SYBASE®

An **SAP**® Company

Developer Guide: OData SDK

Sybase Unwired Platform 2.1
ESD #2

Contents

CHAPTER 1   **OData SDK Overview**

The OData SDK is for building native mobile applications. It consists of a collection of runtime libraries and classes.

The OData SDK supports Android, BlackBerry and iOS platforms and it is based on the native device SDKs of the platforms. There is an implementation for each platform. Native applications installed on the devices allow the client application to leverage the support provided by the given platform, for example:

- Adapt to each device's form factor (for example, automatic layout)
- Exploit different input methods (for example, touch screen, keyboard or trackball)
- Cache data in native device data stores for better performance
- Tightly integrate with the features of the device

The general description of the SDK components follows. For detailed platform specific descriptions, see the respective chapters on Android, BlackBerry and iOS.

OData for SAP Products

OData stands for "Open Data Protocol" and is a resource-based web protocol for querying and updating data. It is released by Microsoft under the Open Specification Promise to allow anyone to freely interoperate with OData implementations. OData defines operations on resources using HTTP verbs (GET, PUT, POST, and DELETE), and it identifies those resources using a standard URI syntax. Data is transferred over HTTP using the Atom or JSON format.

OData for SAP® Products provide SAP Extensions to the OData protocol that enable users to build user interfaces for accessing the data published via OData. The interfaces require human-readable, language-dependent labels for all properties and free-text search within collections of similar entities and across (OpenSearch).

Applications running on mobile devices also require semantic annotations to tell the client which of the OData properties contain a phone number, a part of a name or address, or something related to a calendar event, thus seamlessly integrating with the contacts, calendar, and telephony of the mobile device. The OData standard's metadata document contains information about the model. It will define what information is searchable, which properties may be used in filter expressions, and which properties of an entity will always be managed by the server.

For the sake of simplicity, "OData for SAP" is abbreviated to "OData" throughout this Guide.

OData SDK Components — General Description

The different components of the OData SDK are implemented as static runtime libraries and each component can be used independently.

The following components are included in the OData SDK. See the detailed platform specific descriptions in the respective sections.

OData Parser

Parses and generates valid OData Protocol messages to/from native objects. It eliminates the need for mobile developers to work with the low-level details of the OData protocol directly. Functionalities supported by this component include:

- Parsing OData XML structures to native OData objects
- Validating OData XML during parsing by checking the existence of mandatory fields and structures
- Providing easy access to all OData fields and structures via the objects resulting from the parsing
- Building OData XML structures from native OData objects

Cache Management

The runtime cache is responsible for storing and accessing OData related objects in the memory of the device for quick and easy access. Functionalities supported by this component include:

- Storing/accessing OData objects in the memory (both metadata and application data)
- Searching for OData entries in memory using their searchable fields
- Managing the size of the cache

Persistence

Implements a convenient and secure storage of data on the device. Mobile applications can access the locally stored data even when network connection is unavailable. Functionalities supported by this component include:

- Storing objects and raw data on the physical storage of the device
- Easy and quick access of the stored objects and raw data
- Data encryption for sensitive data

Supportability

Implements standard SAP logging, tracing and error handling to enable end-to-end supportability from client to back-end. Functionalities supported by this component include:

- Common exception and error handling

- Event logging
- Tracing (SAP Passport)

Connectivity

This network layer handles all network related tasks, hides the complexity of the network communication, and provides an easy to use API to the applications. For crossing a company firewall for enterprise use cases, you need to use SUP. Therefore, the connectivity component in the OData SDK offers a connection to SUP by default. For development and demo purposes, the SDK also provides a possibility to use HTTP or HTTPS. Functionalities supported by this component include:

- Synchronous and asynchronous HTTP request handling
- Basic authentication (user/password)
- Timeout handling
- Compressed payload handling
- Request types as supported by OData Protocol
- Connection pools for optimal performance

Documentation Roadmap for Unwired Platform

Sybase® Unwired 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. *Fundamentals* is available on the Sybase Product Documentation Web site.

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

CHAPTER 2   **Developing iOS Applications**

Provides information about using advanced Sybase® Unwired Platform features to create applications for Apple iOS devices. The audience is advanced developers who are familiar working with APIs, but who may be new to Sybase Unwired Platform.

Describes requirements for developing a device application for the platform. Also included are task flows for the development options, procedures for setting up the development environment and API references.

1. *Setting Up the Development Environment*

   Import the associated iOS libraries into the iOS development environment.

2. *Developing Applications in the Xcode IDE*

   After you import mobile applications and associated libraries into the iOS development environment, use the iOS API references to create or customize your device applications.

3. *OData SDK Components and APIs*

   The iOS OData SDK provides the means to easily build an app which relies on the OData protocol and the additions made by SAP.

4. *Deploying Applications to Devices*

   Complete steps required to deploy mobile applications to devices.

Setting Up the Development Environment

Import the associated iOS libraries into the iOS development environment.

Note: For more information on Xcode, refer to the Apple Developer Connection: *http:// developer.apple.com/tools/Xcode/.*

1. Start Xcode 4.2 and select **Create a new Xcode project**.

2. Under **iOS**, select **Applications**.

3. In the right pane, select **Empty Application** as the project template and click **Next**.

4. Enter `<ProjectName>` as the **Product Name**, `<CorpID>` as the **Company Identifier**, select **Universal** as the **Device Family** product, then click **Next**.

5. Select a location to save the project and click **Create** to open it.

   Xcode creates a folder, `<ProjectName>`, to contain the project file, `<ProjectName>.xcodeproj` and another `<ProjectName>` folder, which contains a number of automatically generated files.

6. Select the **Build Settings** tab and scroll to the **Architectures** section.

7. Set **Base SDK** for **All** configurations to `iOS 5.0`.

8. Select the Valid architecture as `armv6 armv7`, Architectures as `Optimized (armv7)` and the Targeted device family as `iPhone/iPad`. This ensures that the build of the application can run on either iPhone or iPad.

> **Note:** When you migrate an existing project from an older version of Xcode to Xcode 4.2, you may see a build error: `No architectures to compile for (ARCHS=i386, VALID_ARCHS=armv6,armv7)`. You can resolve this Xcode 4 issue by manually editing "Valid Architectures" under Targets, to add `i386`.

9. In the **Deployment** section, set the iOS Deployment Target to `iOS 4.3` or `iOS 4.2` or `iOS 5.0`, as appropriate for the device version where you will deploy. Earlier SDKs and deployment targets are not supported.
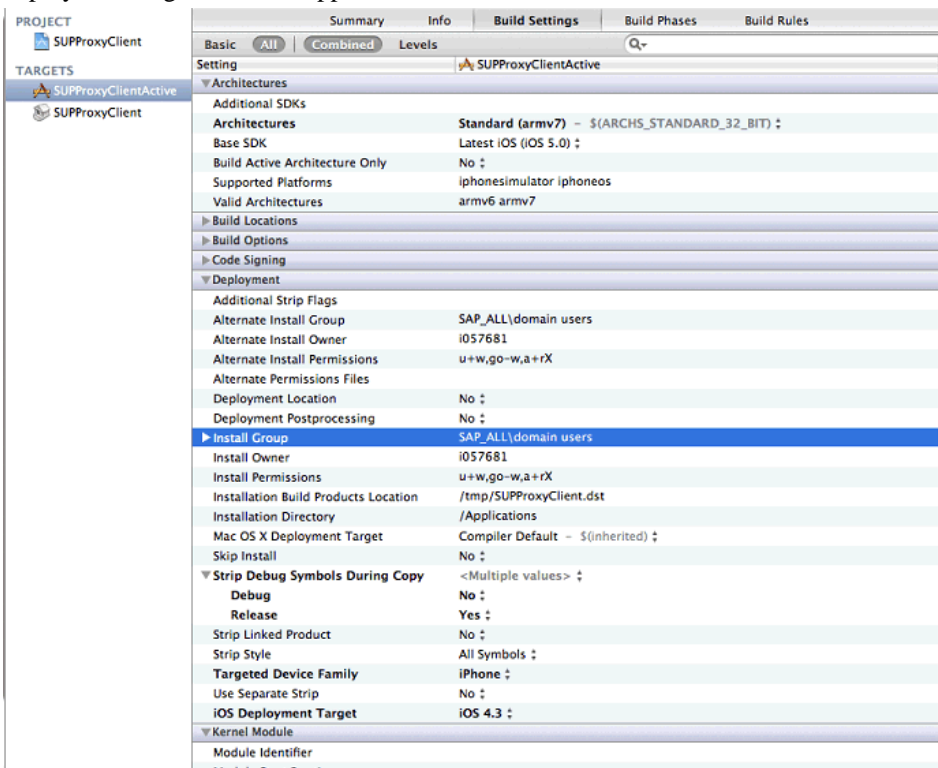
10. Connect to the Microsoft Windows machine where Mobile SDK is installed:
    a) From the Apple Finder menu, select **Go > Connect to Server**.
    b) Enter the machine name where the SDK is installed or IP address of the server, for example, `smb://<machine DNS name>` or `smb://<IP Address>`.
       You see the shared directory.

11. Navigate to the `<UnwiredPlatform_InstallDir>\MobileSDK\OData\iOS\` directory in the Unwired Platform installation directory, and copy the `includes` and

libraries folders to the `<ProjectName>/<ProjectName>` directory on your Mac.

12. Right-click the `<ProjectName>` folder under the project, select **Add Files to "<ProjectName>"**, navigate to the `<ProjectName/ProjectName>/libraries/Debug-universal` directory, select all the libraries, unselect **Copy items into destination group's folder (if needed)**, and click **Add**.

The libraries are added to the project in the Project Navigator.

13. Click the project root and then, in the middle pane, click the **<ProjectName>** project.

   a) In the right pane click the **Build Settings** tab, then scroll down to the **Search Paths** section.

   b) Enter the location of your `includes` folder (`"$SRCROOT/<ProjectName>/includes/public/**"`) in the **Header Search Paths** field.

   $SRCROOT is a macro that expands to the directory where the Xcode project file resides.

14. Set the Other Linker Flags to `-ObjC -all_load` for both the release and for the debug configuration. It is important that the casing of -ObjC is correct (upper case 'O' and upper case 'C'). Objective-C only generates one symbol per class. You must force the linker to load the members of the class. You can do this with the help of the -ObjC flag. You must also force inclusion of all your objects from your static library by adding the linker flag `-all_load`.

15. Add the following frameworks from the SDK to your project by clicking on the active target, and selecting **Build Phase** > **Link Binary With Libraries**. Click on the + button and select the following binaries from the list:

   - CoreFoundation.framework
   - QuartzCore.framework
   - Security.framework
   - libicucore.A.dylib
   - libstdc++.dylib
   - libz.1.2.5.dylib
   - CFNetwork.framework
   - MobileCoreServices.framework
   - SystemConfiguration.framework
   - MessageUI.framework

16. Select **Product > Clean** and then **Product > Build** to test the initial set up of the project. If you have correctly followed this procedure, you should receive a **Build Succeeded** message.

Developing Applications in the Xcode IDE

After you import mobile applications and associated libraries into the iOS development environment, use the iOS API references to create or customize your device applications.

For a comprehensive list of API references, extract the contents from the following zip files:

- `<UnwiredPlatform_InstallDir>\MobileSDK\OData\iOS\docs\SUPProxyClient-API-Docs.zip`
- `<UnwiredPlatform_InstallDir>\MobileSDK\OData\iOS\docs\SDMConnectivity-API-Docs.zip`
- `<UnwiredPlatform_InstallDir>\MobileSDK\OData\iOS\docs\SDMParser-API-Docs.zip`
- `<UnwiredPlatform_InstallDir>\MobileSDK\OData\iOS\docs\SDMSupportability-API-Docs.zip`

See also
- *OData SDK Components and APIs* on page 24

Initializing an Application

Initialize an application.

Syntax

```
+(LiteSUPUserManager *) getInstance: (NSString *) appID
```

Parameters

- **appID** – Name of the registered application.

Examples

- **Initialize an application** –

```
LiteSUPUserManager* userManager = [LiteSUPUserManager
getInstance:@"APP_ID"];
```

Setting Connection Profile

Set the server details.

Syntax

```
-(void) setConnectionProfile: (NSString *) supServerHost
withSupPort: (NSInteger) supPort
withServerFarmID: (NSString *) serverFarmID
```

Parameters

- **supServerHost –** Corresponds to the IP Address used to identify the SUP Server.
- **supPort –** Corresponds to the SUP port.
- **serverFarmID –** Corresponds to the server farm ID.

Examples

- **Set the connection profile –**

```
[userManager setConnectionProfile: @"10.53.222.37" withSupPort:
5001 withServerFarmID:@"0"];
```

Assigning and Implementing Delegates

Assign and implement delegates for synchronous and asynchronous user registration.

Examples

- **Register a Delegate –**

```
LiteSUPUserManager* userManager = [LiteSUPUserManager
getInstance:@"APP_ID"];
          [userManager setDelegate:self];
          [userManager
setDidFailToRegisterUser:@selector(regFailed:)];
          [userManager
setDidSuccessfulUserRegistration:@selector(regSuccess:)];
          [userManager setConnectionProfile:@"10.53.222.37"
withSupPort:5001 withServerFarmID:@"0"];
```

- **Implementation of the Delegate –**

```
-(void)regFailed:(NSError*)error{
                UIAlertView* alert = [[UIAlertView alloc]
initWithTitle:@"Error" message:[error localizedDescription]
delegate:self cancelButtonTitle:@"OK" otherButtonTitles:nil];
                [alert show];
                [alert release];
}

-(void)regSuccess:(id)sender{
                UIAlertView* alert = [[UIAlertView alloc]
initWithTitle:@"Success" message:@"User Registration Successful"
delegate:self cancelButtonTitle:@"OK" otherButtonTitles:nil];
                [alert show];
```

```
                               [alert release];
```

Manually Registering an Application

Manually register an application synchronously or ansynchronously by using the user name
and activation code of the application registered through the Sybase Control Center.

Syntax

```
-(void) registerUser : (NSString *) username
withActivationCode: (NSString *) activationCode
```

```
-(void) registerUserAsynchronousWithUserName: (NSString *) username
activationCode: (NSSting *) activationCode
```

Parameters

- **username** – User name of the user to be registered.
- **activationCode** – Activation code of the user created in SCC.

Examples

- **Synchronous Registration of an application without a delegate –**

```
LiteSUPUserManager* userManager = [LiteSUPUserManager
getInstance:@"APP_ID"];
          [userManager setConnectionProfile:@"10.53.222.37"
withSupPort:5001 withServerFarmID:@"0"];
    @try {
                    // Manual On-boarding
                  [userManager registerUser:@"manualuser"
withActivationCode:@"123"];
}
 @catch (NSException * e) {
                  // Exception Handling
}
```

- **Asynchronous Registration of an application using a delegate –** The user registration
  runs as a background process and has to have a delegate registered to provide success/
  failure notifications.

```
LiteSUPUserManager* userManager = [LiteSUPUserManager
getInstance:@"APP_ID"];
          [userManager setDelegate:self];
          [userManager
setDidFailToRegisterUser:@selector(regFailed:)];
          [userManager
setDidSuccessfulUserRegistration:@selector(regSuccess:)];
          [userManager setConnectionProfile:@"10.53.222.37"
withSupPort:5001 withServerFarmID:@"0"];
 [userManager registerUserAsynchronousWithUserName:@"manualuser"
activationCode:@"123"];
```

Automatically Registering an Application using SSO2 Cookie

Registering an application automatically using an SSO2 Token Cookie. The token is fetched from a ticket issuing system and verified by the server.

Syntax

```
-(void) registerUser: (NSString *) username
withSecurityConfig: (NSString *) securityConfig
withPassword: (NSString *) password
withVaultPassword: (NSString *) vaultPassword
```

Parameters

- **username** – User name of the ticket issuing system.
- **securityConfig** – Security configuration for the registered application provided by the administrator in the Sybase Control Center
- **password** – Password used to authenticate the user.
- **vaultPassword** – Password of the vault data store.

Examples

- **Automatically registering an application using SSO2 Cookie –**
```
@try {
[userManager registerUser:@"user"
withSecurityConfig:@"SSO2Cookie" with-Password:@"password";
withVaultPassword:@"vaultpassword"];
@catch (NSException* e) {
    // Exception Handling
}
```

Automatically Registering an Application using HTTP Authentication Provider

Registering an application automatically using HTTP Authentication Provider

Syntax

```
-(void) registerUser: (NSString *) username
withSecurityConfig: (NSString *) securityConfig
withPassword: (NSString *) password
withVaultPassword: (NSString *) vaultPassword
```

Parameters

- **username** – Valid user name.

- **securityConfig** – Security configuration of the registered application provided by the administrator in the Sybase Control Center.
- **password** – Password used to authenticate the user.
- **vaultPassword** – Password required to unlock the data vault .

Examples

- **Registering an application using HTTP Authentication Provider –**

```
@try{
[userManager registerUser:@"user" withSecurityConfig:@"sec-
config" withPassword:@"password"
withVaultPassword:@"vaultpassword"];
@catch (NSException* e) {}
```

Automatically Registering an Application using X.509 Certificates

Registering a user automatically using an X.509 Certificate. This certificate is fetched from a certificate authority and verified by the server.

Syntax

```
-(void) registerUser: (NSString *) username
withSecurityConfig: (NSString *) securityConfig
withPassword: (NSString *) password
withVaultPassword: (NSString *) vaultPassword
```

Parameters

- **username** – User name of the user to be registered.
- **securityConfig** – Security configuration for the registered application provided by the administrator in the Sybase Control Center
- **password** – Contains the Base64 encoded string of the certificate library.
- **vaultPassword** – Password of the vault data store.

Examples

- **Automatically registering an application using X.509 certificates –**

```
@try {
LiteSUPCertificateStore* store = [LiteSUPCertificateStore
getInstance];
NSString* base64cert = [store
getSignedCertificateFromFile:<filepath>
withCertificatePassword:<password>];
[userManager registerUser:<username>
withSecurityConfig:<security_config> withPassword:base64cert];//
withVaultPassword:<vaultpassword>];
}
@catch (NSException * exception) {
```

```
     NSLog(@"%@", [exception reason]);
}
```

Enabling Online Push

To consume push messages, the application developer implements a delegate method from the SDMSUPPushDelegate.h header file.

Syntax

```
- (void) pushNotificationReceived: (NSDictionary *) data
```

Examples

- **Online Push Implementation –**

  1. Import the header file SDMSUPPushDelegate.h in the header file of the class where you implement the push delegate. It is recommended to implement this in the App delegate.
  2. The header file should implement the protocol. This is done by including the <SDMSUPPushDelegate> key to the super class from which the current class inherits the header file.

     ```
     @interface SUPProxyClientAppDelegate :
     NSObject<UIApplicationDelegate, SDMSUPPushDelegate>
     ```
  3. Import the SUPUtilities.h file in the .m file of the class in which you implement the delegate.
  4. In any of the startup methods add the following code snippet:

     ```
     [SUPUtilities setDelegate:self];
     ```
  5. Implement the delegate method which receives the data in the *data* parameter.

     ```
     - (void) pushNotificationReceived:(NSDictionary*)data{
     NSLog(@"%@",[data objectForKey:@"Data]);
     }
     ```

Storing the Application Credentials Securely

Post user registration, if you want the user credentials to be managed by SDK, you can provide a data vault password to securely store the data.

Syntax

```
- (void) setAppCredentials: (NSString *) username
withSecurityConfig: (NSString *) securityConfig
withPassword: (NSString *) password
withVaultPassword: (NSString *) vaultPassword
```

Parameters

- **username** – Valid user name to be stored.

- **securityConfig** – Security configuration of the registered application to be stored.
- **password** – If using certificates, this corresponds to the Base64 encoded string of the certificate library. If using SSO2 cookie, this corresponds to the passowrd of the ticket issuing system.
- **vaultPassword** – Password of the secure store provided by SDK.

Examples

- **Using Data Vault Password –**

```
@try{
                [userManager setAppCredentials:<user name>
withSecurityConfig:<security config> withPassword:<password>
withVaultPassword:<vault password>];
}
@catch (NSException* e) {
                // Exception Handling
}
```

Getting Application End-point

Retrieve the application end-point that corresponds to the gateway service document.

Syntax

```
+(NSString*) getApplicationEndPoint
```

Examples

- **Retrieving application end-point –**

```
NSLog(@"%@", [LiteSUPAppSettings getApplicationEndPoint]);
```

Getting Push End-point

Retrieve the push end-point that corresponds to the delivery address that the application uses in the subscription request for notifications.

Syntax

```
+(NSString*) getPushEndPoint
```

Examples

- **Retrieve the push end-point –**

```
NSLog(@"%@", [LiteSUPAppSettings getPushEndPoint]);
```

Getting Server Details

Retrieve the server name provisioned in the client repository.

Syntax

```
+ (NSString *) getServer
```

Returns

Returns the server name as a string.

Examples

- **Retrieve the server details –**
  ```
  LiteSUPAppSettings getServer();
  ```

Getting Port Number

Retrieve the port number provisioned in the client repository.

Syntax

```
+ (int) getPortNumber
```

Returns

Returns the port number as an integer.

Examples

- **Retrieve the port number –**
  ```
  [LiteSUPAppSettings getPortNumber];
  ```

Getting FarmID

Retrieve the farm ID provisioned in the client repository.

Syntax

```
+ (NSString*) getFarmId
```

Returns

Returns the farm ID as a string.

Examples

- **Retrieve the Farm ID –**
  ```
  [LiteSUPAppSettings getFarmId];
  ```

Checking the Provisioning Status of the Public Key

Check if the public key is provisioned on the client.

Syntax

```
+ (BOOL) isSUPKeyProvisioned
```

Returns

Returns the result as a BOOL.

Examples

- **Check the provisioning status of the public key –**
  ```
  [LiteSUPAppSettings isSUPKeyProvisioned];
  ```

Deleting Users

Deletes a registered user. If the user credentials are managed by SDK, this API deletes the user credentials from the vault and deletes the user from the server.

