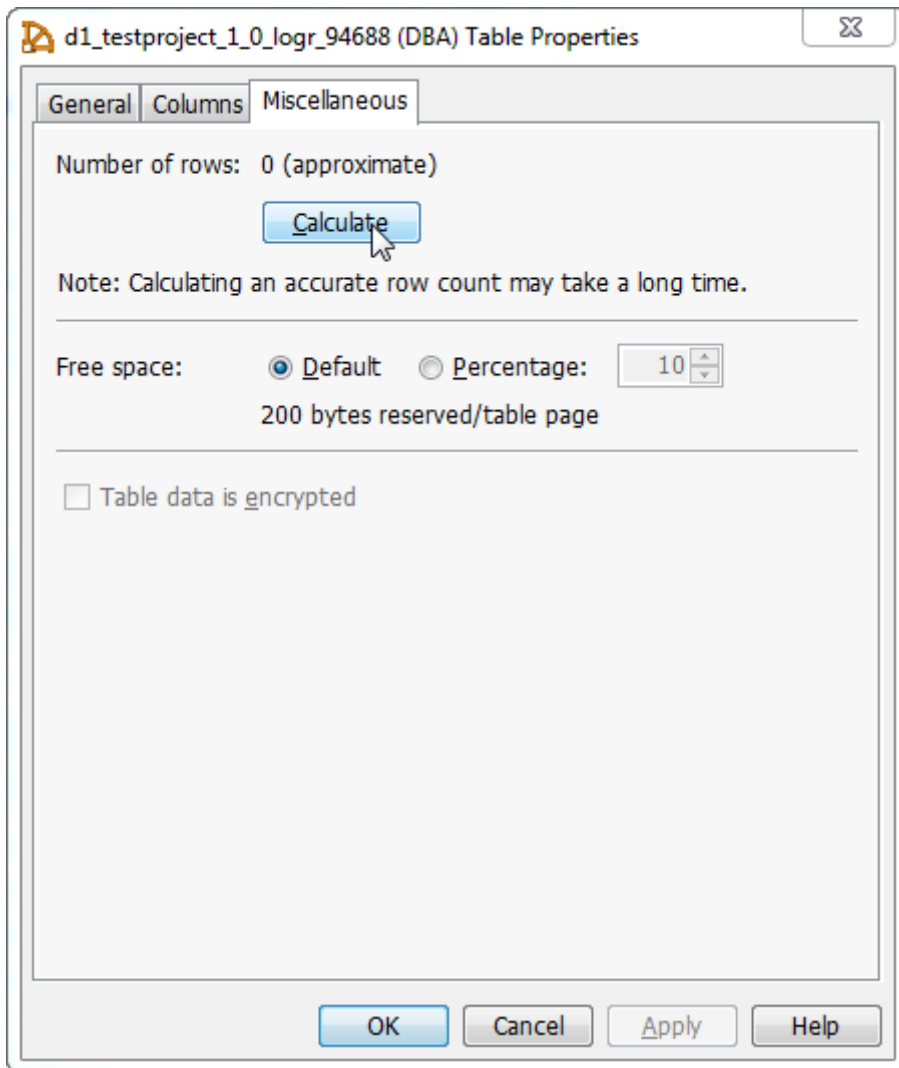
- **User ID and Password** – dba and sql respectively
- **Server name** – `hostName_primary`
- **Database name** – default



4. Double-click the **Tables** folder and search for the log record table. The log record name is typically `packageName_logr...` where `packageName` is the name of the deployed package.



5. Right-click the log record table and select **Properties**.
6. In the Properties dialog, select the **Miscellaneous** tab, then click **Calculate**.



The number returned includes logically deleted rows. The returned number of rows depends on the number of application users of the package, and the retention window setting. As a general guideline, the number of rows should be fewer than 10,000.

## **Reducing the Log Record Size**

Use SAP Control Center to delete log record entries by setting a date range window.

The SAP Mobile Server does not remove any logically deleted rows until it receives confirmation that the device hosting the application has synchronized after the record is logically deleted from SAP Mobile Server.

## Testing Applications

1. Clean up the client log data:
  - a) Expand **Domains > default > Packages**.
  - b) Select *packageName* then select the **Client Log** tab.
  - c) Select **Clean**, then enter starting and ending dates.

The LOGICAL\_DEL flag is set to true for records within the range.

---

**Note:** Allow time for clients to synchronize. Logically deleted records are retained until the client synchronizes and downloads the delete records that clean up the client database. The length of time to wait for synchronization to complete depends on the clients' activities.

---

- d) Click **OK** to clean the client log data.
2. Clean the logically deleted records from SAP Mobile Server:
    - a) Select the **General** tab.
    - b) Select **Error Cleanup**.

This starts a cleanup task that asynchronously removes all logically deleted records from clients that have performed a synchronization after the time specified in the Clean operation.

For example, if the Clean operation is performed at 1:00am on Feb 27, all clients that synchronize after that time have their records physically removed. As a result, it takes time to reduce the size of the log record table.

---

**Note:** Clean up the client log data (step one) during periods of low client activity: when a single transaction processing a large log record table is active, client synchronization is blocked, degrading client responses and performance. As a best practice, once the log record table has been cleaned to a reasonable size, schedule the clean/error cleanup tasks on a daily basis.

---

# Localizing Applications

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

## Generating Resource Files

---

Generate resource files for localizing controls and captions for each locale.

1. In Visual Studio, select the Form you want to localize, and open it in View Designer.
2. In the Properties window, set the form's Localizable property to true.
3. Set the form's Language property to the language of the locale. For example, Chinese (Simplified).
4. Set the text for the form and the Text property for all buttons to the appropriate text for the locale.

---

**Note:** You can resize a button to accommodate a longer string, if necessary. The button size is persisted in the resource file.

---

5. Repeat steps 3 and 4 to add more languages, as required.
6. Save and build the solution.
7. Expand `Customers.cs`.

The resource files appear under `Customers.cs`. `Customers.resx` is the resource file for the default culture, which is built into the main assembly. Resource files for other locales include a language code, and, optionally, a country code. Language and country codes are specified in ISO-639 and ISO-3166, respectively.

For example, `Customers.zh-CHS.resx` is the resource file for Chinese as spoken in Chinese (Simplified).

## Adding a Resource File Template and String Variables

---

Add a resource file template to the project and then edit the template with the XML Designer to define localizable strings that appear in dialog boxes and error messages.

1. In the Project menu, select **Add New Item**.
2. In the Templates box, select the Resources File template. Enter the file name "SampleApp.resx" in the Name box.

The resources file template contains resources that are accessed when the application cannot find resources appropriate to the user interface culture.

The file gets added to your project in Solution Explorer and automatically opens in the XML Designer in Data view.

3. In the Data pane, select an empty row and enter "strDatabaseDeleted" in the name column and "Database deleted" in the value column.
4. Add rows for other strings, as required.

---

**Note:** You do not need to specify the type or mimetype for a string; strings are used for objects. The `type` specifier holds the data type of the object being saved. The `MIME type` specifier holds the base type (base64) of the binary information stored, if the object consists of binary data.

---

5. In the File menu, select **Save** SampleApp.resx.
6. Repeat this procedure to create resource files for the other languages.  
For example, we may create an additional resource file named "SampleApp.zh-CHS.resx". That file is to contain resources that are specific to Chinese as spoken in Chinese (Simplified).

## Localizing the Application Code

---

Provide access to the manually added resources.

1. In the Code Editor, import the `System.Resources` namespace at the beginning of the code module.

```
using System.Resources;
```

2. Add the following code at the beginning of the class. The `ResourceManager` constructor takes two arguments. The first is the root name of the resources — that is, the name of the resource file without the culture and `.resx` suffixes. The second argument is the main assembly.

```
ResourceManager LocRM = new  
ResourceManager("SampleApp.SampleApp",  
typeof(Customer).Assembly);
```

3. Modify the code to use the `ResourceManager.GetString()` method to replace hard-coded strings. For example, modify:

```
AddString("Database deleted");
```

to:

```
AddString(LocRM.GetString("strDatabaseDeleted"));
```

---

**Note:** By default, the `ResourceManager` object is case-sensitive. If you want to perform case-insensitive lookups you can set the resource manager's `IgnoreCase` property to `true`. However, for performance reasons, specify the correct case for your resource names.

---

4. Repeat the localization procedures for `CustomerDetails.cs`.

## Validating the Localization Changes

---

Test that your changes appear in your application.

1. Launch the application on a device or emulator that supports the language for the locale.
2. Launch your application and verify that it is localized.  
The application appears with the appropriate user interface language for the operating system of the device or emulator.





# Packaging Applications

Package applications according to your security or application distribution requirements.

You can package all libraries into one package. This packaging method provide more security since packaging the entire application as one unit reduces the risk of tampering of individual libraries.

You may package and install modules separately only if your application distribution strategy requires sharing libraries between SAP Mobile Platform applications.

## Signing

---

Code signing is required for applications to run on physical devices.

You can run unsigned modules, which will result in warnings.

## Compiling an Application in Visual Studio

---

Deploy applications to a Device Emulator or connected device.

1. To deploy an application from Visual Studio, compile the project and deploy the application to the emulator or physical device.
2. If you are using a Device Emulator, define a shared folder and copy the file in that folder from your machine so the Emulator can access it.
3. Using Windows and a connected device, use the Virtual folder on your machine to copy the application's `.cab` file to the device or memory card.



# Client Object API Usage

The SAP Mobile Platform Client Object API consists of generated business object classes that represent mobile business objects (MBOs) that are designed and built in the SAP Mobile WorkSpace development environment. Device applications use the Client Object API to retrieve data and invoke mobile business object operations.

Refer to these sections for more information on using the APIs described in *Developer Guide: Windows and Windows Mobile Object API Applications* > *Developing the Application Using the Object API*.

## Client Object API Reference

---

Use the SAP Mobile Platform Client Object API C# docs as a Client Object API reference.

Review the reference details in the Client Object API documentation, located in `SMP_HOME\MobileSDK23\ObjectAPI\apidoc`.

There is a subdirectory for `cs`.

---

**Note:** Due to an Ultralite limitation, the first client object API call must be on the main thread in the application.

---

## Application APIs

---

The `Application` class, in the `Sybase.Mobile` namespace, manages mobile application registrations, connections and context.

---

**Note:** SAP recommends that you use the Application API operations with no `timeout` parameter, and register an `ApplicationCallback` to handle completion of these operations.

---

### Application

Methods or properties in the `Application` class.

#### **GetInstance**

Retrieves the `Application` instance for the current mobile application.

#### **Syntax**

```
public static Application GetInstance()
```

### **Returns**

GetInstance returns a singleton Application object.

### **Examples**

- **Get the Application Instance**

```
Application app = Application.GetInstance();
```

### **GetInstance (String)**

Return a named Application instance for the current mobile application. Named application instances are intended to support performance tests, where multiple instances of an application are running simultaneously as separate processes on the same computer. This operation is currently only provided for Windows C# applications

### **Syntax**

```
public static Application GetInstance(string instanceID)
```

### **Returns**

GetInstance returns a singleton Application object.

### **Examples**

- **Get the Application Instance**

```
Application app = Application.GetInstance(123);
```

### **ApplicationIdentifier Property**

Identifier for the current application. The property must be set before calling StartConnection, RegisterApplication or UnregisterApplication.

### **Syntax**

```
public string ApplicationIdentifier { get; set; }
```

### **Examples**

- **Get the Status**

```
application.ApplicationIdentifier = "SMP101";
```

### **RegistrationStatus property**

Retrieves the current status of the mobile application registration.

**Syntax**

```
public int RegistrationStatus { get; set; }
```

**Returns**

RegistrationStatus returns one of the values defined in the RegistrationStatus class.

```
public class RegistrationStatus {
    public static final int REGISTERED = 203;
    public static final int REGISTERING = 202;
    public static final int REGISTRATION_ERROR = 201;
    public static final int UNREGISTERED = 205;
    public static final int UNREGISTERING = 204;
}
```

**Examples**

- **Get the Registration Status** – Registers the application if it is not already registered.

```
if (app.RegistrationStatus != RegistrationStatus.REGISTERED)
{
    // If the application has not been registered to the server,
    // register now
    app.RegisterApplication();
}
else
{
    // start the connection to server
    app.StartConnection();
}
```

**RegisterApplication**

Creates the registration for this application and starts the connection. This method is equivalent to calling RegisterApplication(0).

**Syntax**

```
public void RegisterApplication()
```

**Parameters**

None.

**Examples**

- **Register an Application** – Start registering the application and return at once.

```
app.RegisterApplication();
```

### **Usage**

You must set up the `ConnectionProperties` and `ApplicationIdentifier` before you can invoke `registerApplication`.

The maximum length of the Application ID is 64 characters. The total length of the Application Connection ID cannot exceed 128 characters. For Windows platforms, the Application Connection ID format is: `deviceId$`

`{_instanceId}_applicationId`, if the `instanceId` is given; otherwise the format is `deviceId__applicationId`. For Windows Mobile, the Application Connection ID format is `deviceId__applicationId`. The `instanceId` separator is a single underscore (`_`), while the `applicationId` separator is two underscores.

```
Application app = Application.GetInstance();

// set Application ID - need to match as the server side Application
ID
app.ApplicationIdentifier = "SMP101";
ConnectionProperties props = app.ConnectionProperties;
props.ServerName = "server.mycompany.com";
props.PortNumber = 5001;
LoginCredentials loginCred = new LoginCredentials("supAdmin",
"supPwd");
props.LoginCredentials = loginCred;

SMP101DB.SetApplication(app);

if (app.RegistrationStatus != RegistrationStatus.REGISTERED)
{
// If the application has not been registered to the server,
// register now
app.RegisterApplication();
}
```

### **RegisterApplication (int timeout)**

Creates the registration for this application and starts the connection. An `ApplicationTimeoutException` is thrown if the method does not succeed within the number of seconds specified by the timeout.

If a callback handler is registered and network connectivity is available, the sequence of callbacks as a result of calling `RegisterApplication` is:

```
OnRegistrationStatusChanged(RegistrationStatus.REGISTERING, 0, "")
OnConnectionStatusChanged(ConnectionStatus.CONNECTING, 0, "")
OnConnectionStatusChanged(ConnectionStatus.CONNECTED, 0, "")
OnRegistrationStatusChanged(RegistrationStatus.REGISTERED, 0, "")
```

When the `connectionStatus` of `CONNECTED` has been reached and the application's `applicationSettings` have been received from the server, the application is now in a suitable state for database subscriptions and/or synchronization. If a callback handler is registered and

network connectivity is unavailable, the sequence of callbacks as a result of calling `registerApplication` is:

```
OnRegistrationStatusChanged(RegistrationStatus.REGISTERING, 0, "")
OnRegistrationStatusChanged(RegistrationStatus.REGISTRATION_ERROR,
code, message)
```

In such a case, the registration process has permanently failed and will not continue in the background. If a callback handler is registered and network connectivity is available for the start of registration but becomes unavailable before the connection is established, the sequence of callbacks as a result of calling `registerApplication` is:

```
OnRegistrationStatusChanged(RegistrationStatus.REGISTERING, 0, "")
OnConnectionStatusChanged(ConnectionStatus.CONNECTING, 0, "")
OnConnectionStatusChanged(ConnectionStatus.CONNECTION_ERROR, code,
message)
```

In such a case, the registration process has temporarily failed and will continue in the background when network connectivity is restored.

As a best practice, if a timeout exception occurs in `RegisterApplication` or `StartConnection`, the application should wait for the appropriate callback, and optionally add a user message to the application, "please wait" for example, instead of closing the application. This prevents a build up of start up requests by needlessly restarting the application which can adversely affect performance.

Wait for the application callback, such as `OnConnectionStatusChanged()` if `ApplicationTimeoutException` is encountered when calling `RegisterApplication` (int timeout), instead of closing the application. This allows the application code to catch `ApplicationTimeoutException` and does not throw an exception.

## Syntax

```
public void RegisterApplication(int timeout)
```

## Parameters

- **timeout** – Number of seconds to wait until the registration is created. If the the timeout is greater than zero and the registration is not created within the timeout period, an `ApplicationTimeoutException` is thrown (the operation might still be completing in a background thread). If the timeout value is less than or equal to 0, then this method returns immediately without waiting for the registration to finish (a non-blocking call). If the timeout value is less than or equal to 0, then this method returns immediately without waiting for the registration to finish (a non-blocking call).

## Examples

- **Register an Application** – Registers the application with a one minute waiting period.

```
app.RegisterApplication(60);
```

### Usage

You must set up the `ConnectionProperties` and `ApplicationIdentifier` before you can invoke `registerApplication`.

The maximum length of the Application ID is 64 characters. The total length of the Application Connection ID cannot exceed 128 characters. For Windows platforms, the Application Connection ID format is: `deviceId$`

`{_instanceId}__applicationId`, if the `instanceId` is given; otherwise the format is `deviceId__applicationId`. For Windows Mobile, the Application Connection ID format is `deviceId__applicationId`. The `instanceId` separator is a single underscore (`_`), while the `applicationId` separator is two underscores.

```
Application app = Application.GetInstance();
// set Application ID - need to match as the server side Application
ID
app.ApplicationIdentifier = "SMP101";
app.ApplicationCallback = new MyApplicationCallbackHandler();
ConnectionProperties props = app.ConnectionProperties;
props.ServerName = "server.mycompany.com";
props.PortNumber = 5001;
LoginCredentials loginCred = new LoginCredentials("supAdmin",
"supPwd");
props.LoginCredentials = loginCred;

SMP101DB.SetApplication(app);

if (app.RegistrationStatus != RegistrationStatus.REGISTERED)
{
app.RegisterApplication(60);
}
```

### ApplicationCallback Property

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

For information on the `IApplicationCallback` class, see *Developer Guide: Windows and Windows Mobile Object API Applications > Client Object API Usage > Callback and Listener APIs > IApplicationCallback API*.

### Syntax

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

### Examples

- **Get the current ApplicationCallback handler**

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



**StartConnection**

Starts the connection for this application. This method is equivalent to calling `StartConnection(0)`, but is a non-blocking call which returns immediately. Use `getConnectionStatus` or the `ApplicationCallback` to retrieve the connection status.

**Syntax**

```
public void StartConnection()
```

**Returns**

None.

**Examples**

- **Start the Application**

```
StartConnection()
```

**Usage**

If you delete an application from SAP Control Center, when the client application calls `StartConnection()`, the following callback is triggered inside the `ApplicationCallback` handler:

```
void OnConnectionStatusChanged(int connectionStatus, int errorCode,
String errorMessage);
errorCode = 580
errorMessage = "Error: 580 Message: 'TM
Error:InvalidAuthenticationParameters'"
```

To continue using the application, call `UnregisterApplication()` to clean up the client state, and re-register using `RegisterApplication()`. You lose the previous subscription on the server side. Delete the client database and perform another initial synchronization.

**StartConnection (int timeout)**

Starts the connection for this application. If the connection was previously started, then this operation has no effect. You must set the appropriate `ConnectionProperties` before calling this operation. An `ApplicationTimeoutException` is thrown if the method does not succeed within the number of seconds specified by the timeout.

If connection properties are improperly set, a `ConnectionPropertyException` is thrown. You can set the `ApplicationCallback` before calling this operation to receive asynchronous notification of connection status changes. If a callback handler is registered and network connectivity is available, the sequence of callbacks as a result of calling `StartConnection` is:

```
OnConnectionStatusChanged (ConnectionStatus.CONNECTING, 0, "")  
OnConnectionStatusChanged (ConnectionStatus.CONNECTED, 0, "")
```

If a callback handler is registered and network connectivity is unavailable, the sequence of callbacks as a result of calling `StartConnection` is:

```
OnConnectionStatusChanged (ConnectionStatus.CONNECTING, 0, null)  
OnConnectionStatusChanged (ConnectionStatus.CONNECTION_ERROR, code,  
message)
```

After a connection is successfully established, it can transition at any later time to `CONNECTION_ERROR` status or `NOTIFICATION_WAIT` status and subsequently back to `CONNECTING` and `CONNECTED` when connectivity resumes.

---

**Note:** The application must have already been registered for the connection to be established. See *RegisterApplication* for details.

---

### **Syntax**

```
public void StartConnection(int timeout)
```

### **Parameters**

- **timeout** – The number of seconds to wait until the connection is started. If the timeout is greater than zero and the connection is not started within the timeout period, an `ApplicationTimeoutException` is thrown (the operation may still be completing in a background thread). If the timeout value is less than or equal to 0, then this method returns immediately without waiting for the registration to finish (a non-blocking call).

### **Returns**

None.

### **Examples**

- **Start the Application**

```
StartConnection(timeout)
```

### **ConnectionStatus Property**

This property represents the current status of the mobile application connection, one of the `ConnectionStatus` class values.

### **Syntax**

```
public int ConnectionStatus { get; set; }
```

### **Returns**

`ConnectionStatus` has the following possible values:

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

## Examples

- **Get the Status**

```
if (app.ConnectionStatus != ConnectionStatus.CONNECTED)
{
    // start the connection to server
    app.StartConnection(100);
}
```

## GetConnectionProperties

Retrieves the connection parameters from the application's connection properties instance. You must set connection properties before calling `StartConnection`, `RegisterApplication` or `UnregisterApplication`.

## Syntax

```
public ConnectionProperties ConnectionProperties
```

## Parameters

None.

## Returns

Returns the connection properties instance.

## ApplicationSettings Property

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

### **Syntax**

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

### **Returns**

Application settings that have been received from the SAP Mobile Server.

### **Examples**

- **Get the Application Settings**

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

### **BeginDownloadCustomizationBundle (System.IO.Stream writer)**

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

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

### **Syntax**

```
public void BeginDownloadCustomizationBundle (System.IO.Stream  
writer)
```

### **Parameters**

- **writer** – A writer stream that you provide.

### **Returns**

None.

### **Examples**

- **Download the default resource bundle**

```
string file = "Example1.zip";  
Stream writer = new FileStream(file, FileMode.OpenOrCreate,  
FileAccess.ReadWrite);  
Application.GetInstance().BeginDownloadCustomizationBundle(writer  
);
```

### **BeginDownloadCustomizationBundle (string customizationBundleID System.IO.Stream writer)**

Start downloading the specified resource bundle into the output stream.

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

## Syntax

```
public void BeginDownloadCustomizationBundle (String
customizationBundleID System.IO.Stream writer)
```

## Parameters

- **customizationBundleID** – The resource bundle name.
- **writer** – A writer stream that you provide.

## Returns

None.

## Examples

- **Download the specified resource bundle**

```
string file = "Example2.zip";
Stream writer = new FileStream(file, FileMode.OpenOrCreate,
FileAccess.ReadWrite);
Application.GetInstance().BeginDownloadCustomizationBundle("Examp
le:2.0", writer);
```

## StopConnection

Stops the connection for this application. This method is equivalent to calling `StopConnection(0)`.

## Syntax

```
public void StopConnection()
```

## Returns

None.

## Examples

- **Stop the Connection for the Application**

```
StopConnection();
```

## StopConnection (int timeout)

Stop the connection for this application. An `ApplicationTimeoutException` is thrown if the method does not succeed within the number of seconds specified by the timeout.

If no connection was previously stopped, then this operation has no effect. You can set the `ApplicationCallback` before calling this operation to receive asynchronous notification of connection status changes.

If a callback handler is registered, the sequence of callbacks as a result of calling `StopConnection` is:

- `OnConnectionStatusChanged(ConnectionStatus.DISCONNECTING, 0, "")`
- `OnConnectionStatusChanged(ConnectionStatus.DISCONNECTED, 0, "")`

### **Syntax**

```
public void StopConnection(int timeout)
```

### **Parameters**

- **timeout** – The number of seconds to wait until the connection is stopped. If the timeout value is less than or equal to 0, then this method returns immediately without waiting for the registration to finish (a non-blocking call).

### **Returns**

None.

### **Examples**

- **Stop the Application**

```
StopConnection(60)
```

### **UnregisterApplication**

Delete the registration for this application, and stop the connection. If no registration was previously created, or a previous registration was already deleted, then this operation has no effect. This method is equivalent to calling `UnregisterApplication(0)`, but is a non-blocking call which returns immediately. You can set the `applicationCallback` before calling this operation to receive asynchronous notification of registration status changes.

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

### **Syntax**

```
UnregisterApplication()
```

### **Parameters**

None.

### **Examples**

- **Unregister an Application** – Unregisters the application.

```
app.UnregisterApplication();
```

**UnregisterApplication(int timeout)**

Delete the registration for this application, and stop the connection. If no registration was previously created, or a previous registration was already deleted, then this operation has no effect. You must set the appropriate `ConnectionProperties` before calling this operation. You can set the `ApplicationCallback` before calling this operation to receive asynchronous notification of registration status changes.

If a callback handler is registered and network connectivity is available, the sequence of callbacks as a result of calling `UnregisterApplication` should be:

- `OnConnectionStatusChanged(ConnectionStatus.DISCONNECTING, 0, "")`
- `OnConnectionStatusChanged(ConnectionStatus.DISCONNECTED, 0, "")`
- `OnRegistrationStatusChanged(RegistrationStatus.UNREGISTERING, 0, "")`
- `OnRegistrationStatusChanged(RegistrationStatus.UNREGISTERED, 0, "")`

If a callback handler is registered and network connectivity is unavailable, the sequence of callbacks as a result of calling `UnregisterApplication` should be:

- `OnConnectionStatusChanged(ConnectionStatus.DISCONNECTING, 0, "")`
- `OnConnectionStatusChanged(ConnectionStatus.DISCONNECTED, 0, "")`
- `OnRegistrationStatusChanged(RegistrationStatus.UNREGISTERING, 0, "")`
- `OnRegistrationStatusChanged(RegistrationStatus.REGISTRATION_ERROR, code, message)`

**Syntax**

```
UnregisterApplication(int timeout)
```

**Parameters**

- **timeout** – Number of seconds to wait until the application is unregistered. If the timeout value is less than or equal to 0, then this method returns immediately without waiting for the registration to finish (a non-blocking call).

**Examples**

- **Unregister an Application** – Unregisters the application with a one minute waiting period.

```
app.UnregisterApplication(60);
```

**ConnectionProperties**

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

### **NetworkProtocol**

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

#### **Syntax**

```
public string NetworkProtocol
```

#### **Parameters**

None.

#### **Returns**

Returns the network protocol for the server connection URL.

### **LoginCertificate**

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

#### **Syntax**

```
public LoginCertificate LoginCertificate
```

#### **Parameters**

None.

#### **Returns**

Returns the login certificate.

### **LoginCredentials**

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

#### **Syntax**

```
public LoginCredentials LoginCredentials
```

#### **Parameters**

None.

#### **Returns**

Returns the login credentials.



**PortNumber**

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

**Syntax**

```
public int PortNumber
```

**Parameters**

None.

**Returns**

Returns the port number.

**ServerName**

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

**Syntax**

```
public string ServerName
```

**Parameters**

None.

**Returns**

Returns the server name.

**SecurityConfiguration**

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

**Syntax**

```
public string SecurityConfiguration
```

**Parameters**

None.

### **Returns**

Returns the security configuration.

### **UrlSuffix**

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

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

---

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

---

### **Syntax**

```
public string UrlSuffix
```

### **Parameters**

None.

### **Returns**

Returns the URL suffix.

### **Usage**

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

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

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

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

result in the following URL suffix:

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

### **FarmId**

Retrieve or set the Farm ID for the server connection URL. This optional property is used in the URL discovery process when connecting through a proxy server or Relay Server. The `FarmId` is substituted into the default URL templates for Relay Server on into a configured

`UrlSuffix`. The `FarmId` is used only until a connection is successfully made and the permanent `UrlSuffix` is stored.

### **Syntax**

```
public string FarmId
```

### **Parameters**

None.

### **Returns**

Returns the Farm ID.

### **HttpHeaders**

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

### **Syntax**

```
public StringProperties HttpHeaders
```

### **Parameters**

None.

### **Returns**

Returns the HTTP headers.

### **HttpCookies**

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

### **Syntax**

```
public StringProperties HttpCookies
```

### **Parameters**

None.

### **Returns**

Returns the HTTP cookies.

### **HttpCredentials**

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

### **Syntax**

```
public LoginCredentials HTTPCredentials
```

### **Parameters**

None.

### **Returns**

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

## **ApplicationSettings**

Methods or properties in the ApplicationSettings class.

### **IsApplicationSettingsAvailable**

Checks whether the application settings are available from the SAP Mobile Server.

### **Syntax**

```
public bool IsApplicationSettingsAvailable()
```

### **Parameters**

None.

### **Returns**

Returns true if the application settings are available.

### **Examples**

- **Check if application settings are available**

```
bool isSettingsAvailable =  
    Sybase.Mobile.Application.GetInstance().ApplicationSettings.IsApp  
    licationSettingsAvailable();
```

### **GetStringProperty**

Retrieves a string property from the applicationSettings.

### **Syntax**

```
public String GetStringProperty(ConnectionPropertyType type)
```

**Parameters**

- **type** – Type of ConnectionPropertyType.

**Returns**

Returns a string property value.

**Examples**

- **Get string property**

```
string user_name =
appSettings.GetStringProperty(ConnectionPropertyType.UserName);
```

**GetIntegerProperty**

Retrieves an integer property from the applicationSettings.

**Syntax**

```
public int? GetIntegerProperty(ConnectionPropertyType type)
```

**Parameters**

- **type** – Type of ConnectionPropertyType.

**Returns**

Returns an integer property value.

**Examples**

- **Get integer property**

```
int? min_length =
appSettings.GetIntegerProperty(ConnectionPropertyType.PwdPolicy_L
ength);
```

**GetBooleanProperty**

Retrieves a boolean property from the applicationSettings.

**Syntax**

```
public bool? GetBooleanProperty(ConnectionPropertyType type)
```

**Parameters**

- **type** – Type of ConnectionPropertyType.

### **Returns**

Returns a boolean property value.

### **Examples**

- **Get boolean property**

```
bool? pwdpolicy_enabled =  
appSettings.GetBooleanProperty(ConnectionPropertyType.PwdPolicy_E  
nabled);
```

### **Custom1**

A custom application setting for use by the application code.

### **Syntax**

```
public String Custom1
```

### **Parameters**

None.

### **Returns**

Returns a custom application setting.

### **Custom2**

A custom application setting for use by the application code.

### **Syntax**

```
public String Custom2
```

### **Parameters**

None.

### **Returns**

Returns a custom application setting.

### **Custom3**

A custom application setting for use by the application code.

### **Syntax**

```
public String Custom3
```

### **Parameters**

None.

**Returns**

Returns a custom application setting.

**Custom4**

A custom application setting for use by the application code.

**Syntax**

```
public String Custom4
```

**Parameters**

None.

**Returns**

Returns a custom application setting.

**DomainName****Syntax**

```
public String DomainName
```

**Parameters**

None.

**Returns**

Returns the domain name.

**ConnectionId****Syntax**

```
public String ConnectionId
```

**Parameters**

None.

**Returns**

Returns a Connection ID for this application setting.

## **ConnectionPropertyType**

Methods or properties in the `ConnectionPropertyType` class.

See the generated API reference provided with the Mobile SDK for a complete list of methods in the `ConnectionPropertyType` class.

### **PwdPolicy\_Enabled**

Indicates whether the password policy is enabled.

#### **Syntax**

```
ConnectionPropertyType PwdPolicy_Enabled
```

#### **Parameters**

None.

#### **Returns**

#### **Examples**

- **PwdPolicy\_Enabled**

```
bool? pwdpolicy_enabled =  
appSettings.GetBooleanProperty(ConnectionPropertyType.PwdPolicy_Enabled);
```

### **PwdPolicy\_Default\_Password\_Allowed**

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

#### **Syntax**

```
ConnectionPropertyType PwdPolicy_Default_Password_Allowed
```

#### **Parameters**

None.

#### **Returns**

None.



## Examples

- **PwdPolicy\_Default\_Password\_Allowed**

```
bool? default_password_allowed =  
appSettings.GetBooleanProperty(ConnectionPropertyType.PwdPolicy_D  
efault_Password_Allowed);
```

## PwdPolicy\_Length

Defines the minimum length for a password.

## Syntax

```
ConnectionPropertyType PwdPolicy_Length
```

## Parameters

None.

## Returns

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

## Examples

- **PwdPolicy\_Length**

```
int? min_length =  
appSettings.GetIntegerProperty(ConnectionPropertyType.PwdPolicy_L  
ength);
```

## PwdPolicy\_Has\_Digits

Indicates if the password must contain digits.

## Syntax

```
ConnectionPropertyType PwdPolicy_Has_Digits
```

## Parameters

None.

## Returns

Returns true if the password must contain digits.

### Examples

- **PwdPolicy\_Has\_Digits**

```
bool? has_digits =  
appSettings.GetBooleanProperty(ConnectionPropertyType.PwdPolicy_H  
as_Digits);
```

### PwdPolicy\_Has\_Upper

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

### Syntax

```
ConnectionPropertyType PwdPolicy_Has_Upper
```

### Parameters

None.

### Returns

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

### Examples

- **PwdPolicy\_Has\_Upper**

```
bool? has_upper =  
appSettings.GetBooleanProperty(ConnectionPropertyType.PwdPolicy_H  
as_Upper);
```

### PwdPolicy\_Has\_Lower

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

### Syntax

```
ConnectionPropertyType PwdPolicy_Has_Lower
```

### Parameters

None.

### Returns

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

## Examples

- **PwdPolicy\_Has\_Lower**

```
bool? has_lower =
appSettings.GetBooleanProperty(ConnectionPropertyType.PwdPolicy_Has_Lower);
```

## PwdPolicy\_Has\_Special

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

## Syntax

```
ConnectionPropertyType PwdPolicy_Has_Special
```

## Parameters

None.

## Returns

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

## Examples

- **PwdPolicy\_Has\_Special**

```
bool? has_special =
appSettings.getBooleanProperty(ConnectionPropertyType.PwdPolicy_Has_Special);
```

## PwdPolicy\_Expires\_In\_N\_Days

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

## Syntax

```
ConnectionPropertyType PwdPolicy_Expires_In_N_Days
```

## Parameters

None.

## Returns

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

### Examples

- **PwdPolicy\_Expires\_In\_N\_Days**

```
int? expires_in_n_days =  
appSettings.getIntegerProperty(ConnectionPropertyType.PwdPolicy_Expires_In_N_Days);
```

### PwdPolicy\_Min\_Unique\_Chars

Specifies the minimum number of unique characters in the password.

### Syntax

```
ConnectionPropertyType PwdPolicy_Min_Unique_Chars
```

### Parameters

None.

### Returns

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

### Examples

- **PwdPolicy\_Min\_Unique\_Chars**

```
int? min_unique_characters =  
appSettings.getIntegerProperty(ConnectionPropertyType.PwdPolicy_Min_Unique_Chars);
```

### PwdPolicy\_Lock\_Timeout

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

### Syntax

```
ConnectionPropertyType PwdPolicy_Lock_Timeout
```

### Parameters

None.

### Returns

An integer specifying the timeout value.

## Examples

- **PwdPolicy\_Lock\_Timeout**

```
int? lock_timeout =
appSettings.getIntegerProperty(ConnectionPropertyType.PwdPolicy_Lock_Timeout);
```

## PwdPolicy\_Retry\_Limit

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

## Syntax

```
ConnectionPropertyType PwdPolicy_Retry_Limit
```

## Parameters

None.