Syntax

```
-(void) deleteUser
```

Examples

- **Deleting Users –**
  ```
  LiteSUPUserManager* userManager = [LiteSUPUserManager
  getInstance:<application_id>];
  [userManager deleteUser];
  ```

Getting Application Seed Data from Afaria

Get the application seed data from Afaria®.

Syntax

```
- (NSMutableDictionary *) getSettingsFromAfariaWithUrl: (NSURL *)
configurationUrl
UrlScheme: (NSString *) urlScheme
```

Parameters

- **configurationUrl –** URL passed by the Afaria client.

- **urlScheme** – URL scheme of the calling application. Afaria library can use this to pass it to the Afaria client.

Returns

Returns a NSMutableDictionary containing properties that are read from the seed file.

Examples

- **Get Settings From Afaria –**

```
- (BOOL)application:(UIApplication *)application openURL:(NSURL
*)url sourceApplication:(NSString *)sourceApplication annotation:
(id)annotation{
LiteSUPUserManager* userManager = [LiteSUPUserManager
getInstance:<application_id>];
[userManager getSettingsFromAfariaWithUrl:url
UrlScheme:<url_scheme>];
}
```

Provisioning Connection Settings from Afaria

Connection Settings for an application can be provisioned using the Afaria client that is installed on the mobile device.

Syntax

```
- (NSInteger) setConnectionProfileFromAfaria: (NSURL *) url
appUrlScheme: (NSString *) urlScheme
```

Parameters

- **url** – URL generated by Afaria and is specific to the URL scheme.
- **urlScheme** – URL scheme of the application. This is used by the Afaria library to communicate with the Afaria client.

Examples

- **Set Connection Profile Settings –**

```
- (BOOL)application:(UIApplication *)application openURL:(NSURL
*)url sourceApplication:(NSString *)sourceApplication annotation:
(id)annotation{
LiteSUPUserManager* userManager = [LiteSUPUserManager
getInstance:<application_id>];
[userManager setConnectionProfileFromAfaria:url
appUrlScheme:<url_scheme>];
}
```

Provisioning Certificates Using URLScheme with Afaria

Returns the certificate as a base64 encoded string if the URL Scheme is registered with Afaria.

Syntax

```
- (NSString *) getSignedCertificateFromAfariaForURLScheme: (NSString
*) urlScheme
withUsername: (NSString *) username
withPassword: (NSString *) password
```

Parameters

*   **urlScheme –** URL scheme of the calling application. Afaria library can use this to pass it to the Afaria client.
*   **username –** Common name used to generate the CSR.
*   **password –** Certificate password.

Returns

Returns a certificate as a base64 encoded string

The application has to implement the standard receiver delegate

```
- (BOOL)application:(UIApplication *)application openURL:(NSURL
*)url sourceApplication:(NSString *)sourceApplication annotation:
(id)annotation
```

to handle the URL received from the Afaria client if the Afaria settings are missing.

Examples

*   **Certificate Provisioning using URLScheme –**

```
@try {
        LiteSUPCertificateStore* store = [LiteSUPCertificateStore
getInstance];
        NSString* certBase64 = [store
getSignedCertificateFromAfariaForURLScheme:@"SCHEME"
withUsername:@"UserName" withPassword:@"password"];
        NSLog(@"%@", certBase64);
    }
    @catch (NSException *exception) {
        NSLog(@"%@", [exception reason]);
    }
```

Provisioning Certificates Using URL with Afaria

Returns the certificate as a base64 encoded string.

Syntax

```
- (NSString *) getSignedCertificateFromAfariaForURL: (NSSTring *)
url
withUsername: (NSString *) username
withPassword: (NSString *) password
```

Parameters

- **url** – URL passed by the Afaria client.
- **username** – Common name used to generate the CSR.
- **password** – Certificate Authority password that should associate with CSR.

Returns

Returns the certificate as a base64 encoded string.

Examples

- **Provisioning Certificates Using URL –**

```
if ([[url absoluteString] length] != 0 ) {
        @try {
            LiteSUPCertificateStore* store =
[LiteSUPCertificateStore getInstance];
            NSString* certBase64 = [store
getSignedCertificateFromAfariaForURL:[url absoluteString]
withUsername:@"Username" withPassword:@"Password"];
            NSLog(@"%@", certBase64);
        }
        @catch (NSException *exception) {NSLog(@"%@", [exception
reason]);
        }
    }
```

Usage

This API has to mandatorily be called in the

```
- (BOOL)application:(UIApplication *)application openURL:(NSURL
*)url sourceApplication:(NSString *)sourceApplication annotation:
(id)annotation
```

method of your application delegate. The API is called after the Afaria client generates a URL and forwards it to the application via the delegate. This is required only if the above call is the first call to Afaria in the application.

Clearing the Server Verification Key

For a device to switch connection between SUP servers, this API is invoked before registering a new user. This ensures that the server public keys are removed from the SUP client SDK which enables connectivity to the new SUP Server.

Syntax

```
+ (void) clearServerVerificationKey
```

Examples

- **Clear the server verification key –**

  ```
  [LiteSUPUserManager clearServerVerificationKey]
  ```

Data Vault API References

The data vault is a secure storage area provided by the SUP 2.1 SDK client libraries to store sensitive data such as usernames, passwords, authentication certificates within the application. Access to the data vault is protected by two levels of passwords and unique salts.

Creating a Vault

Creates an instance of a vault with a set of attributes.

Syntax

```
+ (LiteSUPDataVault *) createVault: (NSString *) dataVaultID
withPassword: (NSString *) password
withSalt: (NSString *) salt
```

Parameters

- **dataVaultID –** The vault name.
- **password –** The vault password
- **salt –** The salt password

Returns

If successful, creates an instance of LiteSUPDataVault.

Examples

- **Create a Vault –**

  ```
  LiteSUPDataVault* dataVault = [LiteSUPDataVault
  createVault:<vault_name> withPassword:<vault_pwd>
  withSalt:<vault_salt>];
  ```

Opening an Existing Vault

This API is used to check if a vault exists. If the vault does not exist or has been deleted, this method throws an exception.

Syntax

```
+ (bool) vaultExists: (NSString *) dataVaultID
```

Parameters

- **dataVaultID –** The vault name.

Returns

If successful, returns 'true'.

Examples

- **Open an Existing Vault –**
  ```
  LiteSUPDataVault* dataVault = [LiteSUPDataVault
  getVault:<vault_name>];
  ```

Deleting a Vault

Delete the storage for this instance from the persistent storage. Once a vault is deleted, all current instance references become invalid.

Syntax

```
+ (void) deleteVault: (NSString *) dataVaultID
```

Parameters

- **dataVaultID –** The vault name.

Examples

- **Delete a Vault –**
  ```
  [LiteSUPDataVault deleteVault:<vault_name>];
  ```

Locking a Vault

Lock a vault to avoid it from being used. If the vault is locked, this API will have no effect.

Syntax

```
- (void) lock
```

Examples

- **Lock a Vault –**
  ```
  LiteSUPDataVault* vault = [LiteSUPDataVault
  getVault:<vault_name>];
   [vault lock];
  ```

Unlocking a Vault

Unlock a vault for use by an application.

Syntax

```
- (void) unlock: (NSString *) password
withSalt: (NSString *) salt
```

Parameters

- **password –** The vault password.
- **salt –** The vault's salt password.

Examples

- **Unlock a Vault –**

```
LiteSUPDataVault* vault = [LiteSUPDataVault
getVault:<vault_name>];
[vault unlock:<vault_pwd> withSalt:<vault_salt>];
```

Setting a Binary Value

Store a value in the vault. To remove a value, provide 'null' as the second parameter.

Syntax

```
- (void) setValue: (NSString *) dataKey
with Value: (NSData *) byteValue
```

Parameters

- **dataKey –** The key used to store the data.
- **byteValue –** The value to be stored in the vault.

Examples

- **Set a Binary Value –**

```
LiteSUPDataVault* vault = [LiteSUPDataVault
getVault:<vault_name>];
[vault setValue:<key> withValue:<data_value>];
```

Retrieving a Binary Value

Retrieve a value set from the vault.

Syntax

```
- (NSData *) getValue: (NSString *) dataKey
```

Parameters

- **dataKey –** The key in which the data is stored.

Returns

If successful, this returns NSData.

Examples

- **Retrieve a Binary Value –**
```
LiteSUPDataVault* vault = [LiteSUPDataVault
getVault:<vault_name>];
NSData* dataValue = [vault getValue:<key>];
```

Setting the Retry Limit Value for a Vault

Set the maximum number of consecutive failed attempts to unlock the vault.

Syntax
```
-(void) setRetryLimit: (int32_t) numOfAttempts
```

Parameters

- **numOfAttempts –** Maximum failed attempts that is permitted to unlock the vault.

Examples

- **Set Retry Limit Value –**
```
LiteSUPDataVault* vault = [LiteSUPDataVault
getVault:<vault_name>];
[vault setRetryLimit:<retry_number>];
```

Setting the Lock Timeout Value for a Vault

Set the time until which the vault remains in an unlocked state. Once this time is lapsed, the vault reverts to the locked state.

Syntax
```
-(void) setLockTimeout: (int32_t) numberOfSeconds
```

Parameters

- **numberOfSeconds –** Time in seconds for which the vault is unlocked.

Examples

- **Set the Lock Timeout –**

```
LiteSUPDataVault* vault = [LiteSUPDataVault
getVault:<vault_name>];
[vault setLockTimeout:<num_of_seconds>];
```

OData SDK Components and APIs

The iOS OData SDK provides the means to easily build an app which relies on the OData protocol and the additions made by SAP.

Prerequisites for Developing iOS Apps

- An Intel based Mac
- Official iOS SDK and the development environment, which registered developers can download for free. Use the latest officially released iOS SDK.
- For the supported versions of iPad, iPad 2 iOS, please see *Supported Hardware and Software*
- For the supported versions of iPod touch and iPhone, please see *Supported Hardware and Software*. Previous device models do not have the dedicated cryptographical hardware and former iOS versions do not have the required security APIs
- XCode 3.x - gdb (XCode integrated debugger)
- Clang static code analyzer
- Instruments – a set of performance tools and profilers (Leaks, CPU Sampler, Activity Monitor)

OData SDK - iOS
The following figure shows the main components of the OData SDK on iOS.
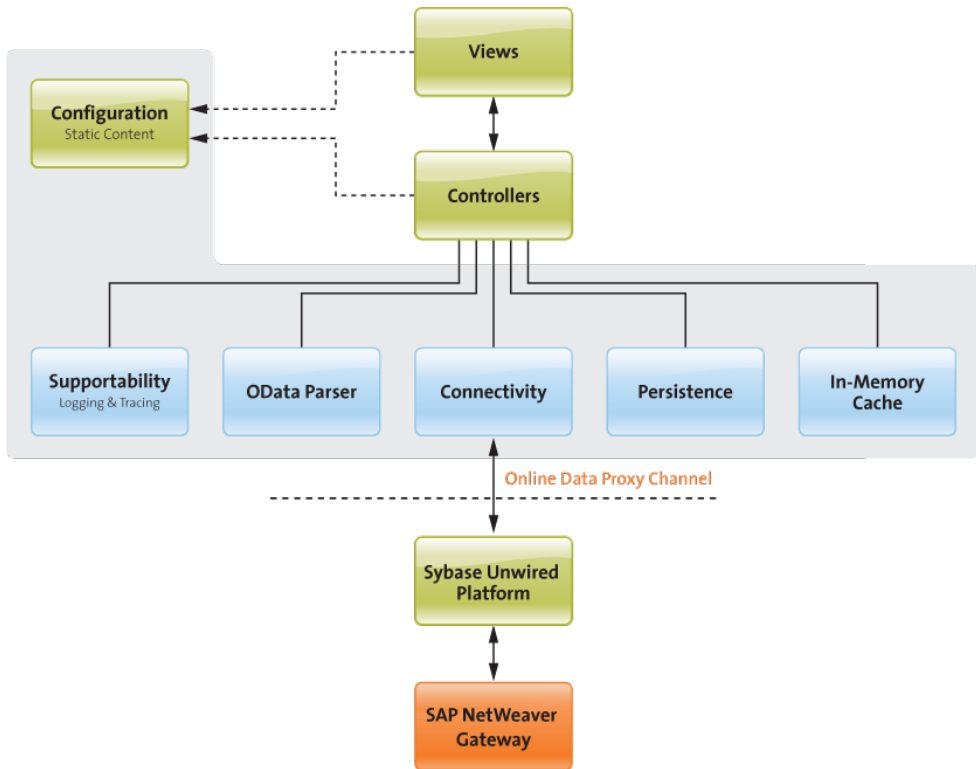
The iOS version of the OData SDK is presented as static libraries and header files. (Custom dynamic libraries are not allowed on iOS.)

The OData SDK for iOS includes a set of core iOS libraries acting independently from each other. Each core library has well-defined responsibilities and provides APIs for OData parsing, caching, persistence, keychain, certificate management, and so on.

The full list of APIs and their descriptions are available after the installation of Sybase Unwired Platform at the following location within your installation folder: `...\UnwiredPlatform\MobileSDK\OData\iOS\docs`

The libraries are provided in binary form as `.a` files, along with the public headers containing the APIs and the input/output structures. As a prerequisite, the public headers and the libraries must be available as separate binaries for release and debug, or merged using the lipo tool.

See also
*   *Developing Applications in the Xcode IDE* on page 8
*   *Deploying Applications to Devices* on page 41

SDMParser

The SDMParser library provides APIs to convert OData XML payloads to native Objective-C objects and structures (arrays, dictionaries).

List of Features

- OData XML or OData with SAP extensions XML (including inlined content) parsing and conversion to Objective-C objects
- URL template retrieval from open search description XMLs
- OData XML composition (create update scenario), also with SAP extensions
- OData error XML parsing
- Function import support
- Generates subscription XMLs
- Media Link Entries
- Convenient C-style APIs
- Action Link Support
- ETag Support

SDMParser Public APIs

```
SDMODataServiceDocument* sdmParseODataServiceDocumentXML(NSData*
const content_in)
SDMODataSchema* sdmParseODataSchemaXML(NSData* const content_in,
SDMODataServiceDocument* const serviceDocument)
NSMutableArray* sdmParseODataEntriesXML(NSData* const content_in,
const SDMODataEntitySchema* const entitySchema, const
SDMODataServiceDocument* const serviceDocument)
SDMODataError* sdmParseODataErrorXML(NSData* const content_in)
NSMutableArray* sdmParseFunctionImportResult(NSData* const
content_in, const SDMODataFunctionImport* const functionImport)
SDMOpenSearchDescription* sdmParseOpenSearchDescriptionXML(NSData*
const content_in)
SDMODataEntryXML* sdmBuildODataEntryXML (const SDMODataEntry *const
entry, const enum TEN_ENTRY_OPERATIONS operation, const
SDMODataServiceDocument *const serviceDocument, const BOOL
serializeInlinedEntries)
SDMODataFeedXML* sdmBuildODataFeedXML (NSArray *const entries, const
enum TEN_ENTRY_OPERATIONS operation, const SDMODataServiceDocument
*const serviceDocument, const BOOL serializeInlinedEntries)
(NSString *)getEtag
```

Technical Details

The listed C-style parser APIs are provided for convenience. You can choose to instantiate the dedicated parser classes. As a reference, the following code excerpt shows how the C-style APIs wrap the parser calls:

```
/**
 * Parses the service document XML and converts it to an Obj-C
service document object.
```

```
 */
SDMODataServiceDocument* sdmParseODataServiceDocumentXML(NSData*
const content_in) {
    SDMODataServiceDocumentParser* svcDocParser =
[[[SDMODataServiceDocumentParser alloc] init] autorelease];
    [svcDocParser parse: content_in];

    return svcDocParser.serviceDocument;
}


/**
 *    Parses and matches the schema with the service document and its
collections. The function returns the same
 *  schema pointer as it can already be found in the serviceDocument.
 */
SDMODataSchema* sdmParseODataSchemaXML(NSData* const content_in,
SDMODataServiceDocument* const serviceDocument) {
    if (!serviceDocument)
        //@throw [[[SDMParserException alloc] initWithName:
@"NoServiceDocument" reason: @"No service document was provided"
userInfo: nil] autorelease];
        @throw [[[SDMParserException alloc] initWithError:
ParserNoServiceDocument detailedError: @"No service document was
provided"] autorelease];

    SDMODataMetaDocumentParser* metaDocParser =
[[[SDMODataMetaDocumentParser alloc] initWithServiceDocument:
serviceDocument] autorelease];
    [metaDocParser parse: content_in];

    return serviceDocument.schema;
}


/**
 * Parses a feed or entry XML and returns an array of parsed entry/
entries.
 * Any "inlined"entries or feed(s) will be parsed when service
document is passed to the function. If "inlined" feed(s) or entries
 * should not be returned pass nil in the service document parameter.
 */
NSMutableArray* sdmParseODataEntriesXML(NSData* const content_in,
const SDMODataEntitySchema* const entitySchema, const
SDMODataServiceDocument* const serviceDocument) {
    if (!entitySchema)
        //@throw [[[SDMParserException alloc] initWithName:
@"NoEntitySchema" reason: @"No entity schema was provided" userInfo:
nil] autorelease];
        @throw [[[SDMParserException alloc] initWithError:
ParserNoEntitySchema detailedError: @"No entity schema was
provided"] autorelease];

    SDMODataDataParser* dataParser = [[[SDMODataDataParser alloc]
initWithEntitySchema: entitySchema andServiceDocument:
serviceDocument] autorelease];
```

```
    [dataParser parse: content_in];

    return dataParser.entries;
}


/**
 * Parses an OData error payload XML
 * @see SDMODataError
 */
SDMODataError* sdmParseODataErrorXML(NSData* const content_in) {
    SDMODataErrorXMLParser* errorParser = [[[SDMODataErrorXMLParser
alloc] init] autorelease];
    [errorParser parse: content_in];

    return errorParser.odataError;
}

/**
 * Parses the result payload XML of a function import.
 * @returns Returns an array of entries.
 * @remark Even if the result is not a feed or entry XML, the parser
creates an entity schema out of the return type definition, so
 * application developers can access the returned data in a uniform
way. The supported return types are:
 * - none
 * - EDMSimpleType       (for example: ReturnType="Edm.Int32"), the
generated "entity" schema will be "element" with type Edm.Int32
 * - ComplexType       (for example:
ReturnType="NetflixCatalog.Model.BoxArt")
 * - Collection of an EDMSimpleType (for example:
ReturnType="Collection(Edm.String)")
 * - Collection of a ComplexType    (for example:
ReturnType="Collection(NetflixCatalog.Model.BoxArt)")
 * - Entry    (for example ReturnType="NetflixCatalog.Model.Title"
EntitySet="Titles")
 * - Feed   (for example
ReturnType="Collection(NetflixCatalog.Model.Title)"
EntitySet="Titles")
 */
NSMutableArray* sdmParseFunctionImportResult(NSData* const
content_in, const SDMODataFunctionImport* const functionImport) {
    SDMFunctionImportResultParser* fiParser =
[[[SDMFunctionImportResultParser alloc] initWithFunctionImport:
functionImport] autorelease];
    [fiParser parse: content_in];

    return fiParser.entries;
}


/**
 * Parses an XML that contains Open Search Description
 * The parsed data is returned in an SDMOpenSearchDescription typed
object.
 */
```

```
SDMOpenSearchDescription* sdmParseOpenSearchDescriptionXML(NSData*
const content_in) {
    SDMOpenSearchDescriptionXMLParser* osdParser =
[[[SDMOpenSearchDescriptionXMLParser alloc] init] autorelease];
    [osdParser parse: content_in];

    return osdParser.openSearchDescription;
}
```

The SDMParser library communicates error conditions to the client via the dedicated
`SDMParserException` exception class. Whenever a mandatory attribute is missing, the
parser throws an exception.

The caller is responsible for error handling; this includes fetching the details included in the
exception, logging information meant for debugging purposes, displaying a localized alert
message, and providing a resolution or stopping the application flow.