## Returns

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

## Examples

- **PwdPolicy\_Retry\_Limit**

```
int? retry_limit =
appSettings.getIntegerProperty(ConnectionPropertyType.PwdPolicy_Retry_Limit);
```

# Connection APIs

---

The Connection APIs contain methods for managing local database information, establishing a connection with the SAP Mobile Server, and authenticating.

## ConnectionProfile

The `ConnectionProfile` class manages local database information. Set its properties, including the encryption key, during application initialization, and before creating or accessing the local client database.

By default, the database class name is generated as "packageName"+"DB".

```
ConnectionProfile cp = SMP101DB.GetConnectionProfile();
cp.PageSize = 4*1024;
cp.EncryptionKey = "Your encryption key"; //Encryption key could be
of arbitrary length. Generally the longer, the better.
cp.Save();
```

---

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

---

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

You can use the `cacheSize` connection property to control the size of the memory cache used by the database.

```
public int CacheSize {get; set; }
```

### **Managing Device Database Connections**

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

---

**Note:** Any database operation triggers the establishment of the database connection. You do not need to explicitly call the `OpenConnection` API.

---

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

All `ConnectionProfile` properties should be set before the first access to database, otherwise they will not take effect.

The `CloseConnection()` method closes all database connections for this package and releases all resources allocated for those connections. This is recommended to be part of the application shutdown process.

### **Improving Device Application Performance with One Writer Thread and Multiple Database Access Threads**

The `MaxDbConnections` property improves device application performance by allowing multiple threads to access data concurrently from the same local database.

Connection management allows you to have at most one writer thread concurrent with multiple reader threads. There can be other reader threads at the same time that the writer thread is writing to the database. The total number of threads are controlled by the `MaxDbConnections` property.

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

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

You can specify the number of total threads using `MaxDbConnections`. The `ConnectionProfile` class in the persistence package includes the `MaxDbConnections` property, which you set before performing any operation in the application. The default value (maximum number of concurrent read threads) is 2.

```
ConnectionProfile connectionProfile =
SMP101DB.GetConnectionProfile();
```

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

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

## 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 = SMP101DB.GetConnectionProfile();
cp.SetProperty("databaseFile", "smp101.udb");
cp.Save();
```

### *Examples*

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/smpprj.db`.

---

### *Usage*

- Be sure to call this API before the database is created..
- The database is UltraLite; use a database file name like `mydb.udb`.
- For DOE applications, the database is SQLite; use a database file name like `mydb.db`.
- If the device client user changes the file name, he or she must make sure the input file name is a valid name and path on the client side.

---

**Note:** SAP recommends using industrial grade SD cards using Single Level Cell (SLC) technology. SD cards that use SLC technology are generally more reliable and faster than MLC cards, although they may be more limited in size and more expensive per unit of storage. Not all SD cards perform equally, and it is advised that customers evaluate the benchmarks available from different suppliers.

---

## Synchronization Profile

---

The Synchronization Profile contains information for establishing a connection with the SAP Mobile Server's data synchronization channel where the server package has been deployed. The `Sybase.Persistence.ConnectionProfile` class manages that information. By default, this information includes the server host, port, domain name, certificate and public key that are pushed by the message channel during the registration process.

Settings are automatically provisioned from the SAP Mobile Server. The values of the settings are inherited from the application connection template used for the registration of the application connection (automatic or manual). You must make use of the connection and security settings that are automatically used by the Object API.

Typically, the application uses the settings as sent from the SAP Mobile Server to connect to the SAP Mobile Server for synchronization so that the administrator can set those at the application deployment time based on their deployment topology (for example, using Relay Server, using e2ee security, or a certificate used for the intermediary, such as a Relay Server Web server). See the *Applications* and *Application Connection Templates* topics in *System Administration*.

```
ConnectionProfile profile = SMP101DB.GetSynchronizationProfile();
profile.DomainName = "default";
profile.ServerName = "host-name";
profile.PortNumber = 2481;
profile.NetworkProtocol = "https";
profile.StreamParams.Trusted_Certificates = "rsa_public_cert.crt";
```

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

```
NetworkStreamParams streamParams =
MyDatabase.GetSynchronizationProfile().StreamParams;
streamParams.Compression= "zlib";
streamParams.Zlib_Upload_Window_Size= 12;
streamParams.Zlib_Download_Window_Size= 12;
```

By default, compression is enabled.

You can allow clients to authenticate with a proxy server by including a username and password in the sync parameters. This feature supports Basic and Digest authentication as described in RFC 2617. With Basic authentication, passwords are included in HTTP headers in clear text; however, you can use HTTPS to encrypt the headers and protect the password. With Digest authentication, headers are not sent in clear text but are hashed.



```
SMP101DB.GetSynchronizationProfile().NetworkStreamParams +=
";http_userid=supAdmin;http_password=supPwd";
```

## **Connect the Data Synchronization Channel Through a Relay Server**

To enable your client application to connect through a Relay Server, you can enter the related configuration in the application connection template through SAP Control Center, and/or setup the configuration properties in the synchronization profile using the object API.

Edit SMP101DB by modifying the values of the Relay Server properties for your Relay Server environment.

To update properties for a Relay Server installed on Apache:

```
ConnectionProfile syncProfile =
SMP101DB.GetSynchronizationProfile();
syncProfile.ServerName = "examplexp-vm1";
syncProfile.PortNumber = 80;
syncProfile.NetworkProtocol = "http";
NetworkStreamParams streamParams =
GetSynchronizationProfile().StreamParams;
streamParams.Url_Suffix = "/cli/iarelayserver/<FarmName>";
GetSynchronizationProfile().DomainName = "default";
```

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

```
ConnectionProfile syncProfile =
SMP101DB.GetSynchronizationProfile();
syncProfile.ServerName = "examplexp-vm1";
syncProfile.PortNumber = 80;
syncProfile.NetworkProtocol = "http";
NetworkStreamParams streamParams =
GetSynchronizationProfile().StreamParams;
streamParams.Url_Suffix = "/ias_relay_server/client/rs_client.dll/
<FarmName>";
GetSynchronizationProfile().DomainName = "default";
```

For more information on relay server configuration, see *System Administration* and *SAP Control Center for SAP Mobile Platform*.

## **Asynchronous Operation Replay**

When an application calls `SubmitPending` on an MBO on which a create, update, or delete operation is performed, an operation replay record is created on the device local database.

When `synchronize` is called, the operation replay records are uploaded to the server. The method returns without waiting for the backend to replay those records. The `synchronize` method downloads all the latest data changes and the results of the previously uploaded operation replay records that the backend has finished replaying. If you choose to disable asynchronous operation replay, each `synchronize` call will wait for the backend to finish replaying all the current uploaded operation replay records.

By default, synchronization will not wait for the operations to be replayed on the backend. When the replay is finished, the `onSynchronize` callback method will be called with this status code in the `SynchronizeContext`:

```
SynchronizationStatus.ASYNC_REPLAY_COMPLETED
```

The application can set the following property in the synchronization profile to use the previous Synchronous Operation Replay behavior.

```
SMP101DB.GetSynchronizationProfile().SetAsyncReplay(false);
```

---

**Note:** Synchronous operation replay against MBOs using an EIS managed cache group policy are automatically treated as asynchronous replay by the SAP Mobile Platform Runtime.

---

## Authentication APIs

---

You can log in to the SAP Mobile Server with your user name and credentials and use the X.509 certificate you installed in the task flow for single sign-on.

### Logging In

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

The package database class provides methods for logging in to the SAP Mobile Server:

- **OnlineLogin()** – authenticates credentials against the SAP Mobile Server.

### Importing the Private Key into the System Certificate Store

Set the private key as exportable and import the `.pfx` file into the system certificate store for SAP Mobile Platform to support mutual SSL authentication.

Import a `.pfx` file into the system certificate store. `bMakePrivateKeyExportable` must be set to true to enable exporting the private key.

```
Sybase.Persistence.CertificateStore.ImportPfxIntoSystemStore(String sFile, String sPassword, bool bMakePrivateKeyExportable)
```

### Sample Code: Setting Up Login Credentials

Illustrates importing the certificate and setting up login credentials, as well as other APIs related to certificate handling:

```
// SMP101DB is a generated database class
// First install certificates on your emulator, for example
"SAP101.p12"
```

```
//Getting certificate from certificate store
CertificateStore myStore =
CertificateStore.GetDefault();
string filter1 = "SAP";
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
SMP101DB.GetSynchronizationProfile().Certificate = lc;

//Perform synchronization
SMP101DB.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);

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

## **Sample Code: Mutual Authentication**

Illustrates client configuration to support mutual authentication, as well as other APIs related to certificate handling:

```
//Step 1: Get the login certificate from a certificate store

CertificateStore myStore = CertificateStore.GetDefault();
StringList labels = myStore.CertificateLabels();
string aLabel = labels.Item(0);
String filter1 = "Test";
labels = myStore.CertificateLabels(filter1, null);
LoginCertificate lc = myStore.GetSignedCertificate(aLabel,
"changeit");

//Step 2: Register the application

Application.GetInstance().ApplicationIdentifier =
"customer.service";
DsTestDB.SetApplication(Application.GetInstance());
```

```
ConnectionProperties pro =
Application.GetInstance().ConnectionProperties;
pro.LoginCertificate = lc;
pro.ServerName = "bfang-desktop";
pro.PortNumber = 8002;
pro.NetworkProtocol = "HTTPS";
pro.FarmId = "0";
pro.UrlSuffix = "";
pro.SecurityConfiguration = "cert";
if (Application.GetInstance().RegistrationStatus ==
RegistrationStatus.UNREGISTERED)
{
    Application.GetInstance().RegisterApplication(100);
}
.....

//Step 3: Get ready to synchronize

DsTestDB.GetSynchronizationProfile().NetworkProtocol = "HTTPS";
DsTestDB.GetSynchronizationProfile().PortNumber = 2482;
DsTestDB.GetSynchronizationProfile().ServerName = "bfang-desktop";
DsTestDB.GetSynchronizationProfile().Certificate = lc;
DsTestDB.getSynchronizationProfile().NetworkStreamParams =
"trusted_certificate=serverCA.crt";

//Step 4: Synchronize

.....
DsTestDB.Synchronize();
.....
```

### **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 SAP Mobile Server synchronization.

Refer to the API Reference for implementation details.

#### **Importing a Certificate into the Data Vault**

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

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 blob, which is a digitally signed copy of the public X.509 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 SAP Mobile Server Connections**

Select the X.509 certificate from the data vault for SAP Mobile 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 SAP Mobile Server with a Certificate**

Once the certificate property is set, use the LoginToSync or OnlineLogin API with no parameters.

```
//connect to SAP Mobile Server with the login certificate
SMP101DB.GetSynchronizationProfile().Certificate = lc;
SMP101DB.OnlineLogin();
```

## Personalization APIs

---

Personalization keys allow the application to define certain input parameter values that are personalized for each mobile user. Personalization parameters provide default values for synchronization parameters when the synchronization key of the object is mapped to the personalization key while developing a mobile business object. The Personalization APIs allow you to manage personalization keys, and get and set personalization key values.

### Type of Personalization Keys

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

A personalization parameter can be a primitive or complex type.

A personalization key is metadata that enables users to store their search preferences on the client, the server, or by session. The preferences narrow the focus of data retrieved by the mobile device (also known as the filtering of data between the client and the SAP Mobile Server). Often personalization keys are used to hold backend system credentials, so that they can be propagated to the EIS. To use a personalization key for filtering, it must be mapped to a synchronization parameter. The developer can also define personalization keys for the application, and can use built-in personalization keys available in the SAP Mobile Server. Two built-in (session) personalization keys — username and password — can be used to perform single sign-on from the device application to the SAP Mobile Server, authentication and authorization on the SAP Mobile Server, as well as connecting to the back-end EIS using the same set of credentials. The password is never saved on the server.

### Getting and Setting Personalization Key Values

The `PersonalizationParameters` class is generated automatically for managing personalization keys. When a personalization parameter value is changed, the call to `save` automatically propagates the change to the server.

An operation can have a parameter that is one of the SAP Mobile Platform list types (such as `IntList`, `StringList`, or `ObjectList`). This code shows how to set a personalization key, and pass an array of values and an array of objects:

```
PersonalizationParameters pp =
SMP101DB.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.

## Synchronization APIs

---

You can synchronize mobile business objects (MBOs) based on synchronization parameters, for individual MBOs, or as a group, based on the group's synchronization policy.

---

**Note:** The `LoginToSync` API is now deprecated. Call `Synchronize` or `BeginSynchronize` before saving synchronization parameters. After saving the synchronization parameters, call `Synchronize` or `BeginSynchronize` again to retrieve the new values filtered by those parameters.

---

## Managing Synchronization Parameters

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

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

To add a synchronization parameter:

```

CustomerSubscription sp = new CustomerSubscription();
sp.Name = "example";
Customer.AddSubscription(sp);

```

To list all synchronization parameters:

```

Sybase.Collections.GenericList<CustomerSubscription> r =
Customer.GetSubscriptions();

```

To remove a synchronization parameter:

```

Sybase.Collections.GenericList<CustomerSubscription> r =
Customer.GetSubscriptions();
CustomerSubscription sub = r.Item(0);
Customer.RemoveSubscription(sub);

```

## **Performing Mobile Business Object Synchronization**

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

This code synchronizes an MBO package using a specified connection:

```
SMP101DB.Synchronize (string synchronizationGroup)
```

The package database class includes two synchronization methods. You can synchronize a specified group of MBOs using the synchronization group name:

```
SMP101DB.Synchronize ("my-sync-group") ;
```

Or, you can synchronize all synchronization groups:

```
SMP101DB.Synchronize () ;
```

There is a default synchronization group within every package. The default synchronization group includes all MBOs except those already included by other synchronization groups. To synchronize a default synchronization group call:

```
SMP101DB.BeginSynchronize ("default") ; or  
SMP101DB.Synchronize ("default") ;
```

If there is no other synchronization group, call `SMP101DB.BeginSynchronize () ;` or `SMP101DB.Synchronize () ;`

To synchronize a synchronization group asynchronously:

```
GenericList<SynchronizationGroup> syncGroups = new  
GenericList<SynchronizationGroup> () ;  
syncGroups.add (SMP101DB.GetSynchronizationGroup ("my-sync-group")) ;  
SMP101DB.BeginSynchronize (syncGroups, "");
```

When an application uses a create, update, or delete operation in an MBO and calls the `SubmitPending` method, an `OperationReplay` object is created for that change. The application must invoke either the `Synchronize` or `BeginSynchronize` method to upload the `OperationReplay` object to the server to replay the change on the backend data source. The `IsReplayQueueEmpty` API is used to check if there are unsent operation replay objects and decide whether a synchronize call is needed.

## **Message-Based Synchronization APIs**

The message-based synchronization APIs enable a user application to subscribe to a server package, to remove an existing subscription from the SAP Mobile Server, to suspend or



resume requests to the SAP Mobile Server, and to recover data related to the package from the server.

---

**Note:** The `BeginOnlineLogin`, `SuspendSubscription`, `ResumeSubscription`, and `VacuumDatabase` methods are for use with DOE-based applications only.

---

### **BeginOnlineLogin**

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

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

**BeginOnlineLogin** sends a message to the SAP Mobile Server with the username and password. The SAP Mobile Server responds with a message to the client with the login success or failure.

When the login succeeds, the `OnLoginSuccess` method of the `ICallbackHandler` is invoked. When the login fails, the `OnLoginFailure` method of the `ICallbackHandler` is invoked.

### **Syntax**

```
public static void BeginOnlineLogin(string username, string password)
```

### **Parameters**

- **userName** – the user name.
- **password** – the password.

### **Returns**

None.

### **Examples**

- **Begin an Online Login** – Start logging in with "supAdminID" for the user name and "supPass" for the password.

```
SMP101DB.BeginOnlineLogin("supAdminID", "supPass");
```

### **Subscribe**

Subscribes to a server package. A subscription message is sent to the SAP Mobile Server and the application receives a subscription request result notification from the the SAP Mobile Server. If the subscription succeeds, the `OnSubscribeSuccess` method of the

ICallbackHandler is invoked. If the subscription fails, the OnSubscribeFailure method of the ICallbackHandler is invoked.

Prerequisites for using **Subscribe**:

- The mobile application is compiled with the client framework and deployed to a mobile device, together with the SAP Mobile Platform client process.
- The device application has already configured SAP Mobile Server connection information.

### **Syntax**

```
public static void Subscribe()
```

### **Parameters**

- **None** – **subscribe** has no parameters.

### **Returns**

None.

### **Examples**

- **Subscribe to a Sample Application** – Subscribe to SMP101DB.

```
SUP101DB.Subscribe();
```

### **Unsubscribe**

Removes an existing subscription to a server package. An unsubscription message is sent to the SAP Mobile Server and the application receives a subscription request result notification from the SAP Mobile Server as a notification. The data on the local database is cleaned. If the unsubscribe succeeds, the OnSubscribeSuccess method of the ICallbackHandler is invoked. If it fails, the OnSubscribeFailure method of the ICallbackHandler is invoked.

The device application must already have a subscription with the server.

### **Syntax**

```
public static void Unsubscribe()
```

### **Parameters**

- **None** – **unsubscribe** has no parameters.

### **Returns**

None.

## Examples

- **Unsubscribe from a Sample Application** – Unsubscribe from SMP101DB.

```
SMP101DB.Unsubscribe();
```

## SuspendSubscription

Sends a suspend request to the SAP Mobile Server to notify the server to stop delivering data changes. A suspend subscription message is sent to the SAP Mobile Server and the application receives a suspend subscription request result notification from the SAP Mobile Server as a notification. If the suspend succeeds, the `OnSuspendSubscriptionSuccess` method of the `ICallbackHandler` is invoked. If the suspend fails, the `OnSuspendSubscriptionFailure` method of the `ICallbackHandler` is invoked.

## Syntax

```
public static void SuspendSubscription()
```

## Parameters

- **None** – `SuspendSubscription` has no parameters.

## Returns

None.

## Examples

- **Suspend a Subscription** – Suspend the subscription to SMP101DB.

```
SMP101DB.SuspendSubscription();
```

## BeginSynchronize

Sends a message to the SAP Mobile Server to synchronize data between the client and the server. There are two different `BeginSynchronize` APIs, one with no parameters that synchronizes all the groups, and one that takes a list of groups.

The synchronization completes in the background through an asynchronous message exchange with the server. If application code needs to know when the synchronization is complete, a callback handler that implements the `OnSynchronize` method must be registered with the database class.

## Syntax

```
public static void BeginSynchronize
public static void
BeginSynchronize(Sybase.Collections.GenericList<Sybase.Persistence.
ISynchronizationGroup> synchronizationGroups, string context)
```

### Parameters

- **synchronizationGroups** – specifies a list of a list of `SynchronizationGroupISynchronizationGroup` objects representing the groups to be synchronized. If omitted, begin synchronizing data for all groups.

**Note:** This parameter is not relevant for DOE packages; pass a null value to this parameter.

### Returns

None.

### Examples

- **Synchronize Data between Client and Server** – Synchronize data for SMP101DB for SMP101DB group.

```
SMP101DB.BeginSynchronize(null, "your context");
```

### ResumeSubscription

Sends a resume request to the SAP Mobile Server.

The resume request notifies the SAP Mobile Server to resume sending data changes for the subscription that had been suspended. On success, **OnResumeSubscriptionSuccess** callback handler method is called. On failure, **OnResumeSubscriptionFailure** callback handler is called.

### Syntax

```
public static void ResumeSubscription()
```

### Parameters

- **None** – **ResumeSubscription** has no parameters.

### Returns

None.

### Examples

- **Resume a Subscription** – Resume the subscription to SMP101DB.

```
SMP101DB.ResumeSubscription();
```

### **VacuumDatabase**

Reclaims unused database space left when records are deleted, reducing the size of the database file.

When a large amount of data is deleted from the database file it leaves behind empty space, or "free" database pages. Running `VacuumDatabase` to rebuild the database reclaims this space. The size of the database file is reduced by the amount of space reclaimed.

### **Syntax**

```
public static void VacuumDatabase()
```

### **Parameters**

- **None** – `VacuumDatabase` has no parameters.

### **Returns**

None.

### **Examples**

- **Reclaim Empty Space in Database** – Reclaim empty space in SMP101DB database.

```
SMP101DB.VacuumDatabase();
```

## **Push Synchronization Applications**

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

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

```
ISynchronizationGroup sg =
SMP101DB.GetSynchronizationGroup("TCNEnabled");

if (!sg.EnableSIS)
{
    sg.EnableSIS = true;
    sg.Interval = 2;
    sg.Save();
    SMP101DB.Synchronize("TCNEnabled");
}
```

When the server detects changes in an MBO affecting a client device, and the synchronization group of the MBO has change detection enabled, the server will send a notification to client

device through messaging channel. By default, a background synchronization downloads the changes for that synchronization group. The application can implement the `onSynchronize` callback method to monitor this condition, and either allow or disallow background synchronization.

```
public int OnSynchronize(GenericList<ISynchronizationGroup> groups,
    SynchronizationContext context)
{
    int status = context.Status;
    if (status == SynchronizationStatus.STARTING_ON_NOTIFICATION)
    {
        // There is changes on the synchronization group
        if (busy)
        {
            return SynchronizationAction.CANCEL;
        }
        else
        {
            return SynchronizationAction.CONTINUE;
        }
    }

    // return CONTINUE for all other status
    return SynchronizationAction.CONTINUE;
}
```

### **Retrieving Information about Synchronization Groups**

The package database class provides methods for querying the synchronized state and the last synchronization time of a certain synchronization group.

```
/// Determines if the synchronization group was synchronized
public static bool IsSynchronized(string synchronizationGroup)

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

### **Resumable Synchronization**

The resumable synchronization APIs enable an interrupted synchronization with partial data downloaded to the device to resume from the point of interruption, without restarting the entire download again.

Resumable synchronization is particularly useful when you synchronize large data sets, and do not want to have to cancel or roll back any partially downloaded data. You can resume the synchronization that meets all of these conditions:

- It must be within the lifetime of the application. You cannot resume the session after the application terminates, for example if the application is closed.
- The resumable option must be set before calling synchronization.
- The synchronization must be performed asynchronously in background mode.

- There is partial data downloaded to the device when the interruption occurs.

Synchronization can be interrupted by either environmental changes (for example, a dropped connection) or when the application code disables or cancels the synchronization (for example, by returning true in the SyncStatusListener ObjectSyncStatus method).

- **Application code disables or cancels the synchronization for any reason** – for example,

---

**Note:** You cannot resume synchronization if the MobiLink Server restarts, since all pending downloads are lost.

---

If your callback handler is registered, and if some data has downloaded data before the interruption, this status is returned in the ICallbackHandler OnSynchronize method at the time of the interruption:

```
SynchronizationStatus.INTERRUPTED_WITH_PARTIAL_DATA
```

The system locks the synchronization queue in a pending synchronization mode that waits for the response to either resume or cancel.

The application determines when to resume the synchronization. The resume Synchronization API first releases the locked synchronization queue, then sends a resume synchronization request to the server. If the request is too early, the resumption may fail immediately if the connection is not yet available. However, if the request is sent too late, the download data may no longer be available on the server.

See the topic *Configuring Resumable Synchronization* in the *SAP Control Center for SAP Mobile Server* collection for information about setting a cache storage size for resumable download data. The default cache size is 5 GB.

## **Resumable Synchronization API**

Resume an interrupted synchronization where partial data has been downloaded to the device.

### **Syntax**

To resume a pending synchronization, call:

```
SMP101DB.ResumePendingSynchronization();
```

The application can also determine when to cancel synchronization. For example, you could add logic to the application that sets a max retry resume to cancel the pending synchronization when the application reaches the max setting. The cancel pending synchronization API also unlocks the locked synchronization queue first, but instead of sending a synchronization request, it rolls back any partial data that has already downloaded to the device. To not resume the pending synchronization, call:

```
SMP101DB.CancelPendingSynchronization();
```

1. When the system is in a pending synchronization state, waiting for either a resume pending synchronization request by calling ResumePendingSynchronization API or a cancellation

request by calling `CancelPendingSynchronization`, the synchronization queue is locked. Any other new blocking or non-blocking synchronization requests are not permitted.

2. If partial data has not downloaded to the device when the interruption occurs, for example, the device is uploading data to the server, or the server is still fetching data from the EIS, the synchronization request terminates immediately and the client reverts to standard behavior. If the interruption is due to:
  - **Environmental changes** – instead of receiving a `SynchronizationStatus.INTERRUPTED_WITH_PARTIAL_DATA` status, the exception in the sync context is returned.
  - **Application code disables or cancels the synchronization for any reason** – `SyncStatusListener`'s `objectSyncStatus` method returns true.

## Examples

- **Sample code for setting the resumable option then synchronize**

```

...
...
...
//turn on the resume synchronization option
SMP101DB.GetSynchronizationProfile().ResumeSynchronization =
true;
MyCallbackHandlerForResumeSync callback = new
MyCallbackHandlerForResumeSync();
SMP101DB.RegisterCallbackHandler(callback);
MyResumeSyncStatusListener listener = new
MyResumeSyncStatusListener();
GenericList<ISynchronizationGroup> groups = new
GenericList<ISynchronizationGroup>();
groups.Add(SynchronizationGroup.GetInstance("default"));

//submit a non-blocking sync request
SMP101DB.BeginSynchronize(groups, null, false, listener);

```

- **CallbackHandler sample code in receiving `SynchronizationStatus.INTERRUPTED_WITH_PARTIAL_DATA` status**

```

//in the implementation of SUPDefaultCallbackHandler class
...
...
...
SynchronizationAction
OnSynchronize(GenericList<ISynchronizationGroup> groups,
SynchronizationContext context)
{
    switch (context.Status)
    {
        ...
        case SynchronizationStatus.INTERRUPTED_WITH_PARTIAL_DATA:
            _isInterruptedWithPartialData = true;

            // By returning continue, the system waits for a request
            to resume or cancel the pending synchronization and the

```



```

synchronization queue is locked.
    // If returning SynchronizationAction.CANCEL instead,
the system automatically cancels the pending synchronization.
    // Application cannot resume the synchronization and the
synchronization queue is not locked.

        return SynchronizationAction.CONTINUE;

        ...
    }
}

```

- **Sample code for resuming or canceling synchronization on sync callback status changed notification**

```

void HandleInterruptedWithPartialData()
{
    if (MAX_INTERRUPTED_STATUS == _interruptedStatusCount)
    {
        CancelPendingSync();
        Console.WriteLine("Resume sync is canceled.");
        Console.WriteLine("Interrupted status count reached the
max: {0}.", MAX_INTERRUPTED_STATUS);
    }
    else
    {
        _isInterruptedWithPartialData = true;
        _interruptedStatusCount++;

        int waitCount = 0;
        while(!_isDeviceConnectedToInternet)
        {
            System.Threading.Thread.Sleep(1000);
            waitCount++;

            if (waitCount > MAX_WAIT)
            {
                Console.WriteLine("There is no network connection
{0} seconds.", MAX_WAIT);
                break;
            }
        }

        if (_isDeviceConnectedToInternet)
        {
            ResumeSync();
        }
        else
        {
            CancelPendingSync();

            Console.WriteLine("Resume sync is canceled.");
            Console.WriteLine("there's no network connectivity
for too long.");
        }
    }
}

```

```
void CancelPendingSync()
{
    Console.WriteLine("going to cancel pending sync...");

    if (SMP101DB.CancelPendingSynchronization())
    {
        Console.WriteLine("failed to cancel pending sync.");
    }
    else
    {
        Console.WriteLine("pending sync has been cancelled.");
    }
}

void ResumeSync()
{
    if (!_isInterruptedWithPartialData)
    {
        Console.WriteLine("no resume as there is no interrupted
with partial data.");
        return;
    }

    Console.WriteLine("going to resume sync...");
    bool resumed = SMP101DB.ResumePendingSynchronization();
    _isInterruptedWithPartialData = false;
    if (!resumed)
    {
        Console.WriteLine("failed to resume pending syn.");
    }
    else
    {
        _resumeSyncCount++;
        Console.WriteLine("resume sync count: {0}.",
_resumeSyncCount);
    }
}
```

### **Resumable Synchronization Recommendations**

Consider these recommendations when implementing resumable synchronization.

- Set a max resume-attempt in your code so the application does not continuously try to resume and fail for the same reason. If the max count is reached, the sync is canceled.
- Check for connectivity before attempting to resume the synchronization. The code can be implemented to wait a short time to recheck connectivity, along with a timeout period to wait a sufficient amount of time.
- Add code that obtains a flag indicating that the sync is disabled when interruption occurs because the application code disables or cancels the synchronization for any reason. For example, the application is running in the background, and you want to resume only when the application is running in the foreground.

## Log Record APIs

---

The Log Record APIs allow you to customize aspects of logging.

- Writing and retrieving log records (successful operations are not logged).
- Configuring log levels for messages reported to the console.
- Enabling the printing of server message headers and message contents, database exceptions, and `LogRecord` objects written for each import.
- Viewing detailed trace information on database calls.

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

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

### LogRecord API

`LogRecord` stores two types of logs.

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

`SMP101DB.GetLogger` – gets the log API. The client can write its own records using the log API. For example:

```
ILogger logger = SMP101DB.GetLogger();
logger.Debug("Write this string to the log records table");
SMP101DB.SubmitLogRecords();
```

`SMP101DB.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.DESENDING);
query.SortCriteria = sortCriteria;

GenericList<ILogRecord> loglist = SMP101DB.GetLogRecords(query);
```

This sample code shows how to find the corresponding MBO with the `LogRecord` and to delete the log record when a record is processed.

```
private void processLogRecords()
{
```

```

Query query = new Query();
GenericList<ILogRecord> logRecords =
SMP101DB.GetLogRecords(query);
foreach (ILogRecord log in logRecords)
{
    // log warning message
    BenchmarkUtils.AddInfo("log " + log.Component + ":"
+ log.EntityKey
+ " code:" + log.Code
+ " msg:" + log.Message);

    if (log.Component.Equals("Customer"))
    {
        long surrogateKey = Convert.ToInt64(log.EntityKey);
        Customer c = Customer.Find(surrogateKey);
        if (c.IsPending)
        {
            c.CancelPending();
        }

        log.Delete();
        log.SubmitPending();
    }
}

SMP101DB.BeginSynchronize(null, null);
}

```

A `LogRecord` is not generated for a successful operation replay. SAP Mobile Server only creates one when an operation fails or completes with warnings. The client is responsible for removing operation replay log records. SAP Mobile Server typically allows a period of time for the client to download and act on the operation replay log record. Therefore, the client should proactively remove these log records when they are consumed. Failure to do so may result in accumulation of operation replay log records until SAP Mobile Server removes them. This sample code illustrates how to find the corresponding MBO with the `LogRecord` and delete the log record when it is processed.

```

private void processLogs()
{
    Query query = new Query();
    GenericList<LogRecord> logRecords =
SMP101DB.getLogRecords(query);
    for(LogRecord log : logRecords)
    {
        // log warning message
        Log.warning("log " + log.getComponent()
+ ":" + log.getEntityKey()
+ " code:" + log.getCode() + " msg:" + log.getMessage());

        if (log.getComponent().equals("Customer"))
        {
            long surrogateKey = Long.parseLong(log.getEntityKey());
            Customer c = Customer.find(surrogateKey);
            if (c.isPending())

```

```

    {
        c.cancelPending();
    }

    // delete the LogRecord after it is processed
    log.delete();
    log.submitPending();
}
}

```

SAP Mobile Server is responsible for deleting client log records uploaded by the application. These application logs are used for audit and/or support services. Determine and set the retention policy from SAP Control Center after consulting with the application's developers. If there are multiple applications using the same package, retain them based on the maximum required time for each application. Client log records are removed that are outside the retention window, and deleted records removed from the client database the next time the application synchronizes. See *Improve Synchronization Performance by Reducing the Log Record Size* in *Troubleshooting* for details about reducing the Log Record size.

## Logger APIs

Use the `Logger` API to set the log level and create log records on the client.

Each package has a `Logger`. To obtain the package logger, use the `GetLogger` method in the generated database class. The `Logger` is an abstraction over the `LogRecord` API to write records of various log levels into the `LogRecord` MBO on the client database.

```

Logger logger = SMP101DB.GetLogger();

// set log level to debug
logger.LogLevel = LogLevel.DEBUG;

// create a log record with ERROR level and the error message.
logger.Error("Some error message");

// Prepare all outstanding client generated log records for upload
SMP101DB.SubmitLogRecords();

```

## Change Log API

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

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

## EntityType

Returns the entity type.

### **Syntax**

```
int EntityType
```

### **Parameters**

None.

### **Returns**

Returns the entity type. The entity type values are defined in the generated C# class EntityType.cs for the package.

### **Examples**

- **Get the Entity Type**

```
EntityType
```

## **OperationType**

Returns the operation type of the MBO.

### **Syntax**

```
char OperationType
```

### **Parameters**

None.

### **Returns**

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

### **Examples**

- **Get the Operation Type**

```
OperationType
```

## **RootEntityType**

Returns the name of the root parent entity type.

### **Syntax**

```
int RootEntityType
```

### **Parameters**

None.

**Returns**

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

**Examples**

- **Get the Root Entity Type**

```
RootEntityType
```

**RootSurrogateKey**

Returns the surrogate key of the root parent entity.

**Syntax**

```
long RootSurrogateKey
```

**Parameters**

None.

**Returns**

The surrogateKey of the root entity.

**Examples**

- **Get the Root Surrogate Key**

```
RootSurrogateKey
```

**SurrogateKey**

Returns the surrogate key of the entity.

**Syntax**

```
long SurrogateKey
```

**Parameters**

None.

**Returns**

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

### **Examples**

- **Get the Surrogate Key**

```
SurrogateKey
```

### **Methods in the Generated Database Class**

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

#### **EnableChangeLog**

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

#### **Syntax**

```
EnableChangeLog ();
```

#### **Returns**

None.

#### **Examples**

- **Enable Change Log**

```
SMP101DB.EnableChangeLog ();
```

#### **GetChangeLogs**

Retrieve a list of change logs.

#### **Syntax**

```
GetChangeLogs ();
```

#### **Returns**

Returns a `GenericList` of type `<Change Log>`.

#### **Examples**

- **Get Change Logs**

```
SMP101DB.GetChangeLogs ();
```



### **DeleteChangeLogs**

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

#### **Syntax**

```
DeleteChangeLogs ();
```

#### **Returns**

None.

#### **Examples**

- **Delete Change Logs**

```
SMP101DB.DeleteChangeLogs ();
```

#### **Usage**

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

```
GenericList<IChangeLog> changes = SMP101DB.GetChangeLogs(myQuery);
```

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

### **DisableChangeLog**

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

#### **Syntax**

```
DisableChangeLog ();
```

#### **Returns**

None.

### Examples

- **Disable Change Log**

```
SMP101DB.DisableChangeLog();
```

### Code Samples

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

```
SMP101DB.EnableChangeLog();
SMP101DB.Synchronize();

// Retrieve all change logs
GenericList<IChangeLog> logs = SMP101DB.GetChangeLogs(new Query());
Console.WriteLine("There are " + logs.Count + " change logs");
foreach (IChangeLog log in logs)
{
    Console.WriteLine(log.EntityType
+ "(" + log.SurrogateKey
+ "): " + log.OperationType);
}

// Retrieve only the change logs for Customer:
Query query = new Query();
AttributeTest at = new AttributeTest("entityType",
SMP101.EntityType.Customer,
AttributeTest.EQUAL);
query.TestCriteria = at;
logs = SMP101DB.GetChangeLogs(query);
Console.WriteLine("There are " + logs.size() + " change logs for
Customer");
foreach (IChangeLog log in logs)
{
    Console.WriteLine(log.EntityType
+ "(" + log.SurrogateKey
+ "): " + log.OperationType);
}
```

## Security APIs

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

### Encrypt the Database

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

The length of the encryption key cannot be fewer than 16 characters.

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.