The Service Document Component
Root object. Contains the schema object, the function imports, the document language, base
URL (if any) and the server type.

```
SDMOdataServiceDocument

-(enum TEN_SERVER_TYPES)getServerType
-(NSString*)getDocumentLanguage
-(NSString*)getBaseUrl
-(SDMODataSchema*)getSchema
-(NSMutableDictionary*)getFunctionImports
```

The Schema Component
The schema contains workspaces and helper methods to work with collections via
workspaces.

```
SDMODataSchema

-(NSArray*) getWorkspacesBySemantic:(const enum
TEN_WORKSPACE_SEMANTICS)workspaceSemantic
-(SDMODataCollection*) getCollectionByName:(NSString*
const)collectionName
-(SDMODataCollection*) getCollectionByName:(NSString*
const)collectionName workspaceOfCollection:
(SDMODataWorkspace**)workspaceOfCollection
```

The Workspace Component
A workspace can contain 0 up to n collections. Each workspace can have a title and a semantic
value.

```
SDMODataWorkspace

-(enum TEN_WORKSPACE_SEMANTICS) getSemantic
-(NSString*) getTitle
-(NSMutableDictionary*) getCollections
```

The Collection Component

Represents one parsed collection.

```
SDMODataCollection

-(id) initWithName:(NSString* const)newName
-(BOOL) isCreatable
-(BOOL) isUpdatable
-(BOOL) isDeletable
-(BOOL) isTopLevel
-(BOOL) doesRequireFilter
-(BOOL) hasMedia
-(SDMODataLink*) getSubscriptionLink
-(int) getContentVersion
-(enum TEN_COLLECTION_SEMANTICS) getSemantic
-(uint8_t) getFlags
-(NSString*) getName
-(NSString*) getTitle
-(NSString*) getMemberTitle
-(NSMutableArray*) getIcons
-(NSMutableArray*) getLinks
-(int) getDisplayOrder
-(SDMODataEntitySchema*) getEntitySchema
-(SDMOpenSearchDescription*) getOpenSearchDescription
```

The Entity Schema Component

An instance of the `EntitySchema` class stores the root of the structure of the given collection with constraints. The entity schema class also provides helper functions to order the visible fields of a collection and the navigation map that maps navigation names to collection names.

```
SDMODataEntitySchema

-(id) init
-(int) getContentVersion
-(uint16_t) getFlags
-(SDMODataPropertyInfo*) getRoot
-(NSMutableDictionary*) getNavigationMap
-(NSArray* const) getVisibleInListPathsInOrder
-(NSArray* const) getVisibleInDetailPathsInOrder
```

The Property Info Component

A property info instance stores the name, type and all constraints of a property, but does not store property values.

```
SDMODataPropertyInfo

-(id) initWithName:(NSString* const)propName andPropEdmType:(const
enum TEN_EDM_TYPES)propEdmType
-(BOOL) isNullable
-(BOOL) isKey
-(BOOL) isCreatable
```

```
-(BOOL) isUpdatable
-(BOOL) isFilterable
-(BOOL) isVisibleInList
-(BOOL) isVisibleInDetail
-(BOOL) isSearchable
-(BOOL) isServerGenerated
-(void) addChildPropertyInfo:(const SDMODataPropertyInfo*
const)child
-(SDMODataPropertyInfo* const) getPropertyInfoByPath:(NSString*
const)path
-(NSString*) getName
-(enum TEN_EDM_TYPES) getType
-(uint16_t) getFlags
-(int) getMaxLength
-(enum TEN_PROPERTY_SEMANTICS) getSemantic
-(uint32_t) getSemanticTypes
-(NSString*) getLabel
-(NSString*) getDescription
-(int32_t) getListDisplayOrder
-(int32_t) getDetailDisplayOrder
-(uint8_t) getScale
-(uint8_t) getPrecision
-(NSMutableDictionary*) getChildren
```

The Function Import Component

Function imports can be used to execute back-end functionalities that are not related to collections, or functionalities other than the possible create, update, delete and read operations for collections. An instance of SMDODataFunctionImport stores all the information and has all the methods necessary to execute such a back-end functionality.

```
SDMODataFunctionImport

-(id) initWithName:(NSString* const)newName
-(NSString*) getName
-(NSString*) getHttpMethod
-(NSMutableDictionary*) getParameters
-(SDMODataEntitySchema*) getReturnTypeSchema
-(uint8_t) getFlags
-(NSString*) getActionFor
-(NSMutableDictionary*) getWritableParameters
-(NSString*) generateFunctionImportUrl:(NSString* const)baseUrl
parameters:(NSDictionary* const)parameters
```

The Link Component

The OData SDK provides four types of link classes depending on the usecase:

- SDMODataLink
- SDMODataRelatedLink (this class inherits all the methods mentioned at SDMODataLink)
- SDMODataMediaResourceLink (this class inherits all the methods mentioned at SDMODataLink)

- SDMODataActionLink(contains the optional parameters of the action and the helper method to assemble the final URL that is required to execute the action)

```
SDMODataLink

-(NSString*) getHRef
-(NSString*) getRel
-(NSString*) getType
-(NSString*) getTitle
-(enum TEN_LINK_SEMANTICS) getSemantic


DMODataRelatedLink

-(NSString*) getTargetCollection


SDMODataMediaResourceLink

-(NSString*) getConcurrencyToken


SDMODataActionLink

-(NSString*) getHttpMethod;
-(NSMutableDictionary*) getDefaultParameterValues;
-(NSDictionary*) getParameters;
-(NSString*) createActionLinkURL:(NSDictionary*)parameters;
```

The Open Search Description Component
An `SDMOpenSearchDescription` instance stores the parsed short name, description and the URL templates for searching data.

```
SDMOpenSearchDescription

-(NSString*) getShortName
-(NSString*) getDescription
-(NSMutableArray*) getUrlTemplates

SDMOpenSearchDescriptionURLTemplate

-(NSString*) getUrlTemplate
-(NSString*) getUrlType
-(NSString*) createUrlWithParameters:(NSDictionary*)parameters
```

The Property Value Objects Component
An instance of the property value object stores a value and its metadata (property info instance). `SDMODataPropertyValueObject` is the base property value class that provides basic validation and value accessors. Derived classes of this class redefine certain methods (for example, validation checks) of the base class and provide methods allowing the library user to access data as typed data instead of string data.

```
SDMODataPropertyValueObject (base class)

-(NSString* const) getHTMLEncodedValue
-(NSString* const) getDefaultValue
-(BOOL) isValid
-(NSString*) getValue
-(void) setValue:(NSString*) value
-(enum TEN_EDM_TYPES) getEdmType
-(const SDMODataPropertyInfo* const) getPropertyInfo
-(BOOL) isValidationDisabled
-(void) setValidationDisabled:(BOOL)validationDisabled
```

Example - ETag Support

```
NSMutableArray *entries = nil;
//Get all entries in a given collection
entries = sdmParseODataEntriesXML([request responseData],
collection.entitySchema, nil);
for(int i=0; i<[entries count];i++) {
SDMODataEntry *entry = [entries objectAtIndex:i];
//For each entry get the ETag attribute
NSString *etag = [entry getEtag];
}
```

SDMCache

The SDMCache is a programming interface that provides in-memory cache for quick data access. Its APIs allow adding, removing and searching items stored in the cache. The cache also acts as a central, shared storage, avoiding the need to pass frequently used data between view controllers.

List of Features

- In-memory management of SDMOData-related objects
- Quick data filtering
- Prefix matching and regular expression support (default search method is prefix matching)

SDMCache Public APIs

```
- (void) setCapacity:(unsigned short) value
- (unsigned short) capacity
- (void) clear
- (void) setODataServiceDocument:(SDMODataServiceDocument*)
serviceDocument_in
- (id) initWithServiceDocument:(SDMODataServicedocument*)
serviceDoc_in
+ (id) cacheWithServiceDoc:(SDMODataServicedocument*) serviceDoc_in
- (NSArray*)filterEntriesOfCollection:(NSString*) collectionName_in
forSearchText:(NSString*)searchText_in
- (SDMODataServiceDocument*) getODataServiceDocument
- (BOOL) removeODataServiceDocument
- (void) setODataEntry:(SDMODataEntry*) entry_in byCollection:
(NSString*) collectionName_in
- (SDMODataEntry*) getODataEntryByCollection:(NSString*)
```

```
collectionName_in andEntryId:(NSString*) entryId_in
- (void) setODataEntries:(NSArray *) entries_in byCollection:
(NSString*) collectionName_in
- (NSArray*) getODataEntriesByCollection:(NSString*)
collectionName_in
- (NSArray*) getCollectionsByWorkspace:
(SDMDataWorkspace*)workspace_in
- (SDMODataCollection*) getCollectionByName:
(NSString*)collectionName_in
- (BOOL) removeODataEntry:(SDMODataEntry*)entry_in forCollection:
(NSString*) collectionName_in
- (BOOL) removeODataEntriesForCollection:(NSString*)
collectionName_in
- (NSArray*) getAllWorkspaces
- (NSArray*) getWorkspacesBySemantic:(enum TEN_WORKSPACE_SEMANTICS)
- (void) setShouldAutoSaveOnMemoryWarning:(BOOL)flag_in
- (BOOL) shouldAutoSaveOnMemoryWarning
- (void) setRegexSearchEnabled:(BOOL)flag_in
- (BOOL) isRegexSearchEnabled
```

Technical Details
The common methods are defined by the SDMCaching protocol. (See also: SDMCache
default implementation.)

SDMPersistence

The SDMPersistence library provides APIs to persist data to the device's physical storage.

List of Features

- Protected data storage to the device's filesystem using iOS 4.0 features
- Configurable storage policy
- Stores and loads NSData, SDMCache and objects adopting the NSCoding protocol

SDMPersistence Public APIs

```
+ (id) instance
- (void) clear
- (NSString*) storeCache:(id<SDMCaching>)cache_in
- (id<SDMCaching>) loadCache:(NSString*) uid_in
- (NSString*) storeData:(NSData*) data_in withId:(NSString*)uid_in
- (NSData*) loadData:(NSString*) uid_in
- (NSString*) storeSerializable:(id<NSCoding>) serializable_in
withId:(NSString*)uid_in
- (id<NSCoding>) loadSerializable:(NSString*) uid_in
- (StoragePolicy) storagePolicy
- setStoragePolicy:(StoragePolicy)policy_in
```

Technical Details
The common methods are defined by the SDMPersisting protocol. For builds targeting
iOS 4.0+ the default is FullProtectionStoragePolicy. (See also:
SDMPersistence default implementation.)

Consider using the SDMPersistence default implementation rather than implementing a custom persistence functionality.

Encryption, Secure Storage

Starting with iOS 4.0 (iOS 4.2 for iPad devices), data can be persisted in secure form. For builds targeting iOS 4.0+, the default storage policy is fully protected. In older iOS versions, data can only be persisted in unprotected form.

All data is stored in the app's dedicated filesystem, the so-called sandbox. The app's sandbox can be accessed exclusively by the app it belongs to. As the sandbox is bound to the app, deleting the app also removes its persisted data. Accessing data on iOS devices is fast and reliable, even when encryption is used.

For encryption, we rely on the Security framework and the dedicated cryptographical hardware available in the supported versions of iPad, iPad 2 and iPhone. Due to the lack of the cryptographical hardware, former iPhone models are not supported. The RSA keys required for the asymmetric key algorithm are retrieved from the app's keychain; if it is not available, they are generated and stored in the keychain during the first API call requiring the keys. For RSA key generation and keychain management, the iOS Security framework APIs are used.

A generic approach for secure data storage has been made available with iOS 4.0. Encryption and decryption of the device's filesystem is managed automatically by the operating system. This behavior is disabled by default, but can be enforced via corporate policy. It is possible to leave out the secure APIs from the library and solely rely on this approach.

Related reading: *http://support.apple.com/kb/HT4175*

WWDC video about secure data storage: *http://developer.apple.com/videos/wwdc/2010/*

SDMConnectivity

The Connectivity library exposes APIs required to set up and start HTTP requests, and retrieve the payloads. For crossing a company firewall for enterprise use cases, you need to use SUP. Therefore, the connectivity component in the OData SDK offers a connection to SUP by default. For development and demo purposes, the SDK also provides a possibility to use HTTP or HTTPS.

List of Features

- Synchronous and asynchronous HTTP request handling
- Concurrent request execution
- Continuous downloading and uploading when the app is sent to the background (iOS 4.0+ only)
- Timeout handling
- Supports compressed payload handling
- Notification about various events (failure, completion, authentication requests)

- Runtime switch between `SDMHttpRequest` and `SUPRequest` (SUP libraries are needed to be linked to the project) through `SDMRequestBuilder`
- ETag support

SDMConnectivity Public APIs

Note: The SUP APIs and their descriptions are available after the installation of Sybase Unwired Platform at the following location within your installation folder: `...\UnwiredPlatform\ClientAPI\apidoc`.

```
-(id)initWithURL:(NSURL *)newURL
+(id)requestWithURL:(NSURL *)newURL
-(void) setUserName:(NSString*)username
-(void) setPassword:(NSString*)username
-(void) setClientDelegate:(id)clientDelegate
-(void) setDidFinishSelector:(SEL)didFinishSelector
-(void) setDidFailSelector:(SEL)didFinishSelector
-(void) setRequestMethod:(NSString*)httpMethod
-(void) startAsynchronous
-(void) startSynchronous
-(void) cancel
-(void) setUploadProgressDelegate
-(void) setDownloadProgressDelegate
-(void) setShowAccurateProgress
-(void)setMaxConcurrentHTTPRequestCount:(const unsigned char)cnt
-(NSInteger) getMaxConcurrentHTTPRequestCount
-(NSString*) responseString
-(NSData*) responseData
-addRequestHeader:(NSString*)header value:(NSString*)value
-appendPostData:(NSData*)postData
-(void) buildPostBody
-(void) setClientCertificateIdentity:(SecIdentityRef)anIdentity
-(void) setEtag:(NSString*)etag withMatchType:
(EtagMatchType)matchType;
```

Example - Request Initialization

```
NSString* serverUrlWithLanguage = [NSString stringWithFormat:@"%@?
sap-language=%@",[ConnectivitySettings url],[[ConnectivitySettings
instance] currentLanguage]];
[self setRequest:[SDMHTTPRequest requestWithURL:[NSURL
URLWithString: serverUrlWithLanguage]]];

NSString* serverUrlWithLanguage = [NSString stringWithFormat:@"%@?
sap-language=%@",[ConnectivitySettings url],[[ConnectivitySettings
instance] currentLanguage]];
m_AsynchRequest = [HTTPRequest requestWithURL:[NSURL URLWithString:
serverUrlWithLanguage]];

[m_AsynchRequest
setDidFinishSelector:@selector(serviceDocFetchComplete:)];

[m_AsynchRequest
setDidFailSelector:@selector(serviceDocFetchFailed:)];
```

```
[m_AsynchRequest setRequestMethod:@"POST"];
```

Example - Request Execution

```
[m_AsynchRequest startSynchronous];

[m_AsynchRequest cancel];
```

Example - Progress Tracking

```
[m_AsynchRequest setUploadProgressDelegate:m_ProgressIndicator];
```

Example - Request Payload

```
NSString* stringPayload = [m_AsynchRequest responseString];

NSData* binaryPayload = [m_AsynchRequest responseData];
```

Example - Request Setup

```
[m_AsynchRequest addRequestHeader:@"myApplicationId" value:kAppId];
[m_AsynchRequest addRequestHeader:@"deviceType" value:@"iphone"];

[m_AsynchRequest appendPostData:urlEncData];
NSData* encodedPostData = [encodedPostStr
dataUsingEncoding:NSUTF8StringEncoding];
[m_AsynchRequest appendPostData:encodedPostData];
// once we have them all, build the POST body
 [m_AsynchRequest buildPostBody];

[m_AsynchRequest addRequestHeader:@"myApplicationId" value:kAppId];
[m_AsynchRequest addRequestHeader:@"deviceType" value:@"iphone"];
```

Example - ETag Support

```
id<SDMRequesting> request;
request = [SDMRequestBuilder requestWithURL:[NSURL
URLWithString:@"http://vmw3811.wdf.sap.corp:50075/sap/opu/sdata/
sap/FINCUSTFACTSHEET/
GOSNoteCollection(CustomerNo='MG001',CompanyCode='0001',NoteID='FOL
33000000000004RAW37000000000007')"]];
[request setEtag:@"201201130306299" withMatchType:EtagIfMatch];
[request setUsername:@"user"];
[request setPassword:@"password"];
 [request setDelegate:self];
 [request
setDidFinishSelector:@selector(collectionEntriesSuccess:)];
 [request setDidFailSelector:@selector(collectionEntriesFailure:)];
 [request startAsynchronous];
```

Technical Details

The SDMConnectivity library wraps internally the socket based CFNetwork APIs and uses NSOperationQueue to collect and fire asynchronous requests. The number of maximum concurrent HTTP requests is limited to MAX_CONCURRENT_THREADS (Default: 5).

The SDMHttpRequestDelegate protocol defines default delegate methods for request status related housekeeping. Client classes can implement this protocol to hook in for requestStarted / requestFinished / requestFailed default delegates.

In cases when clients prefer to use custom selectors for request notifications, they do not neééd to adapt the SDMHttpRequestDelegate protocol, but rather register themselves as delegates and set their own selectors as didStartSelector / didFinishSelector / didFailSelector.

The SDMProgressDelegate defines default delegate methods for upload and download progress notification. The protocol has to be adapted by client classes to hook in for didReceiveBytes / didSendBytes / incrementDownloadSizeBy / incrementUploadSizeBy delegates. You can choose to register a download or upload progress delegate using SDMHttpRequest instance methods -setUploadProgressDelegate and -setDownloadProgressDelegate.

The factory method instantiates SUPRequest by default. SUPRequest must be used to communicate through Online Data Proxy channel; it is part of the ODP SUP libraries, which have to be linked to the project. However, for development and testing purposes you can use HTTP requests. The SDMConnectivity library provides the means to transparently choose between SUPRequest using connections through the Online Data Proxy (ODP) Channel or SDMHttpRequest which leverages the usage of classical HTTP/HTTPS connections.

Protocols:

```
SDMHttpRequestDelegate

+ requestStarted:
+ requestFinished:
+ requestFailed:
+ requestRedirected:
+ request:didRecieveData:
+ authenticationNeededForRequest:
+ proxyAuthenticationNeededForRequest:


SDMProgressDelegate

+ setProgress:
+ request:didReceiveBytes:
+ request:didSendBytes:
+ request:incrementDownloadSizeBy:
+ request:incrementUploadSizeBy:
```

SDMSupportability

The SDMSupportability library provides functionality for logging, tracing and performance measurement.

SDMLogger

The SDMLogger is a programming interface that provides event logging facilities. Its APIs allow generating, retrieving and displaying log items for a specific application.

List of Features

- Easy-to-use APIs
- Low level system log access (via ASL methods)
- Convenience macros which automatically add additional information such as the invoker's file and method name, and line#
- Support for all iOS log levels
- Built-in log viewer
- Export capability via e-mail from the built-in log viewer; users can choose which entries to send, and the e-mail attachment is compressed
- Globalized resource bundle (to be included by clients): contains all the various labels and accessibility hints belonging to the LogViewer, translated to English, German, French, Spanish, Portuguese, Japanese, Russian, and traditional Chinese

Logger Macros

The SDMLOG macros wrap the logger APIs and automatically enhance the log entry with information such as FILE, FUNCTION and LINE#. The following macros are available (matching the exposed APIs):

```
SDMLOGEMERGENCY(msg)
SDMLOGALERT(msg)
SDMLOGCRITICAL(msg)
SDMLOGERROR(msg)
SDMLOGWARNING(msg)
SDMLOGNOTICE(msg)
SDMLOGINFO(msg)
SDMLOGDEBUG(msg)
```

SDMLogger Public APIs

```
+ (void) enableLogging
+ (void) disableLogging
-(void) displayLogsWithLevel:(LoggingLevels)level_in
-(void) displayLogsWithLevel:(LoggingLevels)level_in
forQueryOperation:(QueryOperations)queryOperation;
-(NSArray*) retrieveLogsWithLevel:(LoggingLevels)level_in;
-(NSArray*) retrieveLogsWithLevel:(LoggingLevels)level_in
forQueryOperation:(QueryOperations)queryOperation;
-(void) logMessage:(NSString*) message_in withLevel:
(LoggingLevels)level_in andInfo:(NSString*) info_in
```

```
-(void) logEmergency:(NSString*) message_in withInfo:(NSString*)
info_in
-(void) logAlert:(NSString*) message_in withInfo:(NSString*) info_in
-(void) logCritical:(NSString*) message_in withInfo:(NSString*)
info_in
-(void) logError:(NSString*) message_in withInfo:(NSString*) info_in
-(void) logWarning:(NSString*) message_in withInfo:(NSString*)
info_in
-(void) logNotice:(NSString*) message_in withInfo:(NSString*)
info_in
-(void) logInfo:(NSString*) message_in withInfo:(NSString*) info_in
-(void) logDebug:(NSString*) message_in withInfo:(NSString*) info_in
```

Technical Details

You can enable/disable logging, and display, retrieve and generate logs.

Higher priority log messages are mapped to lower values Mac OS X / iOS system wide (see asl.h). Therefore, use Less or LessEqual query operation to display more critical logs. For example, in order to retrieve all log messages including the lowest, Debug level ones, use the following approach:

```
[[SDMLogger instance] displayLogsWithLevel:DebugLoggingLevel
forQueryOperation:LessEqual];
```

When generating logs, consider using the SDMLOGxxx macros, because they automatically enhance the log entry with information such as FILE, FUNCTION and LINE#.

The common methods are defined by the SDMLogging protocol. (See also: SDMLogger default implementation.)

The built-in crash log support provided by Apple can be used additionally for supportability purposes. You can retrieve crash logs either by using iTunes or by using the iPhone Configuration Utility. Note that Apple imposes restrictions on an application transmitting any data about the user without the user's prior permission, as described in the App Store review guidelines at *http://developer.apple.com/appstore/resources/approval/guidelines.html*, see Chapter 17, Privacy.

SDMPerfTimer

The SDMPerfTimer is a high precision timer class, which uses a high performance timer providing a nanosecond granularity. This timer class is for accurate performance measurements.

List of Features

- Easy-to-use, tiny API set
- Provides high precision timer data
- Low initialization overhead

SDMPerfTimer Public APIs

```
- (uint_64t) getTimeElapsedInMilisec
- (void) start
- (void) reset
```

SAP Passport

For the Single Activity Trace an SAP® Passport has to be issued by the connectivity layer of the library.

The SAP Passport is transported as an HTTP header in the request. The server handles the SAP Passport to generate end-to-end Trace.

Deploying Applications to Devices

Complete steps required to deploy mobile applications to devices.

1. *Apple Push Notification Service Configuration*

   The Apple Push Notification Service (APNS) notifies users when information on a server is ready to be downloaded.

2. *Provisioning an Application for Apple Push Notification Service*

   Use Apple Push Notification Service (APNS) to push notifications from Unwired Server to the iOS application. Notifications can include badges, sounds, or custom text alerts. Device users can customize which notifications to receive through Settings, or turn them off.

3. *Preparing Applications for Deployment to the Enterprise*

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

4. *Sample Code to Enable APNS*

   Provides a code snippet on how to enable Apple Push Notification Services on your device.

See also
- *OData SDK Components and APIs* on page 24

Apple Push Notification Service Configuration

The Apple Push Notification Service (APNS) notifies users when information on a server is ready to be downloaded.

Apple Push Notification Service (APNS) allows users to receive notifications on iPhones. APNS:

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

Provisioning an Application for Apple Push Notification Service

Use Apple Push Notification Service (APNS) to push notifications from Unwired Server to the iOS application. Notifications can include badges, sounds, or custom text alerts. Device users can customize which notifications to receive through Settings, or turn them off.

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

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

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

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

2. Copy the enterprise certificate (`*.p12`) to the computer on which Sybase Control Center has been installed. Save the certificate in *UnwiredPlatform_InstallDir* `\Servers\MessagingServer\bin\`.

3. In Sybase Control Center, expand the **Servers** folder and click **Server Configuration** for the primary server in the cluster.

4. In the **Messaging** tab, select **Apple Push Configuration**, and:

   a) Configure Application name with the same name used to configure the product name in Xcode. If the certificate does not automatically appear, browse to the directory.

   b) Change the push gateway information to match that used in the production environment.

   c) Restart Unwired Server.

5. Verify that the server environment is set up correctly:

   a) Open *UnwiredPlatform_InstallDir*`\Servers\UnwiredServer` `\logs\APNSProvider`.

   b) Open the log file that should now appear in this directory. The log file indicates whether the connection to the push gateway is successful or not.

6. Deploy the application and the enterprise distribution provisioning profile to your users' computers.

7. Instruct users to use iTunes to install the application and profile, and how to enable notifications. In particular, device users must:

- Download the Sybase application from the App Store.
- In the iPhone Settings app, slide the **Notifications** control to **On**.

8. Verify that the APNS-enabled iOS device is set up correctly:

   a) Click **Device Users**.

   b) Review the Device ID column. The application name should appear correctly at the end of the hexadecimal string.

   c) Select the Device ID and click **Properties**.

   d) Check that the APNS device token has been passed correctly from the application by verifying that a value is in the row. A device token appears only after the user is registered with the application in Sybase Control Center.

9. Test the environment by initiating an action that results in a new message being sent to the client.

   If you have verified that both device and server can establish a connection to APNS gateway, the device will receive notifications and messages from the Unwired Server, including workflow messages, and any other messages that are meant to be delivered to that device. Allow a few minutes for the delivery or notification mechanism to take effect and monitor the pending items in the Device Users data to see that the value increases appropriately for the applications.

10. To troubleshoot APNS, use the *UnwiredPlatform_InstallDir*\Servers \Unwired Server\log\trace\APNSProvider log file.

Preparing Applications for Deployment to the Enterprise

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

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

1. Sign up for the iPhone Developer Program, which gives you access to the Developer Connection portal. Registering as an enterprise developer gets you the certificate you need to sign applications.

2. Create a certificate request on your Mac through Keychain.

3. Log in to the Developer Connection portal.

4. Upload your certificate request.

5. Download the certificate to your Mac. Use this certificate to sign your application.

6. Create an AppID.

   Verify that your info.plist file has the correct AppID and application name. Also, in Xcode, right-click **Targets** < **<your_app_target>** and select **Get Info** to verify the AppID and App name.

7. Create an enterprise provisioning profile and include the required device IDs with the enterprise certificate. The provisioning profile authorizes devices to use applications you have signed.

8. Create an Xcode project ensuring the bundle identifier corresponds to the bundle identifier in the specified App ID. Ensure you are informed of the "Product Name" used in this project.

Sample Code to Enable APNS

Provides a code snippet on how to enable Apple Push Notification Services on your device.

Examples

- **Enable APNS –**

```
//Enable APNS
To enable APNS, you need to implement the following in the
application delegate of the application that has to receive
notifications.
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOp-tions:(NSDictionary *)launchOptions {
    [LiteSUPMessagingClient setupForPush:application];
}
* Callback by the system where the token is provided to the client
application so that this
 can be passed on to the provider. In this case,
"deviceTokenForPush" and "setupForPush"
are APIs provided by SUP to enable APNS and pass the token to SUP
Server

- (void)application:(UIApplication *)app
didRegisterForRemoteNotifications-WithDe-viceToken:(NSData
*)deviceToken {
    [LiteSUPMessagingClient deviceTokenForPush:app
deviceToken:deviceToken];
}
* Callback by the system if registering for remote notification
failed.
- (void)application:(UIApplication *)app
didFailToRegisterForRemoteNotifica-tionsWi-thError:(NSError *)err
{
    [LiteSUPMessagingClient pushRegistrationFailed:app
errorInfo:err];
        }
// You can alternately implement the pushRegistrationFailed API:
// +(void)pushRegistrationFailed:(UIApplication*)application
errorInfo: (NSError *)err
* Callback when notification is sent.
- (void)application:(UIApplication *)application
didReceiveRemoteNotifica-tion:(NSDictionary *)userInfo {
    [LiteSUPMessagingClient pushNotification:application
notifyData:userInfo];
}
```

CHAPTER 3   **Developing Android Applications**

Provides information about using advanced Sybase® Unwired Platform features to create applications for Android devices. The audience is advanced developers who are familiar working with APIs, but who may be new to Sybase Unwired Platform.

Describes requirements for developing a device application for the platform. Also included are task flows for the development options, procedures for setting up the development environment and API references.

1. *Setting Up the Development Environment*

Set up the Android Development Environment by downloading the required plugins.

2. *Developing Applications in the Android Development Environment*

After you import mobile applications and associated libraries into the Android development environment, use the Android API references to customize your device applications.

3. *OData SDK Components and APIs*

The Android OData SDK provides a set of features that help application developers build new applications on top of the Android platform. It supports the usage of the OData protocol with SAP additions (OData for SAP) and provides solutions for the most common use-cases an application developer meets with.

4. *Deploying Applications to Devices*

This section describes how to deploy customized mobile applications to devices.

Setting Up the Development Environment

Set up the Android Development Environment by downloading the required plugins.

Prerequisites

- Download the Java Standard Edition (6 Update 24) Development Kit from the following URL: *http://www.oracle.com/technetwork/java/javase/downloads/index.html*

- Download Eclipse Helios (3.6.2) from the following URL: *http://www.eclipse.org/downloads/*

Task

1. Start the Eclipse environment.
2. From the **Help** menu, select **Install New Software**.
3. Click **Add**.
4. In the Add Repository dialog, enter a **Name** for the new plugin.
5. Enter one of the following for **URL**:

    - https://dl-ssl.google.com/android/eclipse/
    - http://dl-ssl.google.com/android/eclipse/

6. Click **OK**.
7. Select the **Developer Tools** checkbox and click **Next**.
8. Review the tools to be downloaded.
9. Click **Next**.
10. Read and accept the license agreement and click **Finish**.
11. Once the installation is complete, restart Eclipse.

See also

- *Developing Applications in the Android Development Environment* on page 47

Setting Up the Android SDK Library in the Plugin

Set up the Android SDK in the ADT Plugin.

1. In the Eclipse environment, from the Window menu, select **Preferences**.
2. In the left navigation pane, select the **Android** node.
3. Click **Browse** to search for the location where you have stored the Android SDK.
4. Click **Apply** and **OK**.

Importing Libraries to your Android Application Project

Reference the libraries required for the Android application project.

1. Download the SDK/ ODP library files to your host development system.
   For Online Data Proxy, you need to download the following .jar files from the location
   `<UnwiredPlatform_InstallDir>\MobileSDK\OData\Android`
   `\libraries\:`
   - sup-json.jar
   - ClientLib.jar
   - SUPProxyClient*.jar
2. Create a new folder, named `libs`, in your Eclipse/Android project.

3. Right click `libs` and choose `Import -> General -> File System`, then click **Next**.

4. Browse the file system to find the library's parent directory (where you downloaded it).

5. Click **OK**, then click the directory name (not the checkbox) in the left pane and check the relevant JAR in the right pane. This puts the library into your project (physically).

6. Right click on your project, choose `Build Path -> Configure Build Path`, then click the Libraries tab, then click **Add JARs...**

7. Navigate to your new JAR in the libs directory and add it. (This is when your new JAR is converted for use on Android.)

This procedure includes a Dalvik-converted JAR in your Android project and makes Java definitions available to Eclipse in order to find the third-party classes when compiling your project's source code.

Online Data Proxy Android API Reference File and JAR File Locations

The Online Data Proxy JAR files and dependencies are installed in the Sybase Unwired Platform installation directory.

The contents and location of the .jar files:

* `<UnwiredPlatform_InstallDir>\MobileSDK\OData\Android\libraries\`

The API references can be extracted from the following zip files:

* `<UnwiredPlatform_InstallDir>\MobileSDK\OData\Android\docs\SUPProxyClient-2.1.1-Docs.zip`
* `<UnwiredPlatform_InstallDir>\MobileSDK\OData\Android\docs\AndroidODataSDK-1.1.0-doc.zip`

Developing Applications in the Android Development Environment

After you import mobile applications and associated libraries into the Android development environment, use the Android API references to customize your device applications.

This section provides a quick reference to APIs used for developing Android Applications.

For a comprehensive list of API references, extract the contents from the following zip files:

* `<UnwiredPlatform_InstallDir>\MobileSDK\OData\Android\docs\SUPProxyClient-2.1.1-Docs.zip`
* `<UnwiredPlatform_InstallDir>\MobileSDK\OData\Android\docs\AndroidODataSDK-1.1.0-doc.zip`

See also
- *Setting Up the Development Environment* on page 45
- *OData SDK Components and APIs* on page 59

Initializing an Application

Initialize an application.

Syntax

```
public static void initInstance(android.content.Content context,
String appID) throws com.sybase.mo.MessagingClientException
```

Parameters

- **context** – Name of the application context.
- **appID** – Name of the registered application.

Examples

- **Initialize an application** –

```
LiteUserManager.initInstance(getApplicationContext(),
"SampleUserProcessing");
        LiteUserManager lurm = LiteUserManager.getInstance();
```

Setting Connection Profile

Set the server details.

Syntax

```
public void setConnectionProfile(String host, int port, String
farmID)
```

Parameters

- **host** – IP Address of the ODP server.
- **port** – Port number of the server.
- **farmID** – This is the company name/ID.

Examples

- **Setting the server details** –

```
lurm.setConnectionProfile("10.66.148.17", 5001, "0");
```

Manually Registering an Application

Manually register an application by using the user name and activation code of the application registered through the Sybase Control Center.

Syntax

Synchronous Registration

```
public void registerUser(String username, String activationCode)
throws com.sybase.mo.MessagingClientException
```

Asynchronous Registration

```
public void asyncRegisterUser(String username, String
activationCode) throws com.sybase.mo.MessagingClientException
```

Parameters

- **username** – User name specified in SCC
- **activationCode** – Activation Code specified in SCC

Examples

- **Register the application manually** –
  ```
  LiteUserManager.registerUser(userName, activationCode);
  ```

Automatically Registering an Application using SSO2 Cookie

Registering an application automatically using an SSO2 Token Cookie. This token is fetched from a ticket issuing system and verified by the server.

Syntax

Synchronous Registration

```
public void registerUser(String username, String securityConfig,
String password, String vaultPassword) throws
com.sybase.mo.MessagingClientException
```

Asynchronous Registration

```
public void asyncRegisterUser(String username, String
securityConfig, String password, String vaultPassword) throws
com.sybase.mo.MessagingClientException
```

Parameters

- **username** – User name of the ticket issuing system.
- **securityConfig** – Security configuration of the registered application provided by the administrator in the Sybase Control Center.

- **password** – Password used to authenticate the user.
- **vaultPassword** – Password of the vault.

Examples

- **Registering a user using SSO2 Cookie –**
  ```
  lurm.registerUser("supuser","SSO2Cookie","s3puser","dtapass");
  ```

Automatically Registering an Application using HTTP Authentication Provider

Registering an application automatically using the HTTP Authentication Provider.

Syntax

```
public void registerUser(String username, String securityConfig,
String password, String vaultPassword) throws
com.sybase.mo.MessagingClientException
```

Parameters

- **username** – Valid user name.
- **securityConfig** – Security configuration of the registered application provided by the administrator in the Sybase Control Center.
- **password** – Password used to authenticate the user.
- **vaultPassword** – Password required to unlock the data vault .

Examples

- **Registering an application using HTTP Authentication Provider –**
  ```
  lurm.registerUser("supuser","SUPGWCHttpAuthConfig","s3puser","dta
  pass");
  ```

Automatically Registering an Application using X.509 Certificate

Registering an application automatically using an X.509 Certificate. This certificate is fetched from a Certificate Authority and verified by the server.

Syntax

Synchronous Registration

```
public void registerUser(String username, String securityConfig,
String password) throws com.sybase.mo.MessagingClientException
```

Asynchronous Registration

```
public void asyncRegisterUser(String username, String
securityConfig, String password) throws
com.sybase.mo.MessagingClientException
```

Parameters

- **username** – Valid user name
- **securityConfig** – Security configuration of the registered application provided by the administrator in the Sybase Control Center.
- **password** – Contains the Base64 encoded string of the certificate library.

Examples

- **Registering a user using X.509 Certificate** –

```
LiteUserManager lurm = LiteUserManager.getInstance();
lurm.registerUser("SUPUSER1","SUPGWCCERTConfig",
LiteCertificateStore.getInstance().getSignedCertificateFromFile("
/data/SUPUSER1.p12", "mobile");
```

Storing the Application Credentials Securely

Post user registration, if you want the user credentials to be managed by SDK, you can provide a data vault password to securely store the data.

Syntax

```
public void setAppCredentials(String username, String
securityConfig, String password, String vaultPassword) throws
LiteDataVaultException
```

Parameters

- **username** – Valid user name to be stored.
- **securityConfig** – Security configuration of the registered application to be stored.
- **password** – If using certificates, this corresponds to the Base64 encoded string of the certificate library. If using SSO2 cookie, this corresponds to the passowrd of the ticket issuing system.
- **vaultPassword** – Password of the secure store provided by SDK.

Examples

- **Storing the application credentials securely** –

```
lurm.setAppCredentials(username, sceurityConfig, password,
vaultPassword)
```

Getting Application End-point

Retrieve the application end-point that corresponds to the gateway service document.

Syntax

```
public String getApplicationEndPoint() throws
com.sybase.mo.MessagingClientException
```

Returns

If successful, this function returns the application end-point.

Examples

- **Retrieving application end-point –**

```
LiteAppSettings las = new LiteAppSettings();
las.getApplicationEndPoint()
```

Getting the Push End-point

Retrieve the push end-point that corresponds to the delivery address that the application uses in the subscription request for notifications.

Syntax

```
public String getPushEndPoint() throws
com.sybase.mo.MessagingClientException
```

Examples

- **Retrieve the push end-point –**

```
LiteAppSettings las = new LiteAppSettings();
las.getPushEndPoint();
```

Getting Server Details

Retrieve the SUP server host name.

Syntax

```
public String getServer() throws
com.sybase.mo.MessagingClientException
```

Returns

Returns the server name as a string.

Examples

- **Retrieve the server details –**

```
LiteAppSettings appSettings = new LiteAppSettings();
String ServerName = appSettings.getServer();
```

Getting Port Number

Retrieve the port number provisioned in the client repository.

Syntax

```
public int getPortNumber() throws
com.sybase.mo.MessagingClientException
```

Returns

Returns the port number as an integer.

Examples

- **Retrieve the port number –**

```
LiteAppSettings appSettings = new LiteAppSettings();
int PortNo = appSettings.getPortNumber();
```

Getting FarmID

Retrieve the Farm ID of the SUP Server.

Syntax

```
public String getFarmID() throws
com.sybase.mo.MessagingClientException
```

Returns

Returns the Farm ID as a string.

Examples

- **Retrieve the Farm ID –**

```
LiteAppSettings appSettings = new LiteAppSettings();
String FarmID = appSettings.getFarmID();
```

Checking the Provisioning Status of the Public Key

Check if the public key is provisioned on the client.

Syntax

```
public boolean IsSUPKeyProvisioned() throws
come.sybase.mo.MessagingClientException
```

Returns

If the key is provisioned, the value 'true' is returned, else 'false'.

Examples

- **Check the provisioning status of the public key –**

```
LiteAppSettings appSettings = new LiteAppSettings();
if(appSettings.IsSUPKeyProvisioned())
{
Log.i(null,"IsSUPKeyProvisioned is true");
}
```

Deleting Users

Deletes a registered user. If the user credentials are managed by SDK, this API deletes the user credentials from the vault and deletes the user from the server.

Syntax

```
public void deleteUser()
```

Getting Application Seed Data from Afaria

Get the application seed data from Afaria.

Syntax

```
public static Hashtable getSettingsFromAfaria() throws
com.sybase.afaria.SeedDataAPI.SeedDataAPIException, IOException
```

Returns

Returns a hashtable containing the settings.

Provisioning Connection Settings from Afaria

Connection Settings for an application can be provisioned using the Afaria client that is installed on the mobile device.

Syntax

```
public void setConnectionProfileFromAfaria() throws
com.sybase.mo.MessagingClientException
```

Examples

- **Provisioning the Connection Settings from Afaria –**
  ```
  LiteUserManager lurm = LiteUserManager.getInstance();
  lurm.setConnectionProfileFromAfaria();
  ```

Provisioning Certificates using Afaria

Certificates can be provisioned for Android devices using Afaria.

Syntax
```
static String getSignedCertificateFromAfaria (String CN, String
challengeCode) throws com.sybase.persistence.SSOCertManagerException
```

Parameters

- **CN –** A character-type column name, variable, or constant expression of char, varchar, nchar, nvarchar, or unichar type.
- **challengeCode –** Another character-type column name, variable, or constant expression of char, varchar, nchar, nvarchar, or unichar type.

Returns

Returns a signed certificate.

Examples

- **Provisioning Certificates from Afaria –**
  ```
  LiteCertificateStore.getInstance().getSignedCertificateFromAfaria
  ("sample","~");
  ```

Clearing the Server Verification Key

For a device to switch connection between SUP servers, this API is invoked before registering a new user. This ensures that the server public keys are removed from the SUP client SDK which enables connectivity to the new SUP Server.

Syntax
```
public void clearServerVerificationKey() throws
com.sybase.mo.MessagingClientException
```

Examples

- **Clear the server verification key –**
  ```
  lurm.clearServerVerificationKey();
  ```

Enabling Online Push for Applications

To consume push messages, the application registers a listener object.The client SDK notifies this listener object whenever there is a push message from the server. The listener object should implement the ISDMNetListener interface.

Syntax

```
public void
doPushRegistration(com.sap.mobile.lib.sdmconnectivity.ISDMNetListen
er push)
```

Parameters

- **push** – Object that implements ISDMNetListener interface.

Enabling the Listener for Proxy Setting Changes

To consume updates when there are changes in the Proxy settings, the application registers a listener object.The client SDK notifies this listener object whenever there is a settings update from the server. The listener object should implement the SUPLiteConfigurationChangeListener interface.

Syntax

```
public void
addConfigurationChangeListener(SUPLiteConfigurationChangeListener
oListener) throws com.sybase.mo.MessagingClientException
```

Parameters

- **oListener** – Object that implements the SUPLiteConfigurationChangeListener interface.

Data Vault API References

The data vault is a secure storage area provided by the SUP 2.1 SDK client libraries to store sensitive data such as usernames, passwords, authentication certificates within the application. Access to the data vault is protected by two levels of passwords and unique salts.

Creating a Vault

Creates an instance of a vault with a set of attributes.

Syntax

```
public static LiteDataVault createVault(String sDataVaultID, String
sPassword, String sSalt) throws LiteDataVaultException
```

Parameters

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

Returns

If successful, creates an instance of LiteDataVault.

Opening an Existing Vault

Returns the LiteDataVault singleton instance, tied to a particular vault. If the vault does not exist or has been deleted, this method throws an exception.

There is a singleton instance per data vault ID.

Syntax

```
public static LiteDataVault getVault(String sDataVaultID) throws
LiteDataVaultException
```

Parameters

- **sDataVaultID –** The vault name.

Returns

If successful, returns a singleton instance of the vault..

Deleting a Vault

Delete the storage for this instance from the persistent storage. Once a vault is deleted, all current instance references become invalid.

Syntax

```
public static void deleteVault (String sVaultId) throws
LiteDataVaultException
```

Parameters

- **sVaultId –** The vault name.

Locking a Vault

Lock a vault to avoid it from being used. If the vault is locked, this API will have no effect.

Syntax

```
public void lock() throws LiteDataVaultException
```

Unlocking a Vault

Unlock a vault for use by an application.

Syntax

```
public void unlock(String sPassword, String sSalt) throws
LiteDataVaultException
```

Parameters

- **sPassword** – The vault password.
- **sSalt** – The vault's salt password.

Setting a Binary Value

Store a value in the vault. To remove a value, provide 'null' as the second parameter.

Syntax

```
public void setValue(String sName, byte[] abValue) throws
LiteDataVaultException
```

Parameters

- **sName** – The key in which you store the data.
- **abValue** – The value you want to store.

Retrieving a Binary Value

Retrieve a value set from the vault.

Syntax

```
public byte[] getValue (String sName) throws LiteDataVaultException
```

Parameters

- **sName** – The key that contains the data you want to retrieve

Returns

If successful, returns the value stored in the key.

Setting the Retry Limit Value for a Vault

Set the maximum number of consecutive failed attempts to unlock the vault.

<u>**Syntax**</u>

```
public void setRetryLimit(int iLimit) throws LiteDataVaultException
```

<u>**Parameters**</u>

- **iLimit –** Maximum failed attempts that is permitted to unlock the vault.

<u>**Setting the Lock Timeout Value for a Vault**</u>

Set the time until which the vault remains in an unlocked state. Once this time is lapsed, the vault reverts to the locked state.

<u>**Syntax**</u>

```
public void setLockTimeout(int iTimeoutSeconds) throws
LiteDataVaultException
```

<u>**Parameters**</u>

- **iTimeoutSeconds –** Time in seconds for which the vault is unlocked.

OData SDK Components and APIs

The Android OData SDK provides a set of features that help application developers build new applications on top of the Android platform. It supports the usage of the OData protocol with SAP additions (OData for SAP) and provides solutions for the most common use-cases an application developer meets with.

Prerequisites for Developing Android Apps

Download the Android Software Development Kit. The recommended development environment is Eclipse IDE (version 3.5 and higher). Also download the Android java plug-in for Eclipse. For more details about Android SDK end Eclipse plug-in installation, see: *http://developer.android.com/sdk/installing.html*

The Android OData SDK also provides emulator support for testing, however, in Android platform, debugging and testing on real devices are more effective. To deploy your application directly to a real device, first install the driver of the device on your computer. For debugging an application on a real device, change the settings of your device to accept non-market applications. (You can change the setting at `Settings > Application > Development`)

Each component of the Android OData SDK can be imported to your project as an external library. The components are built on top of the Android SDK with API level 8.