```
ConnectionProfile profile = SMP101DB.GetConnectionProfile();
profile.EncryptionKey = "Your encryption key"; //Encryption key can
be of arbitrary length. Generally the longer, the better.
```

You can use the `generateEncryptionKey()` method to encrypt the local database with a random encryption key.

```
SMP101DB.GenerateEncryptionKey();
// store the encryption key at somewhere for reuse later
ConnectionProfile profile = SMP101DB.GetConnectionProfile();
String key = profile.EncryptionKey;
...
SMP101DB.CreateDatabase();
```

## **End to End Encryption and Compression Support APIs**

Use encryption communication parameters to ensure end to end encryption and eliminate any WAP gap security problems.

You can use the Client Object API to set up end to end encryption, supported by Ultralite, and HTTPS items in the synchronization profile. Encryption and compression are enabled by default.

Refer to the following APIs when setting up end to end encryption and compression support:

- `Sybase.Persistence.ConnectionProfile.StreamParams`
- `Sybase.Persistence.NetworkStreamParams.E2ee_Public_Key`
- `Sybase.Persistence.NetworkStreamParams.E2ee_Type`
- `Sybase.Persistence.NetworkStreamParams.Tls_Type`
- `Sybase.Persistence.NetworkStreamParams.Trusted_Certificates`
- `Sybase.Persistence.NetworkStreamParams.Trusted_Certificates`
- `Sybase.Persistence.NetworkStreamParams.Url_Suffix`
- `Sybase.Persistence.NetworkStreamParams.Zlib_Download_Window_Size`
- `Sybase.Persistence.NetworkStreamParams.Zlib_Upload_Window_Size`
- `Sybase.Persistence.NetworkStreamParams.Compression`

The following code example shows how to enable E2EE for the SMP101 package:

```
ConnectionProfile conn=SMP101DB.GetSynchronizationProfile();
conn.NetworkProtocol = "HTTP";
conn.PortNumber = 2480;
conn.StreamParams().E2ee_Type = "rsa";
conn.StreamParams().E2ee_Public_Key = "e2ee_public_key.key";
conn.Save();
```

### **DataVault**

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

If you have installed the utility files `DataVault.cs` and `DataVaultException.cs` into your application, 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-128. The `DataVault` class allows you create a named vault, and specify a password and salt used to unlock it. The password can be of arbitrary length and can include any characters. The password and salt together generate the AES key. If the user enters the same password when unlocking, the contents are decrypted. If the user enters an incorrect password, exceptions occur. If the user enters an incorrect password a configurable number of times, the vault is deleted and any data stored within it becomes unrecoverable. The vault can also relock itself after a configurable amount of time.

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

### **CreateVault**

Creates a new secure store (a vault).

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

### **Syntax**

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

## Parameters

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

## Returns

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

If a vault already exists with the same name, a `DataVaultException` is thrown 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, returns true if it does and false if the `dataVault` is locked, does not exist, or is inaccessible for any other reason.

## Syntax

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

## Parameters

- **sDataVaultID** – the vault name.

### Returns

Returns true if the vault exists; otherwise returns false.

### Examples

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

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

### vaultExists2

Tests whether the specified vault exists, returns true if the vault exists; otherwise returns false. If an error occurs while reading the keychain, throws an `kDataVaultExceptionReasonIORead` exception.

### Syntax

```
+ (BOOL)vaultExists2:(NSString*)dataVaultID;
```

### Parameters

- **dataVaultID** – the vault name.

### Returns

Returns true if the vault exists; otherwise returns false. If an error occurs while reading the keychain, throws an `kDataVaultExceptionReasonIORead` exception.

### Examples

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

```
@try {
    if ([SUPDataVault vaultExists2:@"myVault"]) {
        [SUPDataVault deleteVault:@"myVault"];
    }
}
@catch ( SUPDataVaultException *exception ) {
    //handle the exception
}
```

### 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.

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");
}
```

**GetDataNames**

Retrieves information about the data names stored in the vault.

The application can pass the data names to `GetValue` or `GetString` to retrieve the data values.

**Syntax**

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

**Parameters**

None.

### Returns

Returns a `DVPasswordPolicy` object, as an array of `DVDataName` structure objects.

### Examples

- **Get data names**

```
// Call getDataNames to retrieve all stored element names from our
data vault.
DataVault.DVDataName[] dataNameArray = oDataVault.GetDataNames();
for ( int i = 0; i < dataNameArray.GetLength( 0 ); i++ )
{
    if ( dataNameArray[ i ].type == DataVault.DVDataType.String )
    {
        String thisStringValue =
oDataVault.GetString( dataNameArray[ i ].name );
    }
    else if ( dataNameArray[ i ].type ==
DataVault.DVDataType.Binary )
    {
        byte[] thisBinaryValue =
oDataVault.GetValue( dataNameArray[ i ].name );
    }
}
```

### SetPasswordPolicy

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

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

### Syntax

```
public void SetPasswordPolicy(DataVault.PasswordPolicy oPolicy)
```

### Parameters

- `oPolicy` – the password policy constraints.

### Examples

- **Set a password policy**

```
// SetPasswordPolicy() will always lock the vault to ensure the
old password
// conforms to the new password policy settings.
oDataVault.SetPasswordPolicy( oPasswordPolicy );
```



Password Policy Structure

A structure defines the policy used to generate the password.

**Table 1. Password Policy Structure**

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

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

### *Settings for Password Policy*

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

---

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

---

- **PROP\_DEF\_PWDPOLICY\_ENABLED** – Boolean property with a default value of false. Indicates if a password policy is enabled by the administrator.
- **PROP\_DEF\_PWDPOLICY\_DEFAULT\_PASSWORD\_ALLOWED** – Boolean property with a default value of false. Indicates if the client application is allowed to use the default password for the data vault.
- **PROP\_DEF\_PWDPOLICY\_MIN\_LENGTH** – Integer property with a default value of 0. Defines the minimum length for the password.
- **PROP\_DEF\_PWDPOLICY\_HAS\_DIGITS** – Boolean property with a default value of false. Indicates if the password must contain digits.
- **PROP\_DEF\_PWDPOLICY\_HAS\_UPPER** – Boolean property with a default value of false. Indicates if the password must contain at least one uppercase character.
- **PROP\_DEF\_PWDPOLICY\_HAS\_LOWER** – Boolean property with a default value of false. Indicates if the password must contain at least one lowercase character.
- **PROP\_DEF\_PWDPOLICY\_HAS\_SPECIAL** – Boolean property with a default value of false. Indicates if the password must contain at least one special character. A special character is a character in this set “~!@#\$\$%^&\*()-+”.

- **PROP\_DEF\_PWDPOLICY\_EXPIRATION\_DAYS** – Integer property with a default value of 0. Specifies the number of days in which password will expire from the date of setting the password. Password expiration is checked only when the vault is unlocked.
- **PROP\_DEF\_PWDPOLICY\_MIN\_UNIQUE\_CHARS** – Integer property with a default value of 0. Specifies minimum number of unique characters in the password. For example, if minimum length is 5 and minUniqueChars is 4 then “aaate” or “ababa” would be invalid passwords. Instead, “aard” would be a valid password.
- **PROP\_DEF\_PWDPOLICY\_LOCK\_TIMEOUT** – Integer property with a default value of 0. Specifies timeout value (in seconds) after which the vault is locked from the unlock time. 0 indicates no timeout.
- **PROP\_DEF\_PWDPOLICY\_RETRY\_LIMIT** – Integer property with a default value of 0. Specifies the number of failed unlock attempts after which data vault is deleted. 0 indicates no retry limit.

### Password Errors

Password policy violations cause exceptions to be thrown.

**Table 2. Password Errors**

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

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

**GetPasswordPolicy**

Retrieves the password policy set by `SetPasswordPolicy`.

Use this method once the `DataVault` is unlocked.

**Syntax**

```
public DataVault.PasswordPolicy GetPasswordPolicy()
```

**Parameters**

None.

**Returns**

Returns a `PasswordPolicy` structure that contains the policy set by `SetPasswordPolicy`.

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

**Examples**

- **Get the current password policy**

```
// Call getPasswordPolicy() to return the current password policy
settings.
    DataVault.PasswordPolicy oCurrentPolicy =
oDataVault.GetPasswordPolicy();
```

**IsDefaultPasswordUsed**

Checks whether the default password is used by the vault.

Use this method once the `DataVault` is unlocked.

**Syntax**

```
public bool IsDefaultPasswordUsed()
```

**Returns**

Returns	Indicates
true	Both the default password and the default salt are used to encrypt the vault.
false	Either the default password or the default salt are not used to encrypt the vault.

**Examples**

- **Check if default password used**

```
// Call isDefaultPasswordUsed() to see if we are using an
// automatically
// generated password (which we are).
bool isDefaultPasswordUsed = oDataVault.IsDefaultPasswordUsed();
```

This code example lacks exception handling. For a code example that includes exception handling, see *Developer Guide: Windows and Windows Mobile Object API Applications* > *Client Object API Usage* > *Security APIs* > *DataVault* > *Code Sample*.

**Lock**

Locks the vault.

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

**Syntax**

```
public void Lock()
```

**Examples**

- **Locks the data vault** – prevents changing the vaults properties or stored content.

```
vault.Lock();
```

**IsLocked**

Checks whether the vault is locked.

**Syntax**

```
public bool IsLocked()
```

**Returns**

Returns	Indicates
true	The vault is locked.

Returns	Indicates
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 attempts exceeds the retry limit, the vault is deleted.

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

### **Syntax**

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

### **Parameters**

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

### **Returns**

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

### **Examples**

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

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

### **SetString**

Stores a string object in the vault.

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 ensures 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.

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**

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

### Examples

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

```
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.

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 ensures 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.

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**

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

### **Examples**

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

```

try
{
    vault.Unlock("password", "salt");
    byte[] retrievedvalue = vault.GetValue("testValue");
}
catch (DataVaultException e)
{
    Console.WriteLine("Exception: " + e.ToString());
}
finally
{
    vault.Lock();
}

```

### **DeleteValue**

Deletes the specified value.

### **Syntax**

```
public static void DeleteValue(string sName)
```

### Parameters

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

### Examples

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

```
DataVault.DeleteValue("myValue");
```

### ChangePassword (two parameters)

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

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

### Syntax

```
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 ensures 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();  
}
```

### ChangePassword (four parameters)

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

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

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

### Syntax

```
public void ChangePassword(
    string sOldPassword,
    string sOldSalt,
    string sNewPassword,
    string sNewSalt
)
```

### Parameters

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

### Examples

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

```
// Call changePassword with four parameters, even if the vault is
// locked.
// Pass null for oldSalt and oldPassword if the defaults were
// used.
oDataVault.ChangePassword( null, null, "password!1A",
    "saltD#ddg#k05%gnd[!1A" );
```

### Code Sample

Create a data vault for encrypted storage of application data.

```
public void DataVaultSampleCode()
{
    try
    {
        DataVault oDataVault = null;

        // If this dataVault already exists, then get it by calling
        getVault()
        // Else create this new dataVault by calling createVault()
        if ( DataVault.VaultExists( "DataVaultExample" ) )
            oDataVault = DataVault.GetVault( "DataVaultExample" );
    }
}
```

```

        else
            oDataVault = DataVault.CreateVault( "DataVaultExample",
"password!1A", "saltD#ddg#k05%gnd[!1A" );

            // Call setLockTimeout(). This allows you to set the timeout
of the vault in seconds
            oDataVault.SetLockTimeout( 1500 );
            int iTimeout = oDataVault.GetLockTimeout();

            // Call setRetryLimit(). This allows you to set the number of
retries before the vault is destroyed
            oDataVault.SetRetryLimit( 10 );
            int iRetryLimit = oDataVault.GetRetryLimit();

            // Call setPasswordPolicy(). The passwordPolicy also includes
the retryLimit and LockTimeout that we set above.
            DataVault.PasswordPolicy oPasswordPolicy = new
DataVault.PasswordPolicy();
            oPasswordPolicy.defaultPasswordAllowed = true;
            oPasswordPolicy.minimumLength = 4;
            oPasswordPolicy.hasDigits = true;
            oPasswordPolicy.hasUpper = true;
            oPasswordPolicy.hasLower = true;
            oPasswordPolicy.hasSpecial = true;
            oPasswordPolicy.expirationDays = 20;
            oPasswordPolicy.minUniqueChars = 3;
            oPasswordPolicy.lockTimeout = 1600;
            oPasswordPolicy.retryLimit = 20;

            // SetPasswordPolicy() will always lock the vault to ensure
the old password
            // conforms to the new password policy settings.
            oDataVault.SetPasswordPolicy( oPasswordPolicy );

            // We are now locked and need to unlock before we can access
the vault.
            oDataVault.Unlock( "password!1A", "saltD#ddg#k05%gnd[!1A" );

            // Call getPasswordPolicy() to return the current password
policy settings.
            DataVault.PasswordPolicy oCurrentPolicy =
oDataVault.GetPasswordPolicy();

            // Call setString() by giving it a name:value pair to encrypt
and persist
            // a string data type within your dataVault.
            oDataVault.SetString( "stringName", "stringValue" );

            // Call getString to retrieve the string we just stored in our
data vault!
            String storedStringValue =
oDataVault.GetString( "stringName" );

            // Call setValue() by giving it a name:value pair to encrypt
and persist
            // a binary data type within your dataVault.

```

```

        byte[] binaryValue = { 1, 2, 3, 4, 5, 6, 7 };
        oDataVault.SetValue( "binaryName", binaryValue );

        // Call getValue to retrieve the binary we just stored in our
data vault!
        byte[] storedBinaryValue =
oDataVault.GetValue( "binaryName" );

        // Call getDataNames to retrieve all stored element names
from our data vault.
        DataVault.DVDataName[] dataNameArray =
oDataVault.GetDataNames();
        for ( int i = 0; i < dataNameArray.GetLength( 0 ); i++ )
        {
            if ( dataNameArray[ i ].type ==
DataVault.DVDataType.String )
            {
                String thisStringValue =
oDataVault.GetString( dataNameArray[ i ].name );
            }
            else if ( dataNameArray[ i ].type ==
DataVault.DVDataType.Binary )
            {
                byte[] thisBinaryValue =
oDataVault.GetValue( dataNameArray[ i ].name );
            }
        }

        // Call changePassword with 2 parameters. Vault must be
unlocked.
        // If you pass null parameters as your new password or your
new salt,
        // it will generate a default password or default salt,
respectively.
        oDataVault.ChangePassword( null, null );

        // Call isDefaultPasswordused() to see if we are using an
automatically
        // generated password (which we are).
        bool isDefaultPasswordUsed =
oDataVault.IsDefaultPasswordUsed();

        // Lock the vault.
        oDataVault.Lock();

        // Call changePassword with 4 parameters even if the vault is
locked.
        // Here, we pass null for oldSalt and oldPassword because
defaults were used.
        oDataVault.ChangePassword( null, null, "password!1A",
"saltD#ddg#k05%gnd[!1A" );

        // Call isDefaultPasswordused() and we will see that the
default password is NOT used anymore.
        isDefaultPasswordUsed = oDataVault.IsDefaultPasswordUsed();
    }

```