OData SDK - Android

The full list of APIs and their descriptions are available after the installation of Sybase Unwired Platform at the following location within your installation folder: `...
\UnwiredPlatform\MobileSDK\OData\Android\docs`

The following figure shows the main components of the OData SDK on Android.
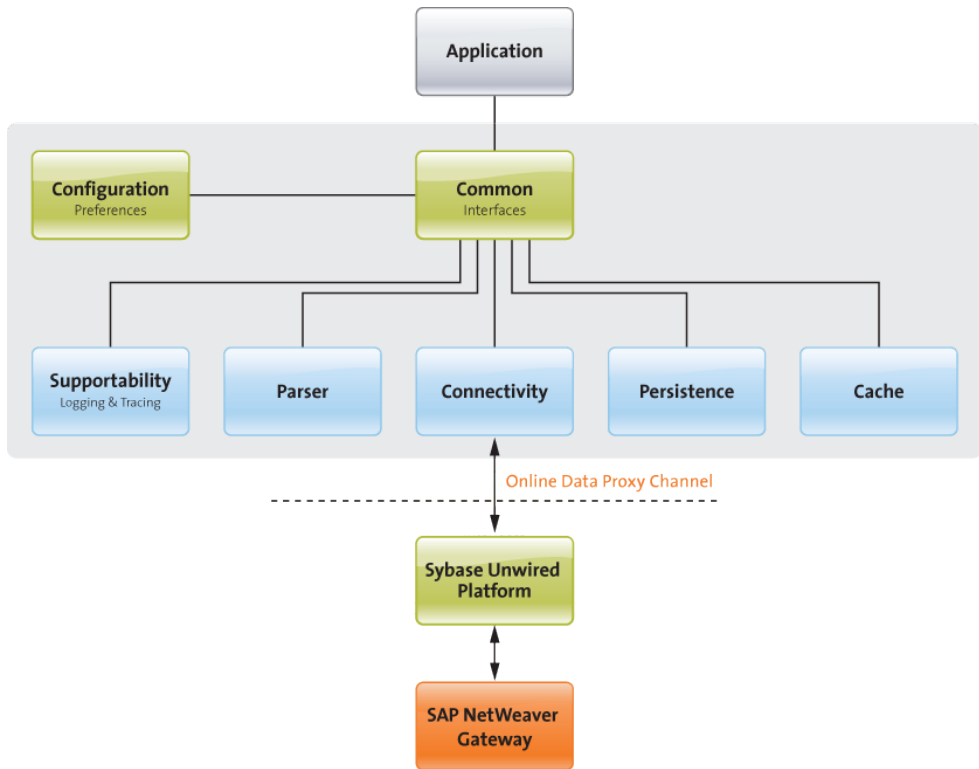
Each component is implemented as a standalone Java project, so they are available for application developers as separate external libraries (jar files). You also need the SDMCommon component to be able to reuse any other components from the Android OData SDK.

SDMCommon

To build an application on the OData SDK, you must first import the SDMCommon component that contains interfaces and configuration for the components. None of the components have dependency on each other, but all of them depend on the SDMCommon component, and all of them have references to interfaces of other components (held by SDMCommon).

Component Replacements

In your own application, you can replace the implementation behind an interface of an Android OData SDK component. For example, if you want to add a new functionality to SDMCache, but keep everything else unchanged (for example, the way it is persisted by SDMPersistence) you can implement your own solution. The new cache can be either a new implementation, or a descendant of SDMCache, as long as it implements the ISDMCache interface from SDMCommon.

See also

SDMParser

The SDMParser component is responsible for transforming between the different representations of OData structures, for example, parsing from XMLs to a Java Object or building XMLs from a Java Object.

List of Features

- Parsing OData XML structures to OData Java Objects
- Providing direct access to the most common OData fields and structures in the Java Objects that are the outcome of parsing
- Providing dynamic access to all OData fields and structures in the Java Objects that are the outcome of parsing
- Building OData XML structures from OData Java Objects
- Partial validation of OData XMLs

SDMParser Public APIs

```
ISDMParser

ISDMODataServiceDocument parseSDMODataServiceDocumentXML(String
serviceDocumentXML)
ISDMODataServiceDocument
parseSDMODataServiceDocumentXML(InputStream stream)

ISDMODataSchema parseSDMODataSchemaXML(String schemaXML,
ISDMODataServiceDocument serviceDocument)
ISDMODataSchema parseSDMODataSchemaXML(InputStream stream,
ISDMODataServiceDocument serviceDocument)

List<ISDMODataEntry> parseSDMODataEntriesXML(String entriesXML,
String collectionId, ISDMODataSchema schema)
List<ISDMODataEntry> parseSDMODataEntriesXML(InputStream stream,
String collectionId, ISDMODataSchema schema)

ISDMODataOpenSearchDescription
```

```
parseSDMODataOpenSearchDescriptionXML(String
openSearchDescriptionXML, String collectionId,
ISDMODataServiceDocument serviceDocument)
ISDMODataOpenSearchDescription
parseSDMODataOpenSearchDescriptionXML(InputStream stream, String
collectionId, ISDMODataServiceDocument serviceDocument)

ISDMODataError parseSDMODataErrorXML(String errorXML)
ISDMODataError parseSDMODataErrorXML(InputStream stream)

List<ISDMODataEntry> parseFunctionImportResultXML(String xml,
ISDMODataFunctionImport functionImport, ISDMODataSchema schema)
List<ISDMODataEntry> parseFunctionImportResultXML(InputStream
stream, ISDMODataFunctionImport functionImport, ISDMODataSchema
schema)

String buildSDMODataEntryXML(ISDMODataEntry entry)
String buildSDMODataDocumentXML(ISDMParserDocument document)

ISDMParserDocument parseXML(String xml)
ISDMParserDocument parseXML(InputStream stream)
```

Example

```
    try {
     //Parsing a feed or a single entry.
     //Assuming that schema and service document already parsed
     //and collection selected
     List<ISDMODataEntry> entries =
        parser.parseODataEntriesXML(responseXML,collectionId,schema);
     //Assuming there is at least one entry in the feed.
     ISDMODataEntry entry = entries.get(0);
     //Retrieving the valid property meta data from the given
SDMOdataSchema.
     List<ISDMODataProperty> properties = entry.getPropertiesData();
     //Assuming there is at least one property for the entry.
     ISDMODataProperty property = properties.get(0);
     boolean visibleInList = property.getAttribute("visible-in-
list");
     String value;
     if (visibleInList) {
         value = property.getValue();
     } else {
         value = "invisible";
     }
    } catch(SDMParserException e) {}
```

Technical Details

The SDMParser component uses javax.xml.parsers.SAXParser as a parser engine, defining its
own extension of `org.xml.sax.helpers.DefaultHandler` class as a handler for
SAXParser.

The outcome documents of SDMParser are all optimized for persistence using SDMPersistence, implementing the ISDMPersistable interface.

To support optimized performance and ensure consistent behavior, SDMParser can persist parsing related data on the device. End users can not delete parser related persisted data, unless they uninstall the whole application. To set the default folder of SDMParser's persistence, change the default value of the appropriate preference: PARSER_DEFAULTFOLDER_PATH (see more at the section about the SDMConfiguration component of the Android OData SDK).

Parsing related data is loaded during the initialization of the SDMParser component. This means that SDMParser must always be initialized before using the SDMParser documents.

As a result of parsing, SDMParser provides Java Object representations of the appropriate OData structures. Each such SDMOData Java Object is a representation of the appropriate Data XML and provides dynamic access to all of its elements and attributes. Besides the full access with the dynamic method, OData Java Objects provide interfaces for a more convenient access of data used in the most common scenarios.

SDMCache

The SDMCache component is responsible for storing and accessing OData related objects in the memory of the device.

List of Features

- Storing SDMOData document objects in the memory
- Accessing SDMOData documents in the memory directly by their key
- Searching for SDMODataEntry objects in the memory using tokenized prefix search on their searchable fields
- Searching for SDMODataEntry objects in the memory using one of the following predefined algorithm: full term prefix search, tokenized contain search, full term contain search, tokenized contain-all search and regex search
- Searching for SDMODataEntry objects in the memory using custom search algorithm
- Managing the number of stored SDMOData documents based on the maximum size of the capacity, removing the least recently used SDMOData document first
- Validating the references between the stored SDMOData Service Document, SDMOData Schema and the SDMOData Entries

SDMCache Public APIs

```
ISDMCache

    void clear();
    void setSDMODataServiceDocument(ISDMODataServiceDocument
serviceDocument);
    void setSDMODataSchema(ISDMODataSchema schema);
   void setSDMODataEntry(ISDMODataEntry entry, String collectionId);
    void setSDMODataEntries(List<ISDMODataEntry> entries, String
```

```
collectionId);
    void setSDMODataDocument(ISDMParserDocument document);
    ISDMODataServiceDocument getSDMODataServiceDocument();
    ISDMODataSchema getSDMODataSchema();
    ISDMODataEntry getSDMODataEntry(String key);
    List<ISDMODataEntry> getSDMODataEntries(String collectionId);
    ISDMParserDocument getSDMODataDocument(String key);
    List<ISDMODataEntry> searchSDMODataEntries(String searchTerm,
String collectionId);
    void removeSDMODataServiceDocument();
    void removeSDMODataSchema();
    void removeSDMODataDocument(String key);
    void removeSDMODataEntries(String collectionId);
    void removeStoredDocuments();
    int getSearchAlgorithm();
    void setSearchAlgorithm(int searchAlgorithm);
    void setEntrySearch(ISDMEntrySearch entrySearch);
```

Technical Details

For capacity management, SDMCache uses an LRU (least recently used) algorithm that ensures that the most recently used entries will not be removed first because of reaching the maximum capacity. Maximum number of capacity can be set using preference with key: ISDMPreferences.SDM_CACHE_CAPACITY. This setting refers to the maximum number of cached entities per Collection.

SDMCache supports several predefined search algorithms optimized for performance. Application developers can also set their own EntrySearch object in order to use a custom search algorithm.

SDMCache is an implementer of the ISDMPersistable interface, so it can be persisted with the SDMPersistence component (see more at the section about the SDMPersistence component of Android OData SDK).

SDMCache validates the incoming SDMOData documents by matching their references to each other. A single SDMCache object can store only one set of documents (one Service Document with one related Schema with any number of related Entries).

The SDMOData document created by the SDMParser component automatically sets the required references to the related objects. All these references are automatically maintained during persisting or loading the SDMCache using SDMPersistence.

SDMCache depends on the SDMOData specific interfaces of SDMParser, but does not depend on the real implementation of SDMParser.

SDMPersistence

The Persistence component is responsible for storing application specific objects and raw data in the device's physical storage.

List of Features

- Storing objects and raw data on the physical storage of the device
- Accessing objects and raw data stored on the physical storage of the device
- Encrypting and storing objects and raw data on the physical storage of the device using the secret key provided by the application developer
- Generating initial secret key
- Accessing and decrypting objects and raw data stored encrypted on the physical storage of the device using the secret key provided by the application developer

SDMPersistence Public APIs

```
ISDMPersistence

    void clear();
    void storeObject(String key, ISDMPersistable object)
    <T extends ISDMPersistable> boolean loadObject(String key, T
object)
    void storeRawData(String key, byte[] data)
    byte[] loadRawData(String key)
    void storeDataStream(String key, InputStream stream)
    InputStream loadDataStream(String key)
    boolean loadSDMCache(ISDMCache cache)
    void storeSDMCache(ISDMCache cache)
    boolean removeData(String key)
    boolean removeCache()
    boolean isDataPersisted(String key)
    void setEncryptionKey(byte[] secretKey, String
secretKeyAlgorithm) throws SDMPersistenceException
```

Technical Details

SDMPersistence preferences:

SDMPersistence can persist data in secure and non-secure mode, based on the value of preference PERSISTENCE_SECUREMODE_BOOLEAN. In secure mode, all data stored by SDMPersistence will be encrypted. Encryption is done using the secret key that is passed to SDMPersistence during initialization or by using the `setEncryptionKey` API. If the Secret Key object is null and the secure mode is turned on for SDMPersistence, SDMPersistenceException will be thrown during runtime. PERSISTENCE_SECUREMODE_BOOLEAN is by default true.

A secret key can be generated with the help of the static API of the `SDMPersistence` class. Use the `SDMPersistence.generateSecretKey (String secretKeyAlgorithm)` API. The API returns a generated secret key with the given algorithm in a byte array format.

Important: if PERSISTENCE_SECUREMODE_BOOLEAN preference is changed during runtime, all previously stored data will be deleted without any notification. It depends on the application whether it asks for confirmation from the user before changing the value of this preference.

SDMPersistence by default stores data in the application's cache folder in the file system on the physical storage of the device. Stored data is not accessible for any other applications, but can be wiped out by the user outside of the application using the device's application settings. The default folder to store persisted data can be changed by changing the value of PERSISTENCE_DEFAULTFOLDER_PATH_STRING preference. If the PERSISTENCE_DEFAULTFOLDER_PATH_STRING preference is changed during runtime, all previously stored data will be automatically moved to the new folder.

SDMPersistence implementation guarantees the proper concurrent file handling as long as there are no other non SDMPersistence objects trying to access the persisted data.

```
ISDMPersistable:
```

All the objects that are to be persisted with SDMPersistence need to implement the ISDMPersistable interface. All valid implementations of ISDMPersistable must implement the declared read and write methods of the interface and must have a public no-arg constructor. SDMOData objects provided by the Android OData SDK are valid implementations of ISDMPersistable.

SDMConnectivity

The SDMConnectivity layer hides the complexity of network communication and provides easy to use APIs to the applications.

List of Features

- Provides interfaces for request handling
- Handles the requests asynchronously
- Can handle the requests by multiple number of threads (configurable)

SDMConnectivity Public APIs

Note: The SUP APIs and their descriptions are available after the installation of Sybase Unwired Platform at the following location within your installation folder: `...\UnwiredPlatform\ClientAPI\apidoc`.

The `SDMRequestManager` class implements the ISDMRequestManager interface, which provides the following methods:

```
ISDMRequestManager

    void makeRequest(final ISDMRequest aRequest);
    void makeRequest(final ISDMBundleRequest aRequest);
    ConnectivityParameters getConnectivityParameter();
    int getQueueSize();
```

```
    Vector getAllRequests();
 Object getRequest();
 void setMainHandlerClassName(final String classname);
 void terminate();
 boolean hasRequests();
 void sendOnSuccess(final ISDMNetListener listener, final
ISDMRequest request, finral HttpResponse response);
```

The number of working threads in the RequestManager class is configurable via the constructor. The number of threads is maximized by the connectivity layer because of performance related issues. If the client initializes the layer with more than the allowed threads, the implementation of the connectivity layer will decrease the thread number to the max allowed number (4).

Methods defined by the ISDMConnectivityParameters interface:

```
SDMConnectivityParameters

    void setUserName(final String aUserName);
    String getUserName();
    void setUserPassword(final String aPassword);
    String getUserPassword();
    void setBaseUrl(final String url);
    String getBaseUrl();

        String getLanguage();
    void setLanguage(final String language);
    void setServerCertificate(Certificate certificate) throws
KeyStoreException;
    final TrustManager[] getTrustManagers();
```

Sending requests with the connectivity layer consists of the following steps:

1. Create the RequestManager class and initialize it with the required parameters.
2. Create the request object. This can be done in the following ways:
   - Implement the ISDMRequest interface.
   - Extend the BaseRequest class, which is the base implementation of the ISDMRequest interface.
   - When the requests' execution order is important, implement the ISDMBundleRequest, add the ISDMRequest instances into it, then pass this bundle to the request manager. Both of them are provided by the connectivity layer.
3. Use the request / request bundle object when making a request to the RequestManager.

SDMBundleRequest is a special set of SDMRequest objects. It provides serial processing of the requests when the SDMRequestManager is in multithreaded mode. Because the single SDMRequest objects are processed by multiple threads, the timing of the responses are not consistent. With SDMBundleRequest, one thread processes the bundled requests, guaranteeing that the responses are arriving in the same order as the requests are added to the bundle.

Example

```
//create and fill parameters for Connectivity library
SDMConnectivityParameters params = new SDMConnectivityParameters();
params.setUserName("test");
params.setUserPassword("testpwd");
mLogger = (ISDMLogger) new SDMLogger();
mPreferences = new SDMPreferences(getApplicationContext(), mLogger);
//create the RequestManager
mRequestManager = new SDMRequestManager(mLogger, mPreferences,
params, 2);
ISDMRequest testRequest = new SDMBaseRequest();
testRequest.setRequestUrl("http://test.de:8080/testpath");
testRequest.setRequestMethod(ISDMRequest.REQUEST_METHOD_GET);
testRequest.setPriority(ISDMRequest.PRIORITY_NORMAL);
//add the request to the connectivity layer
mRequestManager.makeRequest(testRequest);
```

Technical Details
The tasks of the connectivity library have been divided into three main categories:

• Manage the request queues
• Manage the reading writing to the input/output streams
• Manage the platform specific connection creation

The Connectivity component always performs the requests in asynchronous mode. The application's role is to handle the requests in sync mode. The component is able to perform HTTP and HTTPS requests, which you can use for developing and testing purposes, but the default is SUP Request. The threads in the connectivity library are responsible for taking the requests from the queue (FIFO - First in first out - algorithm) and performing the requests. The number of working threads in the connection pool can be configured in the connectivity layer. The queue is handled by the SDMRequestManager, and the working threads take the requests from this queue. Applications are interacting only with the SDMRequestManager class; the other components of the connectivity library are not visible to them. The network component consists of three main parts:

• SDMRequestManager: responsible for queuing the requests, managing the threads and keeping the connection with applications
• AbstractConnectionHandler: responsible for performing the request
• ConnectionFactory: responsible for creating and managing platform dependent connections to the server

An application can have more than one SDMRequestManager instances, for example, when connecting to two different servers at the same time. To support this scenario, SDMRequestManager handles ConnectionHandler as a plugin. This kind of plugin needs to implement the ISDMConnectionHandler and implement a constructor taking three parameters: SDMRequestManager, ISDMLogger implementation and ISDMPreferences implementation.

The class name with package is set by
SDMRequestManager.setMainHandlerClassName(String), or in
SDMPreferences by the
ISDMPreferences.SDM_CONNECTIVITY_HANDLER_CLASS_NAME preference key.
The default plugin is "com.sybase.mobile.lib.client.IMOConnectionHandler", which handles
connections through SUP.

There is built-in support for setting the timeout for the socket connection: the application can
use the SDMPreferences object to modify the value, using the following keys:

- ISDMPreferences.SDM_CONNECTIVITY_CONNTIMEOUT for connection timeout,
  and
- ISDMPreferences.SDM_CONNECTIVITY_SCONNTIMEOUT for socket connection
  timeout.

SDMRequest Object
An SDMRequest object wraps all the information needed by the connectivity library to be
able to perform the requests. The connectivity library interacts with the request object to query
the necessary information about the headers, the post data, and so on. The connectivity layer
also uses the request object to notify the application about the result of the request by using the
ISDMNetListener interface. The connectivity component provides an interface called
ISDMRequest and a base implementation of it, called SDMBaseRequest. The applications
have to extend this interface when creating new application specific requests.

```
ISDMRequest

o        void setRequestUrl(String url);
o        String getRequestUrl();
o        void setRequestMethod(final int reqType);
o        int getRequestMethod();
o        byte[] getData();
o        Hashtable getHeaders();
o        void setPriority(final int value);
o        int getPriority();
o        boolean useCookies();
o        ISDMNetListener getListener();
o        void setListener(ISDMNetListener listener);
```

The connectivity layer notifies the client about the result of a request by the ISDMNetListener
interface. Usage of this feature is not mandatory, but it is recommended to be able to handle
incidental errors. Methods available in the ISDMNetListener interface:

```
ISDMNetListener

    void onSuccess(ISDMRequest  aRequest, HttpResponse aResponse);
    void onError(ISDMRequest aRequest, HttpResponse aResponse,
SDMRequestStateElement        aRequestStateElement);
```

The role of the SDMRequestStateElement object used by the connectivity library is to
provide the application with more detail on the occurred error. Methods available in
ISDMRequestStateElement interface:

```
ISDMRequestStateElement

    int getHttpStatusCode();
    int getErrorCode();
    Exception getException();    void setHttpStatusCode(final int
httpStatus);
    void setErrorCode(final int code);
    void setException(final Exception aException);
```

```
Example

public void onSuccess(ISDMRequest  aRequest, SDMHttpResponse
aResponse) {
    System.out.println("Http response status code:" +
aResponse.getStatusCode());
    System.out.println("Cookie string:" +
aResponse.getCookieString());
    byte[] content = aResponse.getContent();
    String response = new String(content);
    System.out.println("Received content:" + response);
    //get the headers
    Hashtable headers = aResponse.getHeaders();
}
```

SDMConfiguration

Each low level API has its own defaults/constants set in the SDMCommon library. Default values of preferences can be found in the SDMConstants class.

List of Features

- Providing modifiable preferences for SDMComponent libraries
- Persisting modified values of preferences of SDMComponent libraries
- Validating preferences values of SDMComponent libraries
- Providing API for resetting the preferences of SDMComponent libraries to their default values
- Providing API for creating and handling custom preferences
- Persisting the values of custom preferences
- Notifying subscribed listeners in case of any change in preferences

SDMConfiguration Public APIs

```
ISDMPreferences

    public void setIntPreference(String key, int value)
    public void setLongPreference(String key, long value)
    public void setFloatPreference(String key, float value)
    public void setBooleanPreference(String key, boolean value)
    public void setStringPreference(String key, String value)

    public void resetPreference(String key)
```

```
    public boolean containsPreference(String key)

    public Float getFloatPreference(String key)
    public Integer getIntPreference(String key)
    public Long getLongPreference(String key)
    public Boolean getBooleanPreference(String key)
    public String getStringPreference(String key)

    void registerPreferenceChangeListener(String
key,ISDMPreferenceChangeListener     changeListener)

    void unRegisterPreferenceChangeListener(String
key,ISDMPreferenceChangeListener     changeListener);

    public void removePreference(String key) throws
SDMPreferencesException;
```

```
ISDMPreferenceChangeListener

void onPreferenceChanged(String key,Object value)
```

Technical Details
Android offers an optimized storage for preferences called SharedPreferences (even with automatic Preference screen generation from XML). Modified and custom preferences will be automatically persisted into the default SharedPreferences of the application. Preferences must not contain any secure information. For this purpose, use SDMPersistence in secure mode or the Data Vault from SUP.

SDMComponents preferences can be reset to their default values using the `resetPreference()` method or by removing them from SharedPreferences.

Any changes to the SDMComponents preferences will be automatically validated regardless whether they are modified by using SDMPreferences or by using the default API of SharedPreferences. If you change the value of a preference to an invalid value while using the SharedPreferences API of the OS, the invalid value will automatically be removed at runtime and the preference will be set to its default value without any notification. Application developers are encouraged to use the API of ISDMPreferences so they will be notified about invalid values.

You can register a preference change listener for each preference in SDMPreferences (including custom preferences) so that you will be notified if the value of a given preference has changed.

Preference change listener notification and preference validation can only be done after the initialization of the appropriate component. It is not recommended to change the values of SDK related preferences outside runtime. For example, if you change the root folder of persisted data at runtime, the SDMPersistence component will automatically move all the persisted data. However, changing this value before the initialization of the SDMPersistence component can result in the loss of persisted data.

The OData SDK provides reusable custom Preference classes as an extension of standard Preference classes provided by the OS for handling Long, Float, and Integer preferences. These custom classes can be reused in preference XMLs or in the custom preferences screen of the application. Custom preference classes can be found in the SDMCommon Component package of the OData SDK.

Supportability

The OData SDK provides a set of features and concepts for the supportability of the applications built on top of the SDK.

Exceptions

Every component of the Library has its own root exception, named as <SDM Library component name>Exception. For instance, in Connectivity, the root exception is SDMConnectivityException. All component-specific exceptions are extending the component's root exception. Besides root exceptions, SDM components can also throw general exceptions, such as IllegalArgumentException or IllegalStateException.

SDMLogger

The library supports logging via its ISDMLogger interface and provides SDMLogger as an implementation of this interface.

List of Features

- Provides a common interface for handling log messages across the library
- Extends Android's standard logging facility, while keeps method signatures compatible
- Provides facility to store log data
- Provides filterable log retrieval by severity, tag, timestamp (from-to), process id and by correlation id

Technical Details

The interface is similar to Android's standard logging facility (android.util.Log). Logging does not support security and handling sensitive data. It is the responsibility of the applications to handle these requirements. Logging supports retrieving the log data for persistence or other purposes. SDMLogger also implements the ISDMPersistable interface to make the log data persistable. A log header can be set by the application including the following fields:

- Operating System version
- App name
- App Version
- 3rd Party product versions (for example, SQLlite)
- Hardware version
- User
- Timezone

- Language
- SUP/SAP NetWeaver Gateway URL

The SDMConnectivity sets the User, Language and SUP/SAP NetWeaver Gateway URL fields. SDMLogger stores log entries timestamped, in milliseconds granularity of the time the log method called by the application/library component. It can also clean out log messages below a certain level, or clean out the log completely. A preliminary log rotation support is built in. At every log method call, a check runs and verifies whether the number of messages reaches 10000. If the number of messages is greater or equal to this threshold, a low priority background thread is started to clean out the oldest 200 log entries.

SDMLogger provides line-level location logging with the full class name of the logging class. Location detection is done by call stack evaluation. Therefore, SDMLogger provides location parameter setting for the logging class, where the class can set the location instead of using the detection facility. Log messages are stored only above the predefined logging level, which defaults to ERROR log level.

Log priority constants:

- PERFORMANCE = 1
- VERBOSE = 2
- DEBUG = 3
- INFO = 4
- WARN = 5
- ERROR = 6
- ASSERT = 7
- FATAL = 8

Log Methods

```
public void log(final int level, final String tag, final String msg,
            final Throwable tr, final String location)
```

Parameters:

```
level  the log level
msg  The message you would like logged.
tr  An exception to log
location  The line-level location of the log source (full class name
of the class)

public static int d (String tag, String msg)
public static int d (String tag, String msg, Throwable tr)
public static int d (String tag, String msg, Throwable tr, String
location)
Sends a DEBUG log message and logs the exception.

public static int e (String tag, String msg)
public static int e (String tag, String msg, Throwable tr)
public static int e (String tag, String msg, Throwable tr, String
location)
```

```
Sends an ERROR log message and logs the exception.

public static int i (String tag, String msg)
public static int i (String tag, String msg, Throwable tr)
public static int i (String tag, String msg, Throwable tr, String
location)
Sends an INFO log message and logs the exception.

public static int v (String tag, String msg)
public static int v (String tag, String msg, Throwable tr)
public static int v (String tag, String msg, Throwable tr, String
location)
Sends a VERBOSE log message and logs the exception.

public static int w (String tag, Throwable tr)
public static int w (String tag, String msg)
public static int w (String tag, String msg, Throwable tr)
public static int w (String tag, String msg, Throwable tr, String
location)
Sends a WARN log message and logs the exception.

public static int wtf (String tag, Throwable tr)
public static int wtf (String tag, String msg)
public static int wtf (String tag, String msg, Throwable tr)
public static int wtf (String tag, String msg, Throwable tr, String
location)
What a Terrible Failure: Reports a condition that should never
happen. The error will always be logged at level ASSERT.
```

SDMLogger (ISDMLogger implementation that the Library provides) also has the following functionality:

```
public void cleanUp(final int threshold)
Deletes all log entries weaker than the 'threshold' priority.

public void terminate()
Completely clears the collected log data.

public Vector<LogEntry> getLogElements(final int threshold)
This method returns the log data, including all log data with level
'threshold' or above.

public boolean logsToAndroid()
public void logToAndroid(final boolean doIt)
These methods get and set the property which controls sending the log
output to the Android logging facility.

public boolean logsFullLocation()
public void logFullLocation(boolean logFullLocation)
These methods get and set the property which if full location should
be logged automatically based on the current stack trace.
public synchronized String toString()
Returns all log data – including the header – as String.
```

Sample:

```
Operating System version: 11
Application name: MyApp
Application version: 1.0.0
3rd-party products: -
Hardware version: Galaxy Tab
User name: DEMO
Timezone: CET-DST
Language: en
Base URL: http://www.sap.com/gateway/or/whatever

2011-06-28 14:30:23.368 WARN    SDMPreferences
com.sap.mobile.lib.sdmconfiguration.SDMPreferences.getPreference(SD
MPreferences.java:284)   Deprecated method 'getPreference' has been
called.
2011-06-28 14:30:23.468 INFO    SDMPreferences
com.sap.mobile.lib.sdmconfiguration.SDMPreferences.setStringPrefere
nce(SDMPreferences.java:244)    Preference
'SAP_APPLICATIONID_HEADER_VALUE' (String) has been changed to MyApp.
1.0.0.0
public Vector<LogEntry> getLogElementsByTag(final String aTag)
public Vector<LogEntry> getLogElementsByTimeStamp(final long start,
final long end)
public Vector<LogEntry> getLogElementsByPID(final long PID)
public Vector<LogEntry> getLogElementsByCorrelationId(final String
correlationId)
```

These methods return with a Vector of filtered log entries, filtered by TAG, timestamp (interval), process id and correlation id, respectively.

SAP Passport

For the Single Activity Trace an SAP® Passport has to be issued by the connectivity layer of the library.

The SAP Passport is transported as an HTTP header in the request. The server handles the SAP Passport to generate end-to-end Trace. The OData SDK is using JSDR SAP Passport sources integrated in the library at source level. It can be turned on or off with `ISDMPreferences.SAPPASSPORT_ENABLED` preference key. By default it is turned off.

Deploying Applications to Devices

This section describes how to deploy customized mobile applications to devices.

1. *Installing Applications on the Device without Using the Android Market*

   Connect the device to your personal computer and install applications without using the Android market.

2. *Installing Applications using a URL*

   Install applications on an Android device without using the Android market.

3. *Deploying Applications using Afaria*

Deploy Android applications using Afaria.

See also
*   *OData SDK Components and APIs* on page 59

Installing Applications on the Device without Using the Android Market

Connect the device to your personal computer and install applications without using the Android market.

Prerequisites

*   Activate the installation of programs on your device that do not originate from the Android market. Navigate to **Settings > Applications > Unknown Sources** to allow installation of these programs.

Task

1. Download the driver software and install this on your personal computer.
    Example: HTC Sync for all HTC Android Phones and HTC Smart Phones, see *http://www.htc.com/uk/help*.
2. Connect your device to the personal computer through a USB cable.
    The driver software uploads the device software and installs it on the device.
3. On the device display screen, make the selection to enable to mount the memory card.
4. Copy the .apk file to the memory card.
5. Disconnect the USB cable from the device.
6. Using the file manager on the device, access the .apk file from the memory card and follow the instructions as displayed.

Installing Applications using a URL

Install applications on an Android device without using the Android market.

Prerequisites

*   Activate the installation of programs on your device that do not originate from the Android market. Navigate to **Settings > Applications > Unknown Sources** to allow installation of these programs.
*   You must have the URL where the Android package is available as a resource.

Task

1. Enter the URL details in the device browser.
2. Follow the instructions displayed on the browser to install the application.

Deploying Applications using Afaria

Deploy Android applications using Afaria.

See the following sections in *System Administration* for details on how to perform Android provisioning and deployment.

- *System Administration > Device and Application Provisioning Overview> Provisioning with Afaria.*

CHAPTER 4   **Developing BlackBerry Applications**

Provides information about using advanced Sybase® Unwired Platform features to create applications for RIM BlackBerry devices. The audience is advanced developers who are familiar working with APIs, but who may be new to Sybase Unwired Platform.

Using Online Data Proxy, you can connect a device to an OData-based back-end system. All Online Data Proxy client libraries provide secure communication to the SUP server in addition to parsing, caching, persistence, connectivity, supportability and secure storage.

Describes requirements for developing a device application for the platform. Also included are task flows for the development options, procedures for setting up the development environment and API references.

1. *Configuring the BlackBerry Developer Environment*

   This section describes how to set up your BlackBerry development environment and provides the location of required JAR files and COD files.

2. *Creating Projects and Adding Libraries into the BlackBerry Development Environment*

   Set up the BlackBerry project and add required libraries. Use these procedures if you are developing a device application using the BlackBerry JDE or the BlackBerry Java plug-in for Eclipse.

3. *Online Data Proxy BlackBerry API JAR File Locations*

   The Online Data Proxy JAR files and dependencies are installed in the Sybase Unwired Platform installation directory. JAR files are used for compilation and COD files for runtime. Make sure the COD files are deployed to the simulator/device with the device application.

4. *Developing Applications in the BlackBerry Development Environment*

   To learn more about the BlackBerry JDE, BlackBerry Java plug-in for Eclipse, or RIM BlackBerry APIs, go to the BlackBerry Java application development Web site at http://na.blackberry.com/eng/developers/javaappdev

5. *OData SDK Components and APIs*

   The OData SDK for BlackBerry provides the means to easily build an application which relies on the OData protocol and its additions made by SAP.

6. *Deploying Applications to Devices*

   This section describes how to deploy customized mobile applications to devices.

Configuring the BlackBerry Developer Environment

This section describes how to set up your BlackBerry development environment and provides the location of required JAR files and COD files.

See also
• *Creating Projects and Adding Libraries into the BlackBerry Development Environment* on page 81

Installing the BlackBerry Development Environment

Download and install either the BlackBerry JDE or the BlackBerry Java plug-in for Eclipse (eJDE).

For information on transitioning from the BlackBerry JDE to the eJDE, view the video at the Research In Motion Developer Video Library Web site: *http://supportforums.blackberry.com/t5/Java-Development/tkb-p/java_dev%40tkb?labels=video*

Installing the BlackBerry Java Plug-in for Eclipse
The BlackBerry Java Plug-in for Eclipse is an IDE for developing BlackBerry applications.

Prerequisites
You must have a BlackBerry developer account to download the BlackBerry Java Plug-in for Eclipse. You may be required to register if you do not already have an account.

Task

1. Double-click the setup application file.
2. Click **Run**.
3. On the Introduction page, click **Next**.
4. Accept the terms of the license agreement and click **Next**.
5. Create and select a new, empty folder for the installation directory and click **Next**.
6. Review the information on the Pre-installation Summary screen and click **Install**.
7. Click **Done**.
   The installation is complete.
8. (Optional). Copy the `plugin` and `features` folders from the installation to
   `<UnwiredPlatform_InstallDir>\UnwiredPlatform`
   `\Unwired_WorkSpace\Eclipse\sybase_workspace\mobile\eclipse`.
   This step ensures that Sybase Unwired WorkSpace contains the BlackBerry Java Plug-in for Eclipse, and that users can directly use it from Sybase Unwired WorkSpace instead of opening another instance of Eclipse to work with the BlackBerry Java Plug-in for Eclipse.

Downloading the BlackBerry JDE
To generate and distribute BlackBerry device applications, download BlackBerry JDE and its prerequisites from the BlackBerry Web site.

Prerequisites
You must have a BlackBerry developer account to download the BlackBerry JDE. You may be required to register if you do not already have an account. Before you download the JDE, ensure the 32-bit JDK has already been installed, even for 64-bit operating systems; otherwise, MDS will not start.

Task
Go to the BlackBerry Web site at *http://us.blackberry.com/developers/javaappdev/ javadevenv.jsp* to download and install the BlackBerry JDE.
The MDS-CS simulator is installed with the BlackBerry JDE.

Creating Projects and Adding Libraries into the BlackBerry Development Environment

Set up the BlackBerry project and add required libraries. Use these procedures if you are developing a device application using the BlackBerry JDE or the BlackBerry Java plug-in for Eclipse.

See also

Adding Required .jar and .cod Files
Add the following Online Data Proxy .jar file references to the BlackBerry project's Java build path.

Copy the following OData .jar files:

- `sdmcache-1.0.0-preverified.jar` – from `<UnwiredPlatform_InstallDir>\MobileSDK\OData\BB\libraries\` for the BlackBerry client.
- `sdmcommon-1.0.0-preverified.jar` – from `<UnwiredPlatform_InstallDir>\MobileSDK\OData\BB\libraries\` for the BlackBerry client.

- `sdmconfiguration-1.0.0-preverified.jar` – from
  `<UnwiredPlatform_InstallDir>\MobileSDK\OData\BB\libraries\`
  for the BlackBerry client.
- `sdmconnectivity-1.0.0-preverified.jar` – from
  `<UnwiredPlatform_InstallDir>\MobileSDK\OData\BB\libraries\`
  for the BlackBerry client.
- `sdmparser-1.0.0- preverified.jar` – from
  `<UnwiredPlatform_InstallDir>\MobileSDK\OData\BB\libraries\`
  for the BlackBerry client.
- `sdmpersistence-1.0.0-preverified.jar` – from
  `<UnwiredPlatform_InstallDir>\MobileSDK\OData\BB\libraries\`
  for the BlackBerry client.
- `sdmsupportability-1.0.0-preverified.jar` – from
  `<UnwiredPlatform_InstallDir>\MobileSDK\OData\BB\libraries\`
  for the BlackBerry client.

Copy the following ODP .jar files:

- `CommonClientLib.jar` – from `<UnwiredPlatform_InstallDir>`
  `\MobileSDK\OData\BB\libraries\` for the BlackBerry client.
- `MessagingClientLib.jar` – from `<UnwiredPlatform_InstallDir>`
  `\MobileSDK\OData\BB\libraries\` for the BlackBerry client.
- `MocaClientLib.jar` – from `<UnwiredPlatform_InstallDir>`
  `\MobileSDK\OData\BB\libraries\` for the BlackBerry client.
- `sup_json.jar` – from `<UnwiredPlatform_InstallDir>\MobileSDK`
  `\OData\BB\libraries\` for the BlackBerry client.
- `SUPProxyClient-1.0.0.jar` – from `<UnwiredPlatform_InstallDir>`
  `\MobileSDK\OData\BB\libraries\` for the BlackBerry client.

Consuming Java .JAR files for BlackBerry Projects

Add the .jar and .cod files to your BlackBerry project.

Using this procedure, the Java definitions are available in Eclipse in order to find the third-party classes when compiling your project's source code. After compilation you will have one `.cod` file containing the application and the libraries together.

1. Download the library to your host development system.
2. Create a new folder, named `libs`, in your Eclipse/BlackBerry project.
3. Right click `libs` and choose `Import -> General -> File System`, then click **Next**.
4. Browse the file system to find the library's parent directory (where you downloaded it).

5. Click **OK**, then click the directory name (not the checkbox) in the left pane and check the relevant JAR in the right pane. This puts the library into your project (physically).

6. Right click on your project, choose `Build Path -> Configure Build Path`, then click the Libraries tab, then click **Add JARs...**

7. Navigate to your new JAR in the libs directory and add it.

8. Click on the Order and Export tab. After you added the libraries they should be listed. Check all the libraries. This way the libraries will be compiled together with the application and packaged into one `.cod` file.

> **Note:** The following .jar files should not be marked as 'Exported' in the build path:
> - `CommonClientLib.jar`
> - `MessagingClientLib.jar`
> - `MocaClientLib.jar`
> - `sup_json.jar`
>
> The .cod files corresponding to these .jar files have to be deployed on the device while installing the application.

Online Data Proxy BlackBerry API JAR File Locations

The Online Data Proxy JAR files and dependencies are installed in the Sybase Unwired Platform installation directory. JAR files are used for compilation and COD files for runtime. Make sure the COD files are deployed to the simulator/device with the device application.

The contents and location of the .jar and .cod files:

- `<UnwiredPlatform_InstallDir>\MobileSDK\OData\BB\libraries\`

The API references can be extracted from the following zip files:

- `<UnwiredPlatform_InstallDir>\MobileSDK\OData\BB\docs`
  `\SUPProxyClient-2.1.1-alpha-2-docs.zip`
- `<UnwiredPlatform_InstallDir>\MobileSDK\OData\BB\docs`
  `\BBODataSDK-1.1.0-doc.zip`

See also

• *Creating Projects and Adding Libraries into the BlackBerry Development Environment* on page 81

Developing Applications in the BlackBerry Development Environment

To learn more about the BlackBerry JDE, BlackBerry Java plug-in for Eclipse, or RIM BlackBerry APIs, go to the BlackBerry Java application development Web site at *http://na.blackberry.com/eng/developers/javaappdev*

To enable mobile devices to install applications and securely communicate in the enterprise landscape, there are different ways in which you can onboard your mobile device.

This section provides a quick reference to APIs used for developing Android Applications. For a comprehensive list of API references, extract the contents from the following zip files:

- `<UnwiredPlatform_InstallDir>\MobileSDK\OData\BB\docs \SUPProxyClient-2.1.1-alpha-2-docs.zip`
- `<UnwiredPlatform_InstallDir>\MobileSDK\OData\BB\docs \BBODataSDK-1.1.0-doc.zip`

See also
- *OData SDK Components and APIs* on page 95

Initializing an Application

Before you use any of the other BlackBerry ODP APIs, you have to first initialize an application.

Syntax
```
public static void initialize(String appID) throws
com.sybase.mo.MessagingClientException
```

Parameters

- **appID** – Name of the registered application.

Examples

- **Initialize an application** –
  ```
  UserManager.initialize(applicationID);
  ```

Manually Registering an Application

Manually register an application by using the user name and activation code of the application registered through the Sybase Control Center.

Syntax

Synchronous Registration

```
public static void registerUser(String username, String
activationCode) throws UserManagerException,
com.sybase.mo.MessagingClientException
```

Asynchronous Registration

```
public static void asyncRegisterUser(String username, String
activationCode) throws UserManagerException
```

Parameters

- **username** – User name specified in SCC
- **activationCode** – Activation Code specified in SCC

Examples

- **Register the application manually** –
  ```
  UserManager.registerUser(userName, activationCode);
  ```

Automatically Registering an Application using SSO2 Cookie

Registering an application automatically using an SSO2 Token Cookie. This token is fetched from a ticket issuing system and verified by the server.

Syntax

Synchronous Registration

```
public static void registerUser(String username, String
securityConfig, String password, String vaultPassword) throws
UserManagerException, com.sybase.mo.MessagingClientException,
SUPDataVaultException
```

Asynchronous Registration

```
public static void asyncRegisterUser(String username, String
securityConfig, String password, String vaultPassword) throws
UserManagerException
```

Parameters

- **username** – User name of the ticket issuing system.

- **securityConfig –** Security configuration of the registered application provided by the administrator in the Sybase Control Center.
- **password –** Password used to authenticate the user.
- **vaultPassword –** Password required to unlock the data vault .

Examples

- **Registering a user using SSO2 Cookie –**
```
UserManager.registerUser(TISUsername, securityConfig,
TISPassword, vaultPassword);
```

Automatically Registering an Application using HTTP Authentication Provider

Registering an application automatically using the HTTP Authentication Provider.

Syntax
```
public static void registerUser (String username, String
securityConfig, String password, String vaultPassword) throws
UserNamagerException, com.sybase.mo.MessagingClientException,
```

Parameters

- **username –** Valid user name
- **securityConfig –** Security configuration of the registered application provided by the administrator in the Sybase Control Center.
- **password –** Password to identify the back-end system.
- **vaultPassword –** Password required to unlock the data vault .

Examples

- **Registering a user using HTTP Authentication Provider. –**
```
UserManager.registerUser(Username, securityConfig, password,
vaultPassword)
```

Automatically Registering an Application using X.509 Certificate

Registering an application automatically using an X.509 Certificate. This certificate is fetched from a Certificate Authority and verified by the server.

Syntax
Synchronous Registration
```
public static void registerUser(String username, String
securityConfig, String password, String vaultPassword) throws
```

```
UserManagerException, com.sybase.mo.MessagingClientException,
SUPDataVaultException
```

Asynchronous Registration

```
public static void asyncRegisterUser(String username, String
securityConfig, String password, String vaultPassword) throws
UserManagerException
```

Parameters

- **username** – Valid user name
- **securityConfig** – Security configuration of the registered application provided by the administrator in the Sybase Control Center.
- **password** – Contains the Base64 encoded string of the certificate library.
- **vaultPassword** – Password required to unlock the data vault .