```
catch ( DataVaultException dvex )
{
    // Handle datavault exception.
}
catch ( Exception ex )
{
    // Handle exception.
}
finally
{
    // Because this is a test example, we will delete our vault at
the end.
    // This means we will forever lose all data we persisted in
our data vault.
    if ( DataVault.VaultExists( "DataVaultExample" ) )
        DataVault.DeleteVault( "DataVaultExample" );
}
}
```

## Callback and Listener APIs

---

The callback and listener APIs allow you to optionally register a callback handler and listen for device events, application connection events, and package synchronize and replay events.

### ICallbackHandler API

The `ICallbackHandler` interface is invoked when any database event occurs. A default callback handler is provided, which basically does nothing. You should implement a custom `CallbackHandler` to register important events. The callback is invoked on the thread that is processing the event. To receive callbacks for database changes, 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 `Sybase.Persistence.ICallbackHandler` interface.

To allow the `CallbackHandler` to be called, you must use asynchronous synchronization, for example by calling `BeginSynchronize()` instead of `Synchronize()`.

In your handler, override the particular callback that you are interested in (for example, `void OnReplayFailure(Object entity)`). 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.

**Table 3. Callbacks in the ICallbackHandler Interface**

Callback	Description
<pre>void OnImport (Object entity)</pre>	<p>This method is invoked when an import message is successfully applied to the local database. However, it is not committed. One message from server may have multiple import entities and they would be committed in one transaction for the whole message.</p> <hr/> <p><b>Note:</b></p> <ol style="list-style-type: none"> <li>1. Stale data may be read from the database at this time before commit of the whole message. Developers are encouraged to wait until the next <code>OnTransactionCommit ()</code> is invoked, then to read from the database to obtain the updated data.</li> <li>2. Both <code>CallbackHandlers</code> registered for the MBO class of the entity and Package DB will be invoked.</li> </ol> <hr/> <p><b>Parameters:</b></p> <ul style="list-style-type: none"> <li>• <b>entity</b> – the Mobile Business Object that was just imported.</li> </ul>
<pre>void OnLoginFailure ()</pre>	<p>This method will be invoked when login failed for a <code>beginOnlineLogin</code> call.</p> <hr/> <p><b>Note:</b> Only the <code>CallbackHandler</code> registered for package DB will be invoked.</p>
<pre>void OnLoginSuccess ()</pre>	<p>This method is invoked when login succeeds for a <code>beginOnlineLogin</code> call.</p> <hr/> <p><b>Note:</b> Only the <code>CallbackHandler</code> registered for package DB is invoked.</p>

Callback	Description
<pre>void OnReplayFailure (Object entity)</pre>	<p>This method is invoked when a replay request fails.</p> <hr/> <p><b>Note:</b> CallbackHandlers registered for both the MBO class of the entity and the Package DB are invoked.</p> <hr/> <p><b>Parameters:</b></p> <ul style="list-style-type: none"> <li>• <b>entity</b> – the Mobile Business Object to replay.</li> </ul>
<pre>void OnReplaySuccess (Object entity)</pre>	<p>This method is invoked when a replay request succeeds. OnReplaySuccess is an MBO object instance that contains the data prior to the synchronization. You can use the Change Log API to find records that occur after the synchronization.</p> <hr/> <p><b>Note:</b> CallbackHandlers registered for both the MBO class of the entity and the Package DB are invoked.</p> <hr/> <p><b>Parameters:</b></p> <ul style="list-style-type: none"> <li>• <b>entity</b> – the Mobile Business Object to replay.</li> </ul>
<pre>void OnSearchFailure (Object entity)</pre>	<p>This method is invoked when a back-end search fails.</p> <hr/> <p><b>Note:</b> CallbackHandlers registered for both the MBO class of the entity and the Package DB are invoked.</p> <hr/> <p><b>Parameters:</b></p> <ul style="list-style-type: none"> <li>• <b>entity</b> – the back-end search object.</li> </ul>
<pre>void OnSearchSuccess (Object entity)</pre>	<p>This method is invoked when a back end search succeeds.</p> <hr/> <p><b>Note:</b> CallbackHandlers registered for both the MBO class of the entity and the Package DB are invoked.</p> <hr/> <p><b>Parameters:</b></p> <ul style="list-style-type: none"> <li>• <b>entity</b> – the back-end search object.</li> </ul>



Callback	Description
<pre>void OnSubscribeFailure ()</pre>	<p>This method is invoked when subscribe fails.</p> <hr/> <p><b>Note:</b> CallbackHandlers registered for both the MBO class of the entity and the Package DB are invoked.</p> <hr/>
<pre>void OnSubscribeSuccess ()</pre>	<p>This method is invoked when subscribe succeeds.</p> <hr/> <p><b>Note:</b> Only the CallbackHandler registered for the Package DB is invoked.</p> <hr/>
<pre>SynchronizationAction On- Synchronize( Generic- List&lt;IsynchronizationGroup&gt; groups, SynchronizationCon- text context)</pre>	<p>This method is invoked at different stages of the synchronization. This method is called by the database class Synchronize or BeginSynchronize methods when the client initiates a synchronization, and is called again when the server responds to the client that synchronization has finished, or that synchronization failed. The status of the synchronization context, context.Status, specifies the stage of the synchronization.</p> <p><b>Parameters:</b></p> <ul style="list-style-type: none"> <li>• <b>groups</b> – a list of synchronization groups.</li> <li>• <b>context</b> – the synchronization context.</li> </ul> <p><b>Returns:</b> Either SynchronizationAction.CONTINUE or SynchronizationAction.CANCEL. If SynchronizationAction.CANCEL is returned, the synchronize is cancelled if the status of the synchronization context is one of the following.</p> <ul style="list-style-type: none"> <li>• SynchronizationStatus.STARTING</li> <li>• SynchronizationStatus.ASYNC_REPLAY_COMPLETED</li> <li>• SynchronizationStatus.STARTING_ON_NOTIFICATION</li> </ul> <p>The return value has no effect if the status is not in the above list.</p>

Callback	Description
void OnSuspendSubscriptionFailure()	<p>This method is invoked when suspend subscription fails.</p> <hr/> <p><b>Note:</b> Only the CallbackHandler registered for the Package DB is invoked.</p>
void OnSuspendSubscriptionSuccess()	<p>This method is invoked when suspend subscription succeeds.</p> <hr/> <p><b>Note:</b> Only the CallbackHandler registered for the Package DB is invoked.</p>
void OnResumeSubscriptionFailure()	<p>This method is invoked when resume subscription fails.</p> <hr/> <p><b>Note:</b> Only the CallbackHandler registered for the Package DB is invoked.</p>
void OnResumeSubscriptionSuccess()	<p>This method is invoked when resume subscription succeeds.</p> <hr/> <p><b>Note:</b> Only the CallbackHandler registered for the Package DB is invoked.</p>
void OnUnsubscribeFailure()	<p>This method is invoked when unsubscribe fails.</p> <hr/> <p><b>Note:</b> Only the CallbackHandler registered for the Package DB is invoked.</p>
void OnUnsubscribeSuccess()	<p>This method is invoked when unsubscribe succeeds.</p> <hr/> <p><b>Note:</b> Only the CallbackHandler registered for the Package DB is invoked.</p>

Callback	Description
<pre>void OnMessageException (Exception ex)</pre>	<p>This method is invoked when an exception occurs in the processing of a message.</p> <hr/> <p><b>Note:</b> In <code>DefaultCallbackHandlers</code>, <code>onMessageException</code> re-throws the <code>Exception</code> so that the messaging layer can retry the message. The application developer has the option to implement a custom <code>CallbackHandler</code> that does not re-throw the exception, based on exception types or other conditions, so that the message is not retried.</p> <hr/> <p><b>Parameters:</b></p> <ul style="list-style-type: none"> <li>• <b>ex</b> – the exception thrown when processing a message.</li> </ul>
<pre>void OnTransactionCommit ()</pre>	<p>This method is invoked after a message is processed and committed.</p> <hr/> <p><b>Note:</b> Only the <code>CallbackHandler</code> registered for the <code>Package DB</code> is invoked.</p>
<pre>void OnTransactionRollback ()</pre>	<p>This method is invoked after a message is rolled back. It only happens when an <code>Exception</code> was thrown when processing the message, or from a custom <code>Callback</code> method.</p> <hr/> <p><b>Note:</b> Only the <code>CallbackHandler</code> registered for the <code>Package DB</code> is invoked.</p>
<pre>void OnResetSuccess ()</pre>	<p>This method is invoked when all data is cleared by the reset.</p> <hr/> <p><b>Note:</b> Only the <code>CallbackHandler</code> registered for the <code>Package DB</code> is invoked.</p>
<pre>void OnRecoverSuccess ()</pre>	<p>This method is invoked when recover succeeds.</p> <hr/> <p><b>Note:</b> Only the <code>CallbackHandler</code> registered for the <code>Package DB</code> is invoked.</p>
<pre>void OnRecoverFailure ()</pre>	<p>This method is invoked when recover fails.</p> <hr/> <p><b>Note:</b> Only the <code>CallbackHandler</code> registered for the <code>Package DB</code> is invoked.</p>

Callback	Description
<code>void OnSubscriptionEnd()</code>	<p>This method is invoked when a subscription is re-registered or unsubscribed. This method deletes all MBO data on the device.</p> <hr/> <p><b>Note:</b> Only the CallbackHandler registered for the Package DB is invoked.</p>
<code>void OnImportSuccess()</code>	<p>This method is invoked when all data has been successfully imported.</p> <hr/> <p><b>Note:</b> Only the CallbackHandler registered for the Package DB is invoked.</p>
<code>void BeforeImport(Object entity)</code>	<p>This method is invoked before importing the specified entity.</p> <hr/> <p><b>Note:</b> Only the CallbackHandler registered for the Package DB is invoked.</p> <hr/> <p><b>Parameters:</b></p> <ul style="list-style-type: none"> <li>• <b>entity</b> – the Mobile Business Object to be imported.</li> </ul>

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

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

MyCallbackHandler handler = new MyCallbackHandler();
MyPackageDB.RegisterCallbackHandler(handler);
```

## IApplicationCallback API

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

You must register an `IApplicationCallback` implementation to your `Sybase.Mobile.Application` instance to receive these callbacks.

---

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

---

**Table 4. Callbacks in the `IApplicationCallback` Interface**

Callback	Description
<pre>void OnApplicationSettingsChanged(Sybase.Collections.StringList names)</pre>	<p>Invoked when one or more application settings have been changed by the server administration.</p>
<pre>void OnConnectionStatusChanged(int connectionStatus, int errorCode, string errorMessage)</pre>	<p>Invoked when the connection status changes. The possible connection status values are defined in the <code>ConnectionStatus</code> class.</p> <hr/> <p><b>Note:</b> Some of the connection status codes are not returned on certain client platforms due to platform operating system limitations.</p>
<pre>void OnDeviceConditionChanged(int deviceCondition)</pre>	<p>Invoked when a condition is detected on the mobile device that may be of interest to the application or the application user. The possible device condition values are defined in the <code>DeviceCondition</code> class.</p>
<pre>void OnRegistrationStatusChanged(int registrationStatus, int errorCode, string errorMessage)</pre>	<p>Invoked when the registration status changes. The possible registration status values are defined in the <code>RegistrationStatus</code> class.</p>
<pre>void OnHttpCommunicationError(int errorCode, string errorMessage, Sybase.Collections.StringProperties responseHeaders);</pre>	<p>Invoked when an HTTP communication server/MobiLink rejects HTTP/MobiLink communication with an error code.</p> <ul style="list-style-type: none"> <li>• <b>errorCode</b> – Error code returned by the HTTP server or MobiLink. For example: code 401 for authentication failure, code 403 for authorization failure, and code 63 for MobiLink synchronization communication error.</li> <li>• <b>errorMessage</b> – Error message returned by the HTTP server or MobiLink.</li> <li>• <b>responseHeaders</b> – Response headers returned by the HTTP server or MobiLink.</li> </ul>

Callback	Description
<pre>void OnCustomizationBundle- DownloadComplete(string cus- tomizationBundleID, int er- rorCode, string errorMes- sage);</pre>	<p>Invoked when the download of a resource bundle is complete.</p> <ul style="list-style-type: none"> <li>• <b>errorCode</b> – If download succeeds, returns 0. If download fails, returns an error code.</li> <li>• <b>errorMessage</b> – If download succeeds, returns "". If download fails, returns an error message. <ul style="list-style-type: none"> <li>• RESOURCE_BUNDLE_NOTFOUND = 14881</li> <li>• DOWNLOAD_RESOURCE_BUNDLE_STREAM_IS_NULL = 14882</li> <li>• DOWNLOAD_RESOURCE_BUNDLE_FAILURE = 14883</li> </ul> </li> <li>• <b>customizationBundleID</b> – The name of the resource bundle. If null, the default application resource bundle is downloaded.</li> </ul>
<pre>int OnPushNotification (Hashtable notification);</pre>	<p>Invoked if a push notification arrives. You can add logic here to handle the notification. This callback is not called when a notification arrives when the application is not online.</p> <ul style="list-style-type: none"> <li>• <b>returns</b> – an integer to indicate if the notification has been handled. The return value is for future use. You are recommended to return NOTIFICATION_CONTINUE. <ul style="list-style-type: none"> <li>• 0: NOTIFICATION_CONTINUE if the notification was not handled by the callback method.</li> <li>• 1: NOTIFICATION_CANCEL if the notification has already been handled by the callback method.</li> </ul> </li> </ul>

## SyncStatusListener API

You can implement a synchronization status listener to track synchronization progress.

---

**Note:** This topic is not applicable for DOE-based applications.

---

Create a listener that implements the `SyncStatusListener` interface.

```
public interface SyncStatusListener
{
    bool ObjectSyncStatus(ObjectSyncStatusData statusData);
}

public class MySyncListener : SyncStatusListener
{
    // implementation
}
```

Pass an instance of the listener to the synchronize methods.

```
MySyncListener listener = new MySyncListener();
SMP101DB.synchronize("sync_group", listener);
// or SMP101DB.synchronize(listener); if we want to synchronize all
// synchronization groups
```

As the application synchronization progresses, the `ObjectSyncStatus` method defined by the `SyncStatusListener` interface is called and is passed an `ObjectSyncStatusData` object. The `ObjectSyncStatusData` 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 `ObjectSyncStatusData` object and comparing it to the possible values in the `SyncStatusState` enumeration, the application can react accordingly to the state of the synchronization.

The method returns `false` to allow synchronization to continue. If the method returns `true`, the synchronization is aborted.

Possible uses of `ObjectSyncStatus` method include changing form elements on the client screen to show synchronization progress, such as 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:** The `ObjectSyncStatus` method of `SyncStatusListener` is called and executed in the data synchronization thread. If a client runs synchronizations in a thread other than the primary user interface thread, the client cannot update its screen as the status changes. The client must instruct the primary user interface thread to update the screen regarding the current synchronization status.

---

This is an example of `SyncStatusListener` implementation:

```
public class SyncListener : SyncStatusListener
{
    public bool ObjectSyncStatus(ObjectSyncStatusData data)
    {
        switch (data.SyncStatusState) {
            case SyncStatusState.ApplicationSyncDone:
                //implement your own UI indicator bar
                break;
            case SyncStatusState.ApplicationSyncError:
                //implement your own UI indicator bar
        }
    }
}
```

```
        break;
    case SyncStatusState.SyncDone:
        //implement your own UI indicator bar
        break;
    case SyncStatusState.SyncStarting:
        //implement your own UI indicator bar
        break;
    ...
}
return false;
}
```

## Query APIs

---

The Query API allows you to retrieve data from mobile business objects, to page data, and to retrieve a query result by filtering. You can also use the Query API to filter children MBOs of a parent MBO in a one to many relationship.

### Retrieving Data from Mobile Business Objects

You can retrieve data from mobile business objects through a variety of queries, including object queries, arbitrary find, and through filtering query result sets.

#### Object Queries

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

Object Query methods are generated based on the object queries defined by the modeler in SAP Mobile WorkSpace. Object Query methods carry query names, parameters, and return types defined in SAP Mobile 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 SAP Mobile WorkSpace:

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

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

```
Sybase.Collections.GenericList<Customer> customers =
Customer.FindByFirstName("fname");
```

### **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 5. Query and Related Classes**

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

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

#### **Arbitrary Find**

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

The arbitrary find method also lets the user specify a desired ordering of the results and object state criteria. A `Query` class is included in the client object API. 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.

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`.

### *TestCriteria*

You can construct a query SQL statement to query data from a local database. You can create a `TestCriteria` object (in this example, `AttributeTest`) to filter results. You can also query across multiple tables (MBOs) when using the `ExecuteQuery` API.

```
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 = SMP101DB.ExecuteQuery(query2);
```

### *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
- NOT\_START\_WITH
- NOT\_END\_WITH
- 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.Operator = 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.Operator = Sybase.Persistence.AttributeTest.EQUAL;
existQuery.Where(test1);
test.Value = existQuery;
test.SetOperator(Sybase.Persistence.AttributeTest.EXISTS);
query.Where(test);
Sybase.Persistence.QueryResultSet qs = SMP101DB.ExecuteQuery(query);
```

### *SortCriteria*

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

For example,

```
Query query = new Query();

query.Select("c.lname, c.fname");
query.From("Customer", "c");

AttributeTest aTest = new AttributeTest();
aTest.Attribute = "state";
aTest.TestValue = "CA" ;
aTest.TestType = AttributeTest.EQUAL;
query.TestCriteria = aTest;

SortCriteria sort = new SortCriteria();
sort.Add("lname", SortOrder.ASCENDING);
sort.Add("fname", SortOrder.ASCENDING);
query.SortCriteria = sort;
```

### *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;

GenericList<Customer> customers = Customer.FindWithQuery(props);
```

### 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

Aggregate Function	Supported Datatypes
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.Attribute = "c.id";
ts.TestValue = "0";
ts.Operator = AttributeTest.GREATER_EQUAL;
query2.Where(ts);
query2.GroupBy("c.id");

ts2 = new AttributeTest();
ts2.Attribute = "c.id";
ts2.TestValue = "0";
ts2.Operator = 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);
SMP101DB.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 =
SMP101DB.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.Operator = 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 =
SMP101DB.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);
```

---

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

---

### QueryResultSet

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

The following example shows how to filter a result set and get values by taking data from two mobile business objects, creating a `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 = SMP101DB.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. A bidirectional relationship also allows the child MBO to access the associated parent MBO.

Assume there are two MBOs defined in SAP Mobile Server. One MBO is called Customer and contains a list of customer data records. The second MBO is called SalesOrder and contains order information. Additionally, assume there is an association between Customers and Orders on the customer ID column. The Orders application is parameterized to return order information for the customer ID.

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

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

## Persistence APIs

The persistence APIs include operations and object state APIs.

### Operations APIs

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

Any parameters in the create, update, or delete operation that are mapped to the object's attributes are handled internally by the client object API, and are not exposed. Any parameters not mapped to the object's attributes are left as parameters in the generated object API. The code examples for create, update, and delete operations are based on the **fill from attribute** being set. Different MBO settings affect the operation methods.

---

**Note:** If the SAP Mobile Platform object model defines one instance of a create operation and one instance of an update operation, and all operation parameters are mapped to the object's attributes, then a `Save` method can be automatically generated which, when called internally, determines whether to insert or update data to the local client-side database. In other situations, where there are multiple instances of create or update operations, methods such as `Save` cannot be automatically generated.

---

### Create Operation

The `Create` operation allows the client to create a new record in the local database. To execute a create operation on an MBO, create a new MBO instance, and set the MBO

attributes, then call the `Save()` or `Create()` operation. To propagate the changes to the server, call `SubmitPending()`.

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

### **Update Operation**

The `Update` operation updates a record in the local database on the device. To execute update operations on an MBO, get an instance of the MBO, set the MBO attributes, then call either the `Save()` or `Update()` operation. To propagate the changes to the server, call `SubmitPending()`.

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

To update multiple MBOs in a relationship, if the relationship is a composite, call `SubmitPending()` on the parent MBO. If the relationship is not a composite, call `SubmitPending()` on each MBO within the relationship:

```
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();
```

### **Delete Operation**

The `Delete` operation allows the client to delete a new record in the local database. To execute delete operations on an MBO, get an instance of the MBO, set the MBO attributes, then call the `Delete` operation. To propagate the changes to the server, call `SubmitPending()`.

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

For MBOs in a relationship, perform a delete as follows:

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

### **Save Operation**

The Save operation saves a record to the local database. In the case of an existing record, a Save operation calls the Update operation. If a record does not exist, the Save operation creates a new record.

```
//Update an existing customer
Customer cust = Customer.FindById(101);
cust.Save();

//Insert a new customer
Customer cust = new Customer();
cust.Save();
```

### **Other Operation**

Operations other than Create, Update, or Delete operations are called "other" operations. An Other operation class is generated for each operation in the MBO that is not a Create, Update, or Delete operation.

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();
other.SubmitPending();
```

### **Cascade Operations**

Composite relationships are cascaded. Cascade operations allow a single synchronization to execute a chain of related CRUD operations. Multi-level insert is a special case for cascade operations. Multilevel insert allows a single synchronization to execute a chain of related insert operations, creating parent and children objects.

#### *Multilevel Insert*

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();
```

See the SAP Mobile Platform online documentation for information on defining relationships that support cascading (composite) operations, and for specific multilevel insert requirements.

### **Pending Operation**

You can manage the pending state.

You can group multiple operations into a single transaction for improved performance:

```
// load the customer MBO with customer ID 100
Customer customer = Customer.FindByPrimaryKey(100);

// Change phone number of that customer
customer.Phone = "8005551212";

// use one transaction to do save and submitPending
Sybase.Persistence.LocalTransaction tx =
SMP101DB.BeginTransaction();
try
{
    customer.Save();
    customer.SubmitPending();
    tx.Commit();
}
catch (Exception e)
{
    tx.Rollback();
}
```

- **SubmitPending** – submits the operation so that it can be replayed on the SAP Mobile Server. A request is sent to the SAP Mobile Server during a synchronization.

- **CancelPending** – cancels the previous create, update, or delete operations on the MBO. It cannot cancel submitted operations.

CancelPending cancels pending changes for a particular instance or instances (via CancelPendingObjects from the database class). However, if SubmitPending has already been invoked, only the pending state and original state (for update) are removed. The operation replay record generated by the SubmitPending remains. This means that the operation replay record is uploaded to SAP Mobile Server upon synchronization. If the EIS honors the operation replay, the changes are propagated back to the device during the download. The Object API framework forgoes operation replay completion processing when it finds that there are no pending/original states for the instance. Hence, CancelPending is not the inverse operation of submitPending.

- **SubmitPendingOperations** – submits all the pending records for the entity to the SAP Mobile 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.

---

**Note:** Use the SubmitPendingOperations and CancelPendingOperations methods only when there are multiple pending entities on the same MBO type. Otherwise, use the MBO instance's SubmitPending or CancelPending methods, which are more efficient if the MBO instance is already available in memory.

---

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

## Object State APIs

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

### Entity State Management

The object state APIs provide methods for returning information about entities in the database.

All entities that support pending state have the following attributes:

Name	Type	Description
IsNew	bool	Returns true if this entity is new, but has not yet been created in the client database.
IsCreated	bool	Returns true if this entity has been newly created in the client database, and one of the following is true: <ul style="list-style-type: none"> <li>• The entity has not yet been submitted to the server with a replay request.</li> <li>• The entity has been submitted to the server, but the server has not finished processing the request.</li> <li>• The server rejected the replay request (<code>replay-Failure</code> message received).</li> </ul>
IsDirty	bool	Returns true if this entity has been changed in memory, but the change has not yet been saved to the client database.
IsDeleted	bool	Returns true if this entity was loaded from the database and subsequently deleted.
IsPending	bool	Checks if the object's pending flag is turned on or not, i.e. has pending change or not. Returns true if there is a pending change, returns false if there is no pending change.
IsUpdated	bool	Returns true if this entity has been updated or changed in the database, and one of the following is true: <ul style="list-style-type: none"> <li>• The entity has not yet been submitted to the server with a replay request.</li> <li>• The entity has been submitted to the server, but the server has not finished processing the request.</li> <li>• The server rejected the replay request (<code>replay-Failure</code> message received).</li> </ul>
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.

Name	Type	Description
PendingChange	char	If pending is true, this attribute's value is 'C' (create), 'U' (update), 'D' (delete), or 'P' (to indicate that this MBO is a parent in a cascading relationship for one or more pending child objects, but this MBO itself has no pending create, update or delete operations). If pending is false, this attribute's value is 'N'.
ReplayCounter	long	Returns a long value that is updated each time a row is created or modified by the client. This value is a unique value obtained from <code>KeyGenerator.GenerateID</code> method. Note that the value increases every time it is retrieved.
ReplayPending	long	Returns a long value. When a pending row is submitted to the server, the value of <code>ReplayCounter</code> is copied to <code>ReplayPending</code> . 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 <code>ReplayCounter</code> is greater than <code>ReplayPending</code> ).
ReplayFailure	long	Returns a long value. When the server responds with a <code>ReplayFailure</code> message for a row that was submitted to the server, the value of <code>ReplayCounter</code> is copied to <code>ReplayFailure</code> , and <code>ReplayPending</code> is set to 0.

### Entity State Example

Shows how the values of the entities that support pending state change at different stages during the MBO update process. The values that change between different states appear in bold.

Note these entity behaviors:

- The `IsDirty` flag is set if the entity changes in memory but is not yet written to the database. Once you save the MBO, this flag clears.
- The `ReplayCounter` value that gets sent to the SAP Mobile Server is the value in the database before you call `SubmitPending`. After a successful replay, that value is imported from the SAP Mobile 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
<p>After reading from the database, before any changes are made.</p>	<p>IsNew=false                      IsCreated=false                      IsDirty=false                      IsDeleted=false                      IsUpdated=false                      Pending=false                      PendingChange='N'                      ReplayCounter=33422977                      ReplayPending=0                      ReplayFailure=0</p>
<p>One or more attributes are changed, but changes not saved.</p>	<p>IsNew=false                      IsCreated=false                      IsDirty=<b>true</b>                      IsDeleted=false                      IsUpdated=false                      Pending=false                      PendingChange='N'                      ReplayCounter=33422977                      ReplayPending=0                      ReplayFailure=0</p>



Description	Flags/Values
<p>After <code>entity.Save()</code> [entity save] or <code>entity.Update()</code> [entity update] is called.</p>	<p>IsNew=false  IsCreated=false  IsDirty=<b>false</b>  IsDeleted=false  IsUpdated=<b>true</b>  Pending=<b>true</b>  PendingChange='U'  ReplayCounter=<b>33424979</b>  ReplayPending=0  ReplayFailure=0</p>
<p>After <code>entity.SubmitPending()</code> [entity submitPending] is called to submit the MBO to the server.</p>	<p>IsNew=false  IsCreated=false  IsDirty=false  IsDeleted=false  IsUpdated=true  Pending=true  PendingChange='U'  ReplayCounter=33424981  ReplayPending=<b>33424981</b>  ReplayFailure=0</p>

Description	Flags/Values
Possible result: the SAP Mobile Server accepts the update, sends an import and a <code>ReplayResult</code> for the entity, and then refreshes the entity from the database.	IsNew=false IsCreated=false IsDirty=false IsDeleted=false IsUpdated= <b>false</b> Pending= <b>false</b> PendingChange='N' ReplayCounter= <b>33422977</b> replayPending= <b>0</b> ReplayFailure=0
Possible result: The SAP Mobile Server rejects the update, sends a <code>ReplayFailure</code> 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= <b>0</b> ReplayFailure= <b>33424981</b>

### **Pending State Pattern**

When a create, update, delete, or save operation is called on an entity in an application, 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 SAP Mobile Server. This method internally invokes the `SubmitPending` method on each of the pending records.

For message-based synchronization, the call to `SubmitPending` causes a JSON message to be sent to the SAP Mobile Server with the `Replay` method, containing the data for the rows to be created, updated, or deleted. The SAP Mobile Server processes the message and responds with a JSON message with the `ReplayResult` method (the SAP Mobile Server accepts the requested operation) or the `ReplayFailure` method (the server rejects the requested operation).

If the SAP Mobile 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 SAP Mobile 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 SAP Mobile Server. The SAP Mobile Server may optionally send a log record to the client indicating a successful operation.

If the SAP Mobile 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.

The call to `SubmitPending` creates a replay record in local database. When the `SMP101DB.Synchronize()` method is called, the replay records are uploaded to the SAP Mobile Server. The SAP Mobile 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 SAP Mobile 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 SAP Mobile Server. If the SAP Mobile 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 SAP Mobile Server may optionally send a log record to the client.

The `LogRecord` interface has the following getter methods to access information about the log record:

Method Name	Type	Description
<code>Component</code>	<code>string</code>	Name of the MBO for the row for which this log record was written.

Method Name	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: <ul style="list-style-type: none"> <li>• 200 indicates success.</li> <li>• 401 indicates that the client request had invalid credentials, or that authentication failed for some other reason.</li> <li>• 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).</li> <li>• 404 indicates that the client tried to access a nonexistent package or MBO.</li> <li>• 405 indicates that there is no valid license to check out for the client.</li> <li>• 500 indicates an unexpected (unspecified) server failure.</li> </ul>
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` 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.

### **Refresh Operation**

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

For example:

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

## **Generated Package Database APIs**

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

### **Client Database APIs**

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

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

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

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

```
public static Sybase.Persistence.LocalTransaction BeginTransaction()

Customer customer = Customer.FindByPrimaryKey(101);
// Use one transaction to save and submit pending
LocalTransaction tx = SMP101DB.BeginTransaction();
// modify customer information
customer.Save();
customer.SubmitPending();
tx.Commit();
```

## **Large Attribute APIs**

Use large string and binary attributes.

You can import large messages containing binary objects (BLOBs) to the client, send new or changed large objects to the server, and efficiently handle large attributes on the client.

The large attribute APIs allow clients to import large messages from the server or send a replay message without using excessive memory and possibly throwing exceptions. Clients can also access or modify a large attribute without reading the entire attribute into memory. In addition, clients can execute queries without having large attribute values automatically filled in the returned MBO lists or result sets.

### **BigBinary**

An object that allows access to a persistent binary value that may be too large to fit in available memory. A streaming API is provided to allow the value to be accessed in chunks.

#### **Close**

Closes the value stream.

Closes the value stream. Any buffered writes are automatically flushed. Throws a `StreamNotOpenException` if the stream is not open.

#### **Syntax**

```
public void Close()
```

#### **Examples**

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

```
Book book = Book.FindByPrimaryKey(bookID);  
Sybase.Persistence.BigBinary image = book.Cover;  
image.OpenForWrite(256);  
// ...  
image.Close();
```

#### **CopyFromFile**

Overwrites this `BigBinary` object with data from the specified file.

Any previous contents of the file will be discarded. Throws an `ObjectNotSavedException` if this `BigBinary` object is an attribute of an entity that has not yet been created in the database. Throws a `StreamNotClosedException` if the object is not closed.

**Syntax**

```
public void CopyFromFile(string filepath)
```

**Parameters**

- **filepath** – The file containing the data to be copied.

**CopyToFile**

Overwrites the specified file with the contents of this `BigBinary` object.

Any previous contents of the file are discarded. Throws an `ObjectNotSavedException` if this `BigBinary` object is an attribute of an entity that has not yet been created in the database. Throws a `StreamNotClosedException` if the object is not closed.

**Syntax**

```
public void CopyToFile(string filepath)
```

**Parameters**

- **filepath** – The file to be overwritten.

**Flush**

Flushes any buffered writes.

Flushes any buffered writes to the database. Throws a `StreamNotOpenException` if the stream is not open.

**Syntax**

```
public void Flush()
```

**OpenForRead**

Opens the value stream for reading.

Has no effect if the stream was already open for reading. If the stream was already open for writing, it is flushed before being reopened for reading. Throws an `ObjectNotSavedException` if this `BigBinary` object is an attribute of an entity that has not yet been created in the database. Throws an `ObjectNotFoundException` if this object is null.

**Syntax**

```
public void OpenForRead()
```

### Examples

- **Open for reading** – Opens a binary book image for reading.

```
Book book = Book.FindByPrimaryKey(bookID);  
Sybase.Persistence.BigBinary image = book.Cover;  
image.OpenForRead();
```

### OpenForWrite

Opens the value stream for writing.

Any previous contents of the value will be discarded. Throws an `ObjectNotSavedException` if this `BigBinary` object is an attribute of an entity that has not yet been created in the database.

### Syntax

```
public void OpenForWrite(long newLength)
```

### Parameters

- **newLength** – The new value length in bytes. Some platforms may allow this parameter to be specified as 0, with the actual length to be determined later, depending on the amount of data written to the stream. Other platforms require the total amount of data written to the stream to match the specified value.

### Examples

- **Open for writing** – Opens a binary book image for writing.

```
Book book = Book.FindByPrimaryKey(bookID);  
Sybase.Persistence.BigBinary image = book.Cover;  
image.OpenForWrite(256);
```

### Read

Reads a chunk of data from the stream.

Reads and returns the specified number of bytes, or fewer if the end of stream is reached. Throws a `StreamNotOpenException` if the stream is not open for reading.

### Syntax

```
public byte[] Read(long length)
```

### Parameters

- **length** – The maximum number of bytes to be read into the chunk.



## Returns

Read returns a chunk of binary data read from the stream, or a null value if the end of the stream has been reached.

## Examples

- **Read** – Reads in a binary book image.

```
Book book = Book.FindByPrimaryKey(bookID);
Sybase.Persistenc.BigBinary image = book.Cover;
image.OpenForRead();
byte[] binary = image.Read(bufferLength);
while (binary != null)
{
    binary = image.Read(bufferLength);
}
image.Close();
```

## ReadByte

Reads a single byte from the stream.

Throws a `StreamNotOpenException` if the stream is not open for reading.

## Syntax

```
public int ReadByte()
```

## Returns

ReadByte returns a byte of data read from the stream, or -1 if the end of the stream has been reached.

## Seek

Changes the stream position.

Throws a `StreamNotOpenException` if the stream is not open for reading.

## Syntax

```
public void Seek(long newPosition)
```

## Parameters

- **newPosition** – The new stream position in bytes. Zero represents the beginning of the value stream.

## Write

Writes a chunk of data to the stream.

Writes data to the stream, beginning at the current position. The stream may be buffered, so use `Flush` or `Close` to be certain that any buffered changes have been applied. Throws a

`StreamNotOpenException` if the stream is not open for writing. Throws a `WriteAppendOnlyException` if the platform only supports appending to the end of a value and the current stream position precedes the end of the value. Throws a `WriteOverLengthException` if the platform requires the length to be predetermined before writing and this write would exceed the predetermined length.

### **Syntax**

```
public void Write(byte[] data)
```

### **Parameters**

- **data** – The data chunk to be written to the stream.

### **Examples**

- **Write data** – Opens a binary book image for writing.

```
Book book = Book.FindByPrimaryKey(bookID);
Sybase.Persistence.BigBinary image = book.Cover;
image.OpenForWrite(256);
byte[] binary = new byte[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
image.Write(binary);
image.Close();
book.SubmitPending();
```

### **WriteByte**

Writes a single byte to the stream.

Writes a byte of data to the stream, beginning at the current position. The stream may be buffered, so use `Flush` or `Close` to be certain that any buffered changes have been applied. Throws a `StreamNotOpenException` if the stream is not open for writing. Throws a `WriteAppendOnlyException` if the platform only supports appending to the end of a value and the current stream position precedes the end of the value. Throws a `WriteOverLengthException` if the platform requires the length to be predetermined before writing and this write would exceed the predetermined length.

### **Syntax**

```
public void WriteByte(byte data)
```

### **Parameters**

- **data** – The byte value to be written to the stream.

### **BigString**

An object that allows access to a persistent string value that might be too large to fit in available memory. A streaming API is provided to allow the value to be accessed in chunks.

### Close

Closes the value stream.

Closes the value stream. Any buffered writes are automatically flushed. Throws a `StreamNotOpenException` if the stream is not open.

### Syntax

```
public void Close()
```

### Examples

- **Close the value stream** – Writes to the biography file, and closes the file.

```
Author author = Author.FindByPrimaryKey(authorID);
BigString text = author.Biography;
text.OpenForWrite(256);
text.Write("something");
text.Close();
```

### CopyFromFile

Overwrites this `BigString` object with data from the specified file.

Any previous contents of the value will be discarded. Throws an `ObjectNotSavedException` if this `BigString` object is an attribute of an entity that has not yet been created in the database. Throws a `StreamNotClosedException` if the object is not closed.

### Syntax

```
public void CopyFromFile(string filepath)
```

### Parameters

- **filepath** – The file containing the data to be copied.

### CopyToFile

Overwrites the specified file with the contents of this `BigString` object.

Any previous contents of the file are discarded. Throws an `ObjectNotSavedException` if this `BigString` object is an attribute of an entity that has not yet been created in the database. Throws a `StreamNotClosedException` if the object is not closed.

### Syntax

```
public void CopyToFile(string filepath)
```

### **Parameters**

- **filepath** – The file to be overwritten.

### **Flush**

Flushes any buffered writes.

Flushes any buffered writes to the database. Throws a `StreamNotOpenException` if the stream is not open.

### **Syntax**

```
public void Flush()
```

### **OpenForRead**

Opens the value stream for reading.

Has no effect if the stream was already open for reading. If the stream was already open for writing, it is flushed before being reopened for reading. Throws an `ObjectNotSavedException` if this `BigString` object is an attribute of an entity that has not yet been created in the database.

### **Syntax**

```
public void OpenForRead()
```

### **Examples**

- **Open for reading** – Opens the biography file for reading.