Examples

- **Registering a user using X.509 Certificate –**

```
UserManager.registerUser(backendUsername, securityConfig,
CertificateStore.getSignedCertificateFromStore(certInfo),
vaultPassword);
```

Storing the Application Credentials Securely

Post user registeration, if you want the user credentials to be managed by SDK, you can provide a data vault password to securely store the data.

Syntax

```
public static void setAppCredentials(String username, String
securityConfig, String password, String vaultPassword) throws
SUPDataVaultException, UserManagerException
```

Parameters

- **username** – Valid user name to be stored.
- **securityConfig** – Security configuration of the registered application to be stored.
- **password** – If using certificates, this corresponds to the Base64 encoded string of the certificate library. If using SSO2 cookie, this corresponds to the passowrd of the ticket issuing system.
- **vaultPassword** – Password of the secure store provided by SDK.

Examples

- **Using data vault to store data securely –**

```
UserManager.setAppCredentials(Username, securityConfig, Password,
vaultPassword)
```

Checking for Registered Users

Check if a device user is registered or not.

Syntax

```
public static boolean isRegistered() throws UserManagerException
```

Returns

If the user is registered, the function returns 'true'. If the user is not registered, the function returns 'false'.

Examples

- **Check if the user is registered –**
  ```
  boolean UserManager.isRegistered();
  ```

Deleting Users

Deregister an application user when you do not need the application on the device.

When you invoke this API, the user, along with all the client data, is deleted.

Syntax

```
public static void deleteUser() throws UserManagerException,
com.sybase.mo.MessagingClientException
```

Examples

- **Delete the user –**
  ```
  UserManager.deleteUser();
  ```

Enabling Online Push

To consume push messages, the application registers a listener object.The client SDK notifies this listener object whenever there is a push message from the server. The listener object should implement the ISDMNetListener interface.

Syntax

```
public static void
setPushListener(com.sap.mobile.lib.sdmconnectivity.ISDBNetListener
pushListerner)
```

Parameters

- **pushListener –** Object that implements ISDMNetListener interface.

Examples

- **Listener Object –**
  ```
  UserManager.setPushListener(listenerObjectFromApp);
  ```
- **Implementation of APIs in the Listener Object –**
  ```
  ISDMNetListener.onError(ISDMRequest, IHttpResponse,
  ISDMRequestStateElement)
  ISDMNetListener.onSuccess(ISDMRequest, IHttpResponse,
  ISDMRequestStateElement)
  ```

Getting Application End-point

Retrieve the application end-point that corresponds to the gateway service document.

Syntax

```
public static String getApplicationEndPoint() throws
com.sybase.mo.MessagingClientException, UserManagerException,
SUPDataVaultException
```

Returns

If successful, this function returns the application end-point.

Examples

- **Retrieving application end-point –**
  ```
  AppSettings.getApplicationEndPoint()
  ```

Getting Push End-point

Retrieve the push end-point that corresponds to the delivery address that the application uses in the subscription request for notifications.

Syntax

```
public static String getPushEndPoint() throws
com.sybase.mo.MessagingClientException, UserManagerException
```

Returns

If successful, this function returns the push end-point.

Examples

- **Retrieve the push end-point –**
  ```
  AppSettings.getPushEndPoint()
  ```

Getting Server Details

Retrieve the SUP server host name.

Syntax

```
public static String getServer() throws
com.sybase.mo.MessagingClientException
```

Returns

Returns the server name as a string.

Examples

- **Retrieve the server details –**
  ```
  AppSettings.getServer();
  ```

Getting Port Number

Retrieve the port number provisioned in the client repository.

Syntax

```
public static int getPortNumber() throws
com.sybase.mo.MessagingClientException
```

Returns

Returns the port number as an integer.

Examples

- **Retrieve the port number –**
  ```
  AppSettings.getPortNumber();
  ```

Getting FarmID

Retrieve the Farm ID of the SUP Server.

Syntax

```
public static String getFarmID() throws
com.sybase.mo.MessagingClientException
```

Returns

Returns the Farm ID as a string.

Examples

- **Retrieve the Farm ID** –
  ```
  AppSettings.getFarmID();
  ```

Checking the Provisioning Status of the Public Key

Check if the public key is provisioned on the client.

Syntax

```
public static boolean IsSUPKeyProvisioned() throws
com.sybase.mo.MessagingClientException
```

Returns

If the key is provisioned, the value 'true' is returned, else 'false'.

Examples

- **Check the provisioning status of the public key** –
  ```
  AppSettings.IsSUPKeyProvisioned();
  ```

Provisioning Certificates using Afaria

Certificates can be provisioned for BlackBerry devices using Afaria.

Syntax

```
public static String getSignedCertificateFromAfaria(String CN,
String challengeCode) throws
com.sybase.persistence.SSOCertManagerException, IOException
```

Parameters

- **CN** – A character-type column name, variable, or constant expression of char, varchar, nchar, nvarchar, or unichar type. Corresponds to the certificate name.
- **challengeCode** – Another character-type column name, variable, or constant expression of char, varchar, nchar, nvarchar, or unichar type.

Returns

Returns the certificate as a base64 encoded string.

Getting Application Seed Data from Afaria

Get the application seed data from Afaria.

Syntax

```
public static Hashtable getSettingsFromAfaria() throws
com.sybase.afaria.SeedDataAPI.SeedDataAPIException, IOException
```

Returns

Returns a hastable containing the settings.

Clearing the Server Verification Key

For a device to switch connection between SUP servers, this API is invoked before registering a new user. This ensures that the server public keys are removed from the SUP client SDK which enables connectivity to the new SUP Server.

Syntax

```
public static void clearServerVerificationKey() throws
UserManagerException, com.sybase.mo.MessagingClientException
```

Examples

*   **Clear the server verification key –**

    ```
    UserManager.clearServerVerificationKey();
    ```

Data Vault API References

The data vault is a secure storage area provided by the SUP 2.1 SDK client libraries to store sensitive data such as usernames, passwords, authentication certificates within the application. Access to the data vault is protected by two levels of passwords and unique salts.

Creating a Vault

Creates an instance of a vault with a set of attributes.

Syntax

```
public static SUPDataVault createVault(String sDataVaultID, String
sPassword, String sSalt) throws SUPDataVaultException
```

Parameters

*   **sDataVaultID –** The vault name.

- **sPassword** – The vault password
- **sSalt** – The salt password

Returns

If successful, creates an instance of SUPDataVault.

Opening an Existing Vault

Returns the SUPDataVault singleton instance, tied to a particular vault. If the vault does not exist or has been deleted, this method throws an exception.

There is a singleton instance per data vault ID.

Syntax

```
public static SUPDataVault getVault(String sDataVaultID) throws
SUPDataVaultException
```

Parameters

- **sDataVaultID** – The vault name.

Returns

If successful, returns a singleton instance of the vault..

Deleting a Vault

Delete the storage for this instance from the persistent storage. Once a vault is deleted, all current instance references become invalid.

Syntax

```
public static void deleteVault (String sVaultId) throws
SUPDataVaultException
```

Parameters

- **sVaultId** – The vault name.

Locking a Vault

Lock a vault to avoid it from being used. If the vault is locked, this API will have no effect.

Syntax

```
public void lock() throws SUPDataVaultException
```

Unlocking a Vault

Unlock a vault for use by an application.

Syntax

```
public void unlock(String sPassword, String sSalt) throws
SUPDataVaultException
```

Parameters

- **sPassword** – The vault password.
- **sSalt** – The vault's salt password.

Setting a Binary Value

Store a value in the vault. To remove a value, provide 'null' as the second parameter.

Syntax

```
public void setValue(String sName, byte[] abValue) throws
SUPDataVaultException
```

Parameters

- **sName** – The key in which you store the data.
- **abValue** – The value you want to store.

Retrieving a Binary Value

Retrieve a value set from the vault.

Syntax

```
public byte[] getValue (String sName) throws SUPDataVaultException
```

Parameters

- **sName** – The key that contains the data you want to retrieve

Returns

If successful, returns the value stored in the key.

Setting the Retry Limit Value for a Vault

Set the maximum number of consecutive failed attempts to unlock the vault.

Syntax

```
public void setRetryLimit(int iLimit) throws SUPDataVaultException
```

Parameters

- **iLimit** – Maximum failed attempts that is permitted to unlock the vault.

Setting the Lock Timeout Value for a Vault

Set the time until which the vault remains in an unlocked state. Once this time is lapsed, the vault reverts to the locked state.

Syntax

```
public void setLockTimeout(int iTimeoutSeconds) throws
SUPDataVaultException
```

Parameters

- **iTimeoutSeconds** – Time in seconds for which the vault is unlocked.

OData SDK Components and APIs

The OData SDK for BlackBerry provides the means to easily build an application which relies on the OData protocol and its additions made by SAP.

Prerequisites
Download the Eclipse IDE and the BlackBerry java plug-in for Eclipse to be able to develop on BlackBerry platform.

OData SDK - BlackBerry
The full list of APIs and their descriptions are available after the installation of Sybase Unwired Platform at the following location within your installation folder: `...
\UnwiredPlatform\MobileSDK\OData\BlackBerry\docs`.

The following figure shows the main components of the OData SDK on BlackBerry.
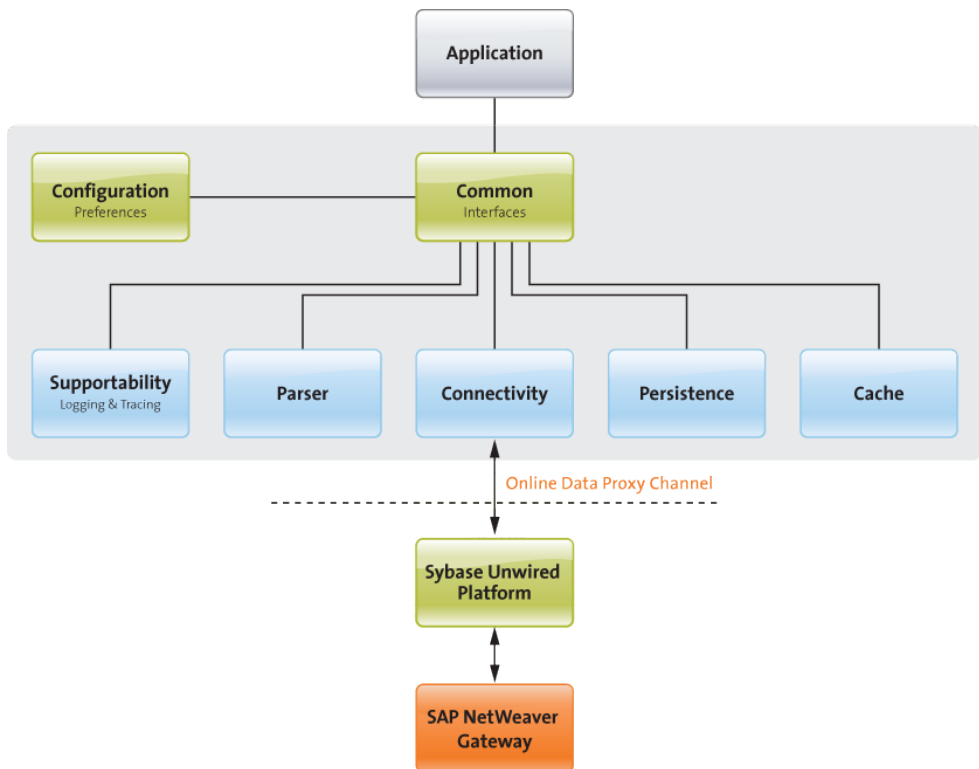
SDMCommon
To build an application on the OData SDK, you must first import the SDMCommon component that contains interfaces and configuration for the components. None of the components have dependency on each other, but all of them depend on the SDMCommon component, and all of them have references to interfaces of other components (held by SDMCommon).

Component Replacements
In your own application, you can replace the implementation behind an interface of a BlackBerry OData SDK component. For example, if you want to add a new functionality to SDMCache, but keep everything else unchanged (for example, the way it is persisted by SDMPersistence) you can implement your own solution. The new cache can be either a new implementation, or a descendant of SDMCache, as long as it implements the ISDMCache interface from SDMCommon.

See also
*   *Developing Applications in the BlackBerry Development Environment* on page 84
*   *Deploying Applications to Devices* on page 113

SDMParser

The parser (SDMParser class) is the core component of the package, it is responsible for processing XMLs. The actual parsing is done by the standard java SAX parser provided by the BlackBerry platform.

Parsing is generic in the sense that an arbitrary (well-formed) XML can be processed, and the information content is returned without any loss:

```
/**
 * Parses the stream source of an XML and converts it to a Java Object
containing all
 * the information that were contained by the source XML.
 *
 * @param xml
 *            A byte array that holds a syntactically valid XML.
 * @return ISDMParserDocument The Object representation of the parsed
XML.
 * @throws SDMParserException
 *            If the XML source is invalid.
 * @throws IllegalArgumentException
 *            If the argument is null.
 */
public abstract ISDMParserDocument parseXML(byte[] xml) throws
SDMParserException, IllegalArgumentException;
```

The ISDMParserDocument interface provides access to all the data stored in the XML. The API user constructs the path inside the XML to the given data (attribute or text value), then the following methods return their value:

```
/**
* Returns the string value of the sub-document contained by this
object and accessible via the
* element names provided by the 'route' argument.
*
* @param route
*            "/" separated route that leads to the object route must
contain indexes as well.
*            Route must end with the index number which uniquely
identifies an XML element.
*            route must start with "/"
* @return The string value of the XML element on the route. It
returns null if route
*         does not identify a unique element.
*/
public abstract String getValue(String route);

/**
* Returns the string value of the XML attribute of the object
accessible via the element names
* provided by the 'route' argument.
*
* @param route
```

```
*              "/" separated route that leads to the object route must
contain indexes as well.
*              Route must end with the index number which uniquely
identifies an XML element and
*              after the element the attribute local name (field name)
must be appended with
*              slash. Example route: "/element1/1/element2/5/element3/2/
attributename"
* @param namespaceURI
*              The Namespace URI of the attribute, or the empty String if
the attribute local
*              name has no Namespace URI.
*
* @return The string value of the given attribute on the given route
*/
public abstract String getAttribute(String route, String
namespaceURI) throws IllegalArgumentException;
```

However, for applications that communicate with the OData Protocol and that are working with OData objects, it is more suitable to use parser methods that provide OData objects (hierarchies).

There are specific parser methods for the document types that come in the OData Protocol responses. These are the service document, metadata document, open search description, error message, atom feed and entry:

```
/**
* Parses the SDMOData Service Document XML and converts it to an
appropriate Java Object.
*
* @param xml
*              The byte array that holds SDMOData Service Document XML
* @return ISDMODataServiceDocument The Object representation of
SDMOData Service Document.
* @throws SDMParserException
*              If the XML source is invalid.
* @throws IllegalArgumentException
*              If the argument is null.
*/
public abstract ISDMODataServiceDocument
parseSDMODataServiceDocumentXML(byte[] xml) throws
SDMParserException, IllegalArgumentException;

/**
* Parses the SDMOData metadata XML and converts it to an appropriate
Java Object.
*
* @param xml
*              The byte array that holds SDMOData Schema XML
* @return ISDMODataSchema The Object representation of the SDMOData
Schema.
* @throws SDMParserException
*              If the XML source is invalid.
* @throws IllegalArgumentException
*              If the argument is null.
```

```
*/
public abstract ISDMODataMetadata parseSDMODataMetadataXML(byte[]
xml, ISDMODataServiceDocument svDoc) throws SDMParserException,
IllegalArgumentException;
```

The service document XML has to be processed before the metadata, because metadata parsing needs the service document object.

```
/**
* Parses the SDMOData Open Search Description XML from stream and
converts it to an *appropriate Java Object.
*
* @param xml
*            The byte array that holds the SDMOData Open Search
Description XML.
* @return ISDMODataOpenSearchDescription The Object representation
of the SDMOData Open Search
*        Description.
* @throws SDMParserException
*            If the XML source is invalid.
* @throws IllegalArgumentException
*            If the argument is null.
*/
public abstract ISDMODataOpenSearchDescription
parseSDMODataOpenSearchDescriptionXML(byte[] xml) throws
SDMParserException, IllegalArgumentException;

/**
* Parses the SDMOData Error XML from stream and converts it to an
appropriate Java Object.
*
* @param xml
*            The byte array that holds the SDMOData Error XML.
* @return ISDMODataError The Object representation of the SDMOData
Error.
* @throws SDMParserException
*            If the XML source is invalid.
* @throws IllegalArgumentException
*            If the argument is null.
*/
public abstract ISDMODataError parseSDMODataErrorXML(byte[] xml)
throws SDMParserException;
```

There are also dedicated methods for feed and parsing, and the parser is also able to process entry XMLs. Both of them need the entity set object representing the collection container of the entry, so the parser has access to the metadata of the entry, which is needed for proper data parsing.

```
/**
* Parses OData XML structures from stream that represent either a
single SDMOData Entry or a
* feed of several SDMOData entries.
*
* @param xml
*            The byte array that holds the XML source of either a
single SDMOData Entry or a
```

```
*            feed of several SDMOData entries.
* @return ISDMODataFeed The vector of the SDMOData Entries contained
by the source XML.
* @throws SDMParserException
*            If the XML source is invalid.
* @throws IllegalArgumentException
*            If the argument is null.
*/
public abstract ISDMODataFeed parseSDMODataEntriesXML(byte[] xml,
ISDMODataEntitySet eSet)
throws SDMParserException, IllegalArgumentException;

/**
* Parses an entry XML.
*
* @param xml
*            byte array the data is read from
* @param eSet
*            the related entity type
* @return ISDMODataEntry
* @throws SDMParserException
* @throws IllegalArgumentException
*/
public ISDMODataEntry parseSDMODataEntryXML(byte[] xml,
ISDMODataEntitySet eSet) throws SDMParserException,
IllegalArgumentException;
```

All the OData related classes are descendants of the generic SDMParserDocument class, meaning that its low level data access methods can be applied for the OData classes as well. This feature is useful when some information from the XML files is not accessible through the high level interfaces.

The structure of the metadata classes is built according to the OData object hierarchy. The information is accessed from two XMLs, the service document and the metadata XML. The service document is parsed first, then the metadata. The ISDMODataMetada object, which is received from the parser after processing the metadata XML is the root of the hierarchy. From this starting point, you can browse the whole hierarchy. Furthermore, from each lower level object, you can access its parent using the public ISDMParserDocument getParent() method. The ISDMParserDocument is the parent of all OData classes, so the result can be type cast to the proper OData type.

Collections and entity sets are in one-to-one relationship, containing even partially overlapping meta information about the corresponding atom feeds. This relationship is implemented through their name attribute, their non-qualified name is the same. However, used as method parameters, collection name is always without namespace, while entity set name is prefixed with the corresponding schema namespace.

Parsing is done without any data loss, that is, all the information contained in the XML is preserved in the resulted data structures. In addition to these data structures, the complete XML is also preserved. This is useful when the objects are persisted, because it is more

efficient to persist a simple string instead of a complex data structure. It is also an advantage when data is stored encrypted.

The only drawback of this solution is when data is restored from the persistent storage, the stored XMLs are parsed again. So this is an expensive operation and should be done as rarely as possible. To avoid degrading user experience, application developers should perform this operation (restore object structure from persistence) in a background thread.

Figure 1: Object Hierarchy
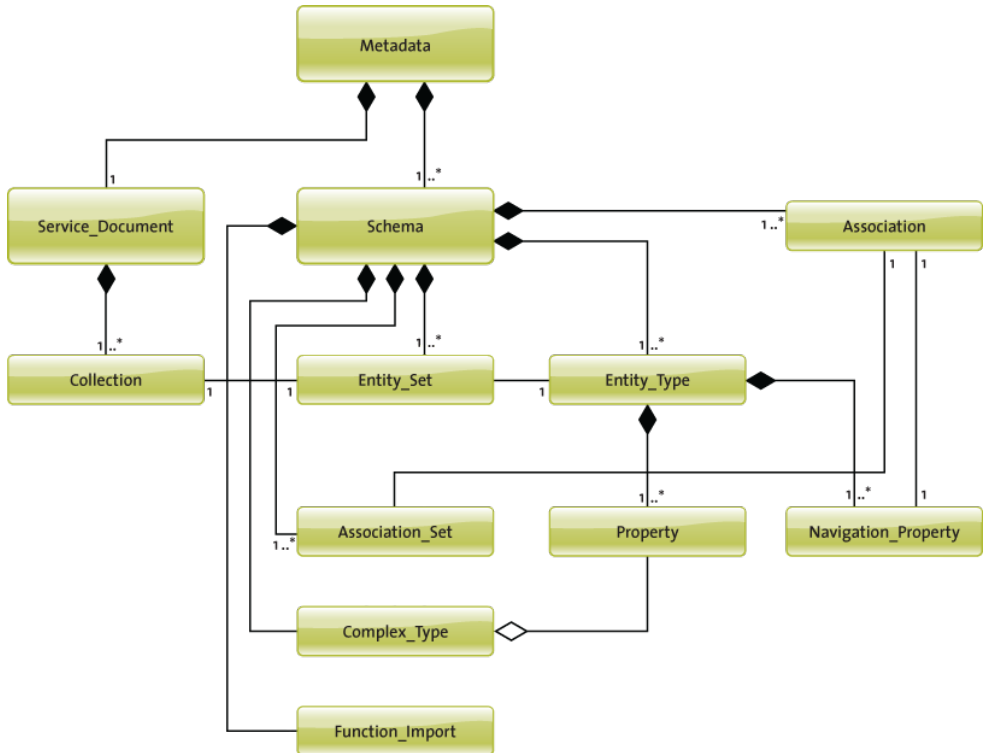
There are certain use cases, where OData entry objects and their XML representations have to be created on the client side. For this, the `SDMDataEntry` class provides the public constructor `SDMDataEntry(ISDMODataEntitySet eSet)`, which creates an empty entry object so that its attributes have to be set one-by-one by calling the corresponding setter method. Finally, the public `String toXMLString()` method generates the XML representation of the entry object.

SDMCache

The SDMCache component is responsible for storing and accessing OData related objects in the memory of the device.