```
Author author = Author.FindByPrimaryKey(authorID);  
BigString text = author.Biography;  
text.OpenForRead();
```

### **OpenForWrite**

Opens the value stream for writing.

Any previous contents of the value will be discarded. Throws an `ObjectNotSavedException` if this `BigString` object is an attribute of an entity that has not yet been created in the database.

### **Syntax**

```
public void OpenForWrite(long newLength)
```

### **Parameters**

- **newLength** – The new value length in bytes. Some platforms may allow this parameter to be specified as 0, with the actual length to be determined later, depending on the amount of

data written to the stream. Other platforms require the total amount of data written to the stream to match the specified value.

### **Examples**

- **Open for writing** – Opens the biography file for writing.

```
Author author = Author.FindByPrimaryKey(authorID);
BigString text = author.Biography;
text.OpenForWrite(256);
```

### **Read**

Reads a chunk of data from the stream.

Reads and returns the specified number of characters, or fewer if the end of stream is reached. Throws a `StreamNotOpenException` if the stream is not open for reading.

### **Syntax**

```
public byte[] Read(long length)
```

### **Parameters**

- **length** – The maximum number of characters to be read into the chunk.

### **Returns**

Read returns a chunk of string data read from the stream, or a null value if the end of the stream has been reached.

### **Examples**

- **Read** – Reads in the biography file.

```
Author author = Author.FindByPrimaryKey(authorID);
BigString text = author.Biography;
text.OpenForRead();
int bufferLength = 1024;

string something = text.Read(bufferLength); //null if EOF
while (something != null)
{
    something = text.Read(bufferLength);
}
text.Close();
```

### **ReadChar**

Reads a single character from the stream.

Throws a `StreamNotOpenException` if the stream is not open for reading.

### **Syntax**

```
public int ReadChar()
```

### **Returns**

`ReadChar` returns a single character read from the stream, or -1 if the end of the stream has been reached.

### **Seek**

Changes the stream position.

Throws a `StreamNotOpenException` if the stream is not open for reading.

### **Syntax**

```
public void Seek(long newPosition)
```

### **Parameters**

- **newPosition** – The new stream position in characters. Zero represents the beginning of the value stream.

### **Write**

Writes a chunk of data to the stream.

Writes data to the stream, beginning at the current position. The stream may be buffered, so use `Flush` or `Close` to be certain that any buffered changes have been applied. Throws a `StreamNotOpenException` if the stream is not open for writing. Throws a `WriteAppendOnlyException` if the platform only supports appending to the end of a value and the current stream position precedes the end of the value. Throws a `WriteOverLengthException` if the platform requires the length to be predetermined before writing and this write would exceed the predetermined length.

### **Syntax**

```
public void Write(string data)
```

### **Parameters**

- **data** – The data chunk to be written to the stream.

### **Examples**

- **Write data** – Writes to the biography file, and closes the file.

```
Author author = Author.FindByPrimaryKey(authorID);  
BigString text = author.Biography;  
text.OpenForWrite(256);  
text.Write("something");
```

```
text.Close();
author.SubmitPending();
```

### WriteChar

Writes a single character to the stream.

Writes a character of data to the stream, beginning at the current position. The stream may be buffered, so use `Flush` or `Close` to be certain that any buffered changes have been applied. Throws a `StreamNotOpenException` if the stream is not open for writing. Throws a `WriteAppendOnlyException` if the platform only supports appending to the end of a value and the current stream position precedes the end of the value. Throws a `WriteOverLengthException` if the platform requires the length to be predetermined before writing and this write would exceed the predetermined length.

### Syntax

```
public void WriteChar(char data)
```

### Parameters

- **data** – The character value to be written to the stream.

## MetaData API

---

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

### MetaData API

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

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

You can generate metadata classes using the `-md` code generation option. You can also generate metadata classes by selecting the option **Generate metadata classes** in the code generation wizard in the mobile application project.

### DatabaseMetaData

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

Any entity for which "allow dynamic queries" is enabled generates attribute metadata. Depending on the options selected in the Eclipse IDE, metadata for attributes and operations may be generated for all classes and entities.

```
DatabaseMetaData dmd = SMP101DB.Metadata;  
foreach (String syncGroup in dmd.SynchronizationGroups)  
{  
    Console.WriteLine(syncGroup);  
}
```

### **ClassMetaData**

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

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

### **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);
```

## **Exceptions**

---

Reviewing exceptions allows you to identify where an error has occurred during application execution. These sections do not contain error codes contained in the exception classes. See the Developer Guide: Device Client Error Reference for detailed information about SAP Mobile Platform error codes.

### **Exception Handling**

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

To use the localization features in exception handling:



- Register an exception message service implementation through the `ServiceRegistry`.

### **Base Exceptions**

A base exception class is defined as the super class for all external exceptions. Specific exceptions always inherit from the base exception. To enable you, the Object API developer, to write a standard exception handler, all external exceptions have an error code and a single error message. Furthermore, the exception may contain another exception as the cause. See the *Developer Guide: Device Client Error Reference* for detailed information.

You can use the `GetLocalizedMessage (string localeName)` method to retrieve an error message for a specified locale.

See the *Object API Applications* section of the *Developer Guide: Device Client Error Reference* for information about possible error codes and the corresponding error messages.

### **Exception Message Service**

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

To resolve localized messages, implement the `IExceptionMessageService` interface.

```
class CustomExceptionMessageService : IExceptionMessageService
{
    public string GetMessage(int errorCode)
    {
        string msg = null;

        msg = "GetMessage(" + errorCode + ")";

        return msg;
    }

    public string GetMessage(int errorCode, string localeName)
    {
        string msg = null;

        msg = "GetMessage(" + errorCode + "," + localeName + ")";

        return msg;
    }
}
```

A default implementation, `ExceptionMessageServiceImpl` allows the default English resource to look up an error message using an error code. You can follow these steps to add other localized resources without implementing a custom message service.

1. Get the default file "SUPErrorMessage.en.txt" (included in the resources folder in the Mobile SDK for the Windows and Windows Mobile platform) and localize it to another language file such as "SUPErrorMessage.de.txt"

2. Use "resgen.exe" to generate its resource file like "SUPErrorMessage.de.resources":

```
>>resgen SUPErrorMessage.de.txt
```

3. Register the default implementation "ExceptionMessageServiceImpl" in the application code:

```
ServiceRegistry.GetInstance().RegisterService(typeof(IEExceptionMessageService),  
    ExceptionMessageServiceImpl.GetInstance());
```

4. Put the generated resource file into the same folder of the application execution file.

5. The application uses the localized error message automatically.

6. You can unregister the exception message service to cancel the use of the localized error message:

```
ServiceRegistry.GetInstance().UnregisterService<IEExceptionMessageService>(typeof(IEExceptionMessageService));
```

### **Service Registry**

You can register objects that implement the `IEExceptionMessageService` interface using the `ServiceRegistry` interface's `RegisterService` and `UnregisterService` methods.

```
ServiceRegistry.GetInstance().RegisterService(typeof(Sybase.Mobile.Framework.IExceptionMessageService), new  
CustomExceptionMessageService());  
...  
ServiceRegistry.GetInstance().UnregisterService<Sybase.Mobile.Framework.IExceptionMessageService>(typeof(Sybase.Mobile.Framework.IExceptionMessageService));
```

### **Example Code for Handling Exceptions**

An example of registering your interface.

```
// Register ExceptionMessageServiceImpl  
ServiceRegistry.GetInstance().RegisterService(typeof(Sybase.Mobile.Framework.IExceptionMessageService),  
ExceptionMessageServiceImpl.GetInstance());  
try  
{  
    // throw Sybase.Persistence.ObjectNotFoundException  
}  
catch (ObjectNotFoundException e)  
{  
    if (e.ErrorCode == ObjectNotFoundException.VALUE_IS_NULL)  
    {  
        string msg = e.Message;  
        msg = e.GetLocalizedMessage("fr");  
        msg = e.GetLocalizedMessage("es");  
        msg = e.GetLocalizedMessage("de");  
    }  
}
```

```
// Register CustomExceptionMessageService
ServiceRegistry.GetInstance().RegisterService(typeof(Sybase.Mobile.
Framework.IExceptionMessageService), new
CustomExceptionMessageService());
try
{
    // throw Sybase.Persistence.ObjectNotFoundException
}
catch (ObjectNotFoundException e)
{
    if (e.ErrorCode == ObjectNotFoundException.VALUE_IS_NULL)
    {
        string msg = e.Message;
        msg = e.GetLocalizedMessage("fr");
        msg = e.GetLocalizedMessage("es");
        msg = e.GetLocalizedMessage("de");
    }
}
finally
{
    // Unregister CustomExceptionMessageService

ServiceRegistry.GetInstance().UnregisterService<Sybase.Mobile.Frame
work.IExceptionMessageService>(typeof(Sybase.Mobile.Framework.IExce
ptionMessageService));
}
```

### **Server-Side Exceptions**

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

A server-side exception results in a stack trace in the server log, and a log record (LogRecordImpl) imported to the client with information on the problem.

### **Client-Side Exceptions**

Device applications are responsible for catching and handling exceptions thrown by the client object API.

---

**Note:** See *Callback Handlers*.

---

## **Exception Classes**

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

- **Sybase.Mobile.ApplicationRuntimeException** – thrown when a call to start the connection, register the application, or unregister the application cannot be completed due to an error.
- **Sybase.Mobile.ConnectionPropertyException** – thrown when a call to start the connection, register the application, or unregister the application cannot be completed due to an error in a connection property value or application identifier

- **Sybase.Mobile.ApplicationTimeoutException** – thrown when a call to start the connection, register the application, or unregister the application times out.
- **Sybase.Persistence.LoginRequiredException** – thrown when the client application does not login to the server.
- **Sybase.Persistence.NoSuchOperationException** – thrown when trying to access operation metadata that does not exist in class metadata.
- **Sybase.Persistence.NoSuchAttributeException** – thrown when trying to access an attribute that does not exist in class or entity metadata and thrown by a dynamic query method (such as ExecuteQuery) if the Query passed in selects for an attribute that does not exist in the MBO queried.
- **Sybase.Persistence.ObjectNotFoundException** – thrown when trying to load an MBO that is not inside the local database.
- **Sybase.Persistence.ObjectNotSavedException** – thrown when a BigBinary or BigString method is called that requires the object to already exist in the database.
- **Sybase.Persistence.PersistenceException** – thrown when trying to access the local database.
- **Sybase.Persistence.ProtocolException** – thrown when an exception occurs during protocol version mismatch.
- **Sybase.Persistence.StreamNotOpenException** – thrown when a BigBinary or BigString method is called that requires the object to be open.
- **Sybase.Persistence.StreamNotClosedException** – thrown when a BigBinary or BigString method is called that requires the object to not be open.
- **Sybase.Persistence.SynchronizeException** – thrown when an error occurs during synchronization.
- **Sybase.Persistence.SynchronizeRequiredException** – thrown when synchronization is needed.
- **Sybase.Persistence.WriteAppendOnlyException** – thrown if a BigBinary or BigString method is called that writes to the middle of a value where only appending is allowed by the underlying database.
- **Sybase.Persistence.WriteOverLengthException** – thrown if the platform requires the length to be predetermined before write and a BigBinary or BigString method is called that writes past the predetermined length.

## Error Codes

Codes for errors occurring during application execution.

### HTTP Error Codes

The SAP Mobile 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 SAP Mobile Platform internal error, or an unrecognized EIS error.

The EIS code and HTTP error code values are stored in log records (`ILogRecord.EisCode`, and `ILogRecord.Code`, respectively).

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

**Table 6. Recoverable Error Codes**

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

**Table 7. Unrecoverable Error Codes**

Error Code	Probable Cause	Manual Recovery Action
401	Backend EIS credentials wrong.	Change the connection information, or backend user password.
403	User authorization failed on the SAP Mobile Server due to role constraints (applicable only for MBS).	N/A
404	Resource (table/Web service/BA-PI) 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	SAP Mobile Platform internal error in modifying the CDB cache.	N/A

Error code 401 is not treated as a simple recoverable error. If the `SupThrowCredentialRequestOn401Error` context variable is set to true (the default), error code 401 throws a `CredentialRequestException`, which sends a credential request notification to the user's inbox. You can change this behavior by modifying the value of the `SupThrowCredentialRequestOn401Error` context variable in SAP Control Center. If `SupThrowCredentialRequestOn401Error` is set to false, error code 401 is treated as a normal recoverable exception.

**Mapping of EIS Codes to Logical HTTP Error Codes**

A list of SAP® error codes mapped to HTTP error codes. By default, SAP error codes that are not listed map to HTTP error code 500.

---

**Note:** These JCO error codes are not applicable for DOE-based applications.

---

**Table 8. Mapping of SAP Error Codes to HTTP Error Codes**

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

# Index

## A

- ActiveSync, installing and configuring 6, 22
- Afaria 13, 25
- Application APIs
  - retrieve connection properties 83
- application callback handlers 156
- application registration 36
- arbitrary find method 161, 162, 164, 167
- AttributeMetaData 192
- AttributeTest 162, 167
- AttributeTest condition 161
- authentication
  - online 39
- AVG 164

## B

- beginOnlineLogin 113
- beginSynchronize 115
- BigBinary 182
- BigString 186

## C

- callback handlers 40, 150
- CallbackHandler 61
- callbacks 39
- cascade operations 171
- certificates 7, 23, 101
- change notification 44
- ClassMetadata 192
- client database 181
- CloseConnection 102
- common APIs 178
- complex type 48
- composite relationships 171
- CompositeTest 167
- CompositeTest condition 161
- concatenate queries 166
- connection profile 36, 37
- ConnectionProfile 101
- ConnectionProperties 87
  - retrieve Farm ID 90
  - retrieve HTTP cookies 91

- retrieve HTTP credentials 92
- retrieve HTTP headers 91
- retrieve login certificate 88
- retrieve login credentials 88
- retrieve network protocol 88
- retrieve port number 89
- retrieve security configuration 89
- retrieve server name 89
- retrieve URL suffix 90

- COUNT 164
- create 50
- Create operation 169
- CreateDatabase 181

## D

- data synchronization protocol 3, 4
- data vault 134
  - change password 146
  - creating 132
  - deleting 135
  - exists 133
  - locked 141
  - locking 141
  - retrieve data names 135
  - retrieve string 143
  - retrieve value 145
  - set string 142
  - set value 144
  - unlocking 142
- database
  - client 181
- database connections
  - managing 102
- DatabaseMetaData 191
- DataVault 132
- DataVaultException 132
- debugging 61, 63
- default password 140
- delete 50
- Delete operation 170
- DeleteDatabase 181
- deploying
  - configuring ActiveSync for 6, 22
- device database 43
- Disaster recovery 53

## Index

documentation roadmap 4  
dynamic query 47

## E

EIS error codes 196, 198  
encryption key 130  
entity states 173, 175  
error codes  
    EIS 196, 198  
    HTTP 196, 198  
    mapping of SAP error codes 198  
    non-recoverable 196  
    recoverable 196  
EXCEPT 166  
exceptions  
    client-side 195  
    server-side 195

## F

filtering results 165  
FROM clause 166

## G

generated code contents 12, 24  
generated code, location 12, 24  
group by 165

## H

High availability 53  
HTTP error codes 196, 198

## I

installing  
    Microsoft ActiveSync 6, 22  
    synchronization software 6, 22  
INTERSECT 166

## J

Javadocs, opening 75  
JMSBridge 61

## L

libraries 19, 30

listeners 39  
localization 69–71  
LogRecord API 123

## M

MAX 164  
maxDbConnections 102  
MBO 45, 47, 48, 50  
MBOLogger 61  
messaging protocol 3, 4  
MetaData API 191  
Microsoft ActiveSync, installing and configuring 6,  
    22  
MIN 164  
mobile middleware services 4  
multilevel insert 171

## N

NoSuchAttributeException 195  
NoSuchOperationException 195

## O

Object API code  
    location of generated 12, 24  
Object Manager API 191  
object query 47, 160  
ObjectNotFoundException 195  
OnImportSuccess 112  
OnlineLogin 106  
OpenConnection 102  
other operation 171

## P

paging data 161, 164  
password policy 140  
    set 136  
Pending operation 172  
pending state 50  
personalization keys 110  
    types 110

## Q

Query class 161



Query object 162, 164, 167  
 QueryResultSet 168

## R

Refresh operation 181  
 relationships 169  
 replay 40  
 resumable synchronization 119  
 resumeSubscription 116

## S

Save operation 171  
 SelectItem 166  
 setting the database file location on the device 103  
 setting the databaseFile location 103  
 signing 73  
 simultaneous synchronization 112  
 Skip 167  
 Skip condition 161  
 SortCriteria 164, 167  
 SortCriteria condition 161  
 status methods 173, 175  
 subqueries 166  
 subscribe 113  
 subscribe() 112  
 SUM 164  
 SUPBridge 61  
 suspendSubscription 115  
 synchronization 43
 

- MBO package 112
  - of MBOs 112
  - replication-based 112
  - simultaneous 112

 synchronization group 44  
 synchronization parameters 45

synchronization profile 37  
 synchronization software
 

- installing 6, 22

 SynchronizationProfile 104, 105  
 SynchronizeException 195

## T

TestCriteria 167  
 TestCriteria condition 161

## U

UltraLite 43  
 UNION 166  
 UNION\_ALL 166  
 unsubscribe 114  
 update 50  
 Update operation 170

## V

VacuumDatabase 117  
 value
 

- deleting 145

 Visual Studio
 

- required DLLs 16, 28

## W

Windows Mobile Device Center 6, 22

## X

X.509 certificates 7, 23