List of Features

- Storing ISDMODataEntry objects in the memory
- Accessing ISDMODataEntry objects in the memory directly by their key
- Searching for ISDMODataEntry objects in the memory using tokenized prefix search on their searchable fields
- Managing the number of stored ISDMODataEntry objects based on the maximum size of the capacity, removing the least recently used OData document first

SDMCache Public APIs

```
ISDMCache

 void initialize(ISDMPreferences preferences);
    void clear();
 void setSDMODataServiceDocument(ISDMODataServiceDocument
serviceDocument);
    void setSDMODataSchema(ISDMODataSchema schema);
   void setSDMODataEntry(ISDMODataEntry entry, String collectionId);
    void setSDMODataEntries(Vector entries, String collectionId);
 ISDMODataServiceDocument getSDMODataServiceDocument();
    ISDMODataSchema getSDMODataSchema();
    ISDMODataEntry getSDMODataEntry(String key);
    Vector getSDMODataEntries(String collectionId);
    Vector getStoredDocuments();
    Vector searchSDMODataEntries(String searchTerm, String
collectionId);
    void removeSDMODataServiceDocument();
    void removeSDMODataSchema();
    void removeSDMODataEntry(String key);
    void removeSDMODataEntries(String collectionId);
    void removeStoredDocuments();
 Hashtable getStoreStructureForPersistency();
 void setStoreStructureForPersistency(Hashtable values);
```

Technical Details

For capacity management, SDMCache uses an LRU (least recently used) algorithm that ensures that the least recently used entries are removed first because of reaching the maximum capacity. Maximum number of capacity can be set using preference with key: ISDMPreferences.CACHE_MAX_ELEMENT_NR. This setting refers to the maximum number of cached entities per Collection.

SDMCache supports the tokenized prefix search. The gp:use-in-search property tag determines whether a field is searchable.

SDMCache depends on OData specific interfaces of SDMParser, but does not depend on the real implementation of SDMParser.

SDMPersistence

The Persistence layer stores the application's state and relevant data on the mobile device using the BlackBerry Persistent Store. The library exposes secure APIs, allowing encrypted data storage and decryption of data.

List of Features

- Storing and loading general objects from Persistent store
- Storing and loading the SDMCache object
- Storing and loading the SDMCache object in a secured way, which means that all fields of all objects within the cache will be encrypted/decrypted during the load/store operations. There is a specific method for the removal of the cache, but for the general objects, just a generic method is provided, where the persistent object id has to be provided as parameter.

SDMPersistence Public APIs

```
ISDMPersistence

void storeCache(final ISDMCache cache)
void storeCacheSecured(final ISDMCache cache)
void storePreferencesSecured(final ISDMPreferences preferences)
ISDMCache loadCache(ISDMCache cache, ISDMParser parser)
ISDMCache loadCacheSecured(ISDMCache cache, ISDMParser parser)
void loadPreferencesSecured(final ISDMPreferences preferences)
void storeObject(final long key, final Object object)
Object loadObject(final long key)
void clearCache()
void clearObject(final long key)
```

Technical Details

To persist data on the BlackBerry platform means storing objects in the storage provided by the platform (Persistent Store). Data is stored as instances of Persistent Objects. A PersistentObject can be any object that implements the Persistable interface. The Persistent Store API allows the implicit persistence of classes, so the following data types automatically implement the Persistable interface and can also be stored in the persistent store:

- java.lang.Boolean
- java.lang.Byte
- java.lang.Character
- java.lang.Integer
- java.lang.Long
- java.lang.Object
- java.lang.Short
- java.lang.String

- java.util.Vector
- java.util.Hashtable

The implementation only uses the above standard data types when persisting data. This approach is used as a custom persistent class cannot be used by two applications on the same device on the BlackBerry platform, and hence is not suitable for a static library component. In addition, this also avoids any limits on the number of custom persistent classes supported by the platform.

The storage for each application is distinct, because each object in the persistent store is associated with a 64-bit ID (type long). Data is stored in the Persistent Store which is a fast and optimized storage on the platform. The BlackBerry Persistent Store APIs are designed to provide a flexible and robust data storage interface. With the BlackBerry Persistent Store APIs, you can save entire Java® objects to the memory without having to serialize the data first. When the application is started, it can retrieve the Java object from the memory and process the information. No size limit exists on a persistent store; however, the limit for an individual object within the store is 64 KB.

When using standard persistent classes, each application must ensure to remove any persisted objects when the application is removed from the device. The BlackBerry OS does not automatically remove these objects in the same way as it does for custom persistent classes.

The applications have to implement the `CodeModuleListener` interface, which can react to module addition and removal events. Register the implementation to the `CodeModuleManager` with the `public static void addListener(Application application, CodeModuleListener listener)` method. The first parameter is the application whose event listener thread will execute the listener's code. This means that this application process must be running when the application removal is triggered. This can be achieved by adding an automatically starting background process to the applications and register the listener there.

An alternate entry point with automatic startup has to be added to the application descriptor:
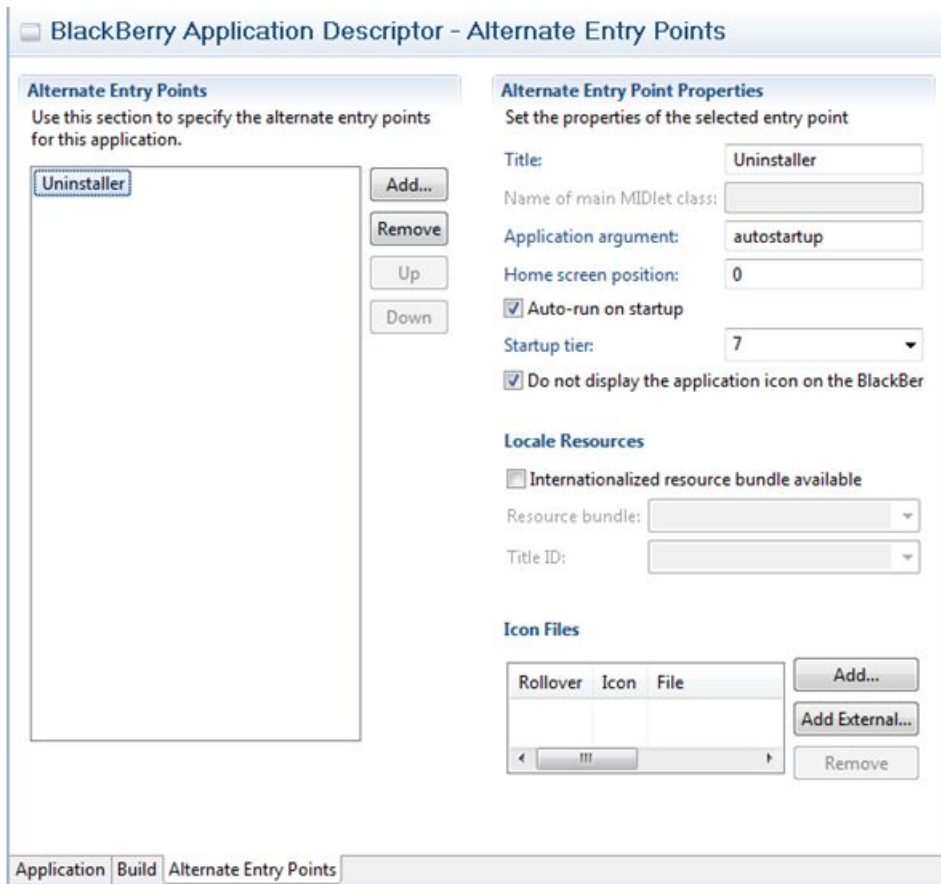
The main method of the application has to be extended with a branch for the background
process, which registers itself for code module changes:

```
public static void main(String[] args) {
  if (args.length >= 1 && args[0].equals("autostartup")) {

  // Background startup of the application. This process registers as
the listener for
  // code module life-cycle changes. This will be an always on
background process, which
  // will react, when its own module is marked for deletion.
    UninstallSampleApp theApp = new UninstallSampleApp(false);
    CodeModuleManager.addListener(theApp, theApp);

    theApp.requestBackground();
    theApp.enterEventDispatcher();
  } else {
  // Normal startup procedure: create a new instance of the
application which will run in
  // the foreground.
```

```
    UninstallSampleApp theApp = new UninstallSampleApp(true);
    theApp.enterEventDispatcher();
        }
}
```

The constructor receives a flag indicating whether it is running in the foreground, so the initialization tasks can be performed according to this information (that is, no UI is needed for the background process).

Implement the listener. It is called every time a module is about to be removed or added to the system, so the events must be filtered according to the module name.

```
public void moduleDeletionsPending(String[] modules) {
String currentModuleName =
ApplicationDescriptor.currentApplicationDescriptor().getModuleName(
);
SDMConstants constants = SDMConstants.getInstance();
for (int i=0; i < modules.length; i++) {
  if (modules[i].equals(currentModuleName)) {

PersistentStore.destroyPersistentObject(constants.getId(SDMConstant
s.SERVICE_DOC_KEY));

PersistentStore.destroyPersistentObject(constants.getId(SDMConstant
s.METADATA_KEY));

PersistentStore.destroyPersistentObject(constants.getId(SDMConstant
s.DATA_ENTRY_KEY));

PersistentStore.destroyPersistentObject(constants.getId(SDMConstant
s.PREFERENCES_KEY));
    break;
    }
}
```

This example shows how to remove the persisted cache components and the preferences, but any persisted application data can be removed the same way.

The BlackBerry Persistent Store APIs do not provide a relational database model. The application must create an effective object model and manage the relationships between objects as necessary, using indices and hash tables. The keys used to store/load objects must always be handled by the applications. Encryption/decryption is performed with the help of the PersistentContent object. Research In Motion (RIM) must track the use of some sensitive BlackBerry APIs for security and export control reasons. To load your application on a BlackBerry smart phone, the application must be signed using a signature key (provided by RIM). The application owner must order signing keys in order to access the BlackBerry runtime, application and cryptography APIs.

If your application is only signed by RIM provided keys, your application can use the Persistent Store, but there will not be any access control to the persisted data. Any kind of application signed by RIM keys can read and replace your persisted data. If you want to protect your data from other applications, you have to use the BlackBerry Signing Authority Tool to

sign the resulting cod file with your private key. If you do not have a private key for signing, you will also need to use the BlackBerry Signing Authority Admin Tool to create a public/ private key pair. See the *BlackBerry Signature Tool Development Guide* and the *BlackBerry Signing Authority Tool Administrator Guide* for more information. In order for your application to access protected persistent content, the developer must set the used signerID in

```
ISDMPreferences.PERSISTENCE_ACCESS_CONTROL_SIGNER_ID
```

preference.

The encryption/decryption in the case of saving a huge number of objects or, for example, a Vector which contains thousands of items can be slow on BlackBerry phones, because the operation must be done on each field of each object. For encryption, the library uses the underlying OS encryption API, no custom API is provided for this purpose. The BlackBerry API offers the `PersistentContent` class for the applications, which can be used to encrypt/decrypt Strings and byte arrays.

SDMConnectivity

The Network layer handles all network layer related tasks, hides the complexity of network communication, and provides easy to use APIs to the applications.

List of Features

- Provides interfaces for request handling
- Handles the requests asynchronously
- Can handle the requests by multiple number of threads (configurable)

Technical Details and SDMConnectivity Public APIs

Note: The SUP APIs and their descriptions are available after the installation of Sybase Unwired Platform at the following location within your installation folder: `...\UnwiredPlatform\ClientAPI\apidoc`.

The `SDMRequestManager` class implements the ISDMRequestManager interface, which provides the following methods:

```
ISDMRequestManager

void makeRequest(final ISDMRequest aRequest);
void makeRequest(final ISDMBundleRequest aBundleRequest)
ISDMConnectivitiyParameters getConnectivityParameters()
Vector getAllRequests()
int getQueueSize()
byte[] getRootContextID()
void terminate()
void pause()
void resume()
```

The number of working threads in the `RequestManager` class is configurable via the `initialize(final SDMConnectivityParameters aParameters, final int aThreadNumber)` method. The number of threads is maximized in four by the connectivity layer, because of performance related issues. If the client initializes the layer with more than the allowed threads, the implementation of the connectivity layer will decrease the thread number to the max allowed number. Methods defined by the `SDMConnectivityParameters` class:

```
ISDMConnectivityParameters

void setUserName(String aUserName)
String getUserName()
void setUserPassword(String aPassword)
String getUserPassword()
void setBaseUrl(String baseUrl)
String getBaseUrl()
String getLanguage()
void setLanguage(String language)
```

Sending requests with the connectivity layer consists of the following steps:

1. Create the `RequestManager` class and initialize it with the required parameters.
2. Create the request object. This can be done by implementing the ISDMRequest interface or by extending the `SDMBaseRequest` class which is the base implementation of the ISDMRequest interface. Both of them are provided by the connectivity layer.
3. Add the request object to the `SDMRequestManager`.

```
Example

//create and fill parameters for Connectivity library
SDMConnectivityParameters params = new SDMConnectivityParameters();
params.setUserName("test");
params.setUserPassword("testpwd");
params.setLogger(Logger.getInstance()); //get the default Logger
//create the RequestManager
SDMRequestManager reqManager = new SDMRequestManager();
//initialize it
reqManager.initialize(params, 2);//set the parameters and the thread
number to be used
//create the request object
ISDMRequest testRequest = new SDMBaseRequest();
testRequest.setRequestUrl("http://test.de:8080/testpath");
testRequest.setRequestMethod(ISDMRequest.REQUEST_METHOD_GET);
testRequest.setPriority(ISDMRequest.PRIORITY_NORMAL);
//add the request to the connectivity layer
reManager.makeRequest(testRequest);
```

The tasks of the connectivity library have been divided into three main categories: managing the request queues, managing reading and writing to the input/output streams, and managing the platform specific connection creation.

The Connectivity component always performs the requests in asynchronous mode. The application's role is to handle the request in sync mode. The component is able to perform

HTTP and HTTPS requests, which you can use for developing and testing purposes, but the default is SUP Request. The threads in the connectivity library are responsible for taking the requests from the queue (FIFO - First in first out - algorithm) and performing the requests.

The number of working threads in the connection pool can be configured in the connectivity layer. There is only one queue, and this is handled by the SDMRequestManager, and the working threads take the requests from this queue. Applications are interacting only with the SDMRequestManager class; the other components of the connectivity library are not visible to them. The network component consists of three main parts:

- SDMRequestManager: responsible for queuing the requests, managing the threads and keeping the connection with applications
- ConnectionHandler: responsible for performing the request
- ConnectionFactory: responsible for creating and managing platform dependent connections to the server

An application can have more than one SDMRequestManager, for example, when connecting to two different servers at the same time.

There is built-in support for setting the timeout for the socket connection, the application can use the SDMConnectivityParameters object to modify the value.

```
int TIMEOUT = 3500;

ISDMPreferences preferences = new SDMPreferences();

preferences.setPreference(ISDMPreferences.CONNECTION_TIMEOUT_MS,
String.valueOf(TIMEOUT));

requestManager = new SDMRequestManager(logger, preferences,
parameters, NUM_OF_HTTP_EXECUTION_THREADS);
```

SDMRequest Object

An SDMRequest object wraps all the information which is needed by the connectivity library to be able to perform the requests. The connectivity library interacts with the request object to query the necessary information about the headers, the post data, and so on.

The connectivity layer also uses the request object to notify the application about the result of the request using the ISDMNetListener interface. The connectivity component provides an interface called the ISDMRequest and a base implementation of it called the SDMBaseRequest. The applications have to extend this base class when creating new application specific requests. The ISDMRequest interface defines the following public APIs:

```
ISDMRequest

void setRequestUrl(final String aUrl)
String getRequestUrl()
void setRequestMethod(final int aRequestMethod)
int getRequestMethod()
byte[] getData()
void setPriority(final int aPriority)
int getPriority()
```

```
boolean useCookies()
void setListener(final ISDMNetListener aListener)
ISDMNetListener getListener()
boolean hasPostData()
void postData(OutputStream os)
void setHeaders(final Hashtable aHashtable)
Hashtable getHeaders()
void appendHeaders(final Hashtable aHashtable)
void appendHeader(final String aHeaderName, final String
aHeaderValue)
```

The ISDMNetListener interface can be used by the client to be notified by the connectivity layer about the result of a request. Usage of this feature is not mandatory, however, you can handle incidental errors with it. Methods available in the ISDMNetListener interface:

```
ISDMNetListener

void onSuccess(ISDMRequest aRequest, IHttpResponse aResponse)
void onError(ISDMRequest aRequest, IHttpResponse aResponse,
ISDMRequestStateElement aRequestStateElement)
```

The role of the SDMRequestStateElement object used by the connectivity library is to provide the application with more detail on the occurred error. Methods available in SDMRequestStateElement object:

```
ISDMRequestStateElement

int getErrorCode()
void setErrorCode(final int code)
int getHttpStatusCode()
void setHttpStatusCode(final int httpStatus)
Exception getException()
void setException(final Exception aException)
String getRedirectLocation()
IHttpResponse getResponse()
```

```
Example

public void onSuccess(ISDMRequest  aRequest, SDMHttpResponse
aResponse) {
     System.out.println("Http response status code:" +
aResponse.getStatusCode());
     System.out.println("Cookie string:" +
aResponse.getCookieString());
     byte[] content = aResponse.getContent();
     String response = new String(content);
     System.out.println("Received content:" + response);
     //get the headers
     Hashtable headers = aResponse.getHeaders();
}
```

SDMConfiguration

Each low level API has its own defaults/constants set in the SDMConfiguration library. Default values of preferences can be found in the SDMConstants class.

List of Features

- Providing modifiable preferences for SDMComponent libraries
- Encrypting/decrypting values of preferences for persistence
- Providing API for resetting the preferences of SDMComponent libraries to their default values
- Providing API for creating and handling custom preferences
- Notifying subscribed listeners in case of any change in preferences

SDMConfiguration Public APIs

```
ISDMPreferences

void setPreference(String key, String value)
String getPreference(String key)
void registerPreferenceChangeListener(String key,
ISDMPreferenceChangeListener changeListener)
void unRegisterPreferenceChangeListener(String key,
ISDMPreferenceChangeListener changeListener)
Hashtable encrypt()
Hashtable decrypt()
void initFromPersistence(Hashtable prefs)
void deletePreference(final String aKey)
void reset()
```

Technical Details

SDMPreferences object is used for storing configuration key-value pairs. Only the String representation of the value can be stored. Persistent storage of this object is available from SDMPersistence. This object calls `encrypt()`, `decrypt()` and `initFromPersistence()` methods of SDMPreferences, so the applications do not have to use these methods explicitly.

During instantiation of SDMPreference, the default values needed for other SDMComponents are filled. SDMComponents preferences can be reset to their default values using the `reset()` method.

You can register a preference change listener for each preference in SDMPreferences (including custom preferences) so that you will be notified if the value of a given preference has changed. Preference change listener notification and preference validation can only be done after the initialization of the appropriate component.

SDMSupportability

The OData SDK provides a set of features and concepts for the supportability of the applications built on top of the SDK.

SDMLogger

The SDMLogger architecture follows the logging implementation in Java 1.5 and provides the same services and structures, but also contains BlackBerry and OData SDK specific implementations.

The component provides the following features:

- Filtering: the client app can set the log level. Provides filterable log retrieval by component and by timestamp (from-to).
- Formatting: before the log message is sent to the handler (which performs the logging), there is a possibility to format the message.
- Handlers: handlers are responsible for logging the messages to the specified place. Depending on the implementation of the handler, the place can be the memory, a file, or the message can be sent to the server. Changing the default handlers in the Logger implementation is invisible for the client.

Current implementation contains implementation for all the interfaces (the IFilter, IHandler and IFormatter). These classes begin with the "Default" prefix.

SDMLogger Public APIs

```
ISDMLogger

ISDMPreferences getPreferences()
void entering(String sourceClass, String sourceMethod)
void entering(String sourceClass, String sourceMethod, Object
param1)
void entering(String sourceClass, String sourceMethod,
            Object[] params)
void exiting(String sourceClass, String sourceMethod)
void exiting(String sourceClass, String sourceMethod, Object result)
void fine(String msg)
void finer(String msg)
void info(String msg)
void log(final int level, String msg)
void log(final int level, String msg, final Object param1)
void log(final int level, String msg, Object[] params)
void log(final int level, String msg, Throwable thrown)
void log(final int level, final String message, final Exception ex)
void logNestedObjects(final int level, String message,
            final Object[] params)
void setHandler(IHandler handler)
void error(String msg)
void p(final String message, long timestamp)
Vector getLogRecords()
Vector getLogRecorsdByComponentName(final String componentName)
```

```
Vector getLogRecorsdByTimeStamp(final long start, final long end)
void clearLogRecords()
int getLogNumber()
String getLogHeader()
```

SAP Passport

For the Single Activity Trace an SAP Passport has to be issued by the connectivity layer of the library.

The SAP Passport is transported as an HTTP header in the request. The server handles the SAP Passport to generate end-to-end Trace. The OData SDK is using JSDR SAP Passport sources integrated in the library at source level. It can be turned on or off with `ISDMPreferences.SDM_TRACING_ENABLED` preference key. By default it is turned off.

Deploying Applications to Devices

This section describes how to deploy customized mobile applications to devices.

1. *Signing*

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

2. *Provisioning Options for BlackBerry Devices*

   To provision the application to BlackBerry devices, you can automatically push the application to the device or send a link to device users so they can install it when desired. For small deployments or evaluation purposes, device users can install the application using BlackBerry Desktop Manager.

3. *BES Provisioning for BlackBerry*

   BlackBerry devices that are connected to a production environment using relay server can use BlackBerry Enterprise Server (BES) to provision supported device types.

4. *BlackBerry Desktop Manager Provisioning*

   You can deploy BlackBerry applications to physical devices through BlackBerry Desktop Manager.

See also

- *OData SDK Components and APIs* on page 95

Signing

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

In general, if your application or library uses an API it must be signed, which occurs in most cases. You can implement code signing from the BlackBerry JDE:

- BlackBerry JDE – download the Signing Authority Tool from the BlackBerry Web site at *http://na.blackberry.com/eng/developers/javaappdev/signingauthority.jsp*. View

---

Deploying and Signing Applications in the BlackBerry JDE plug-in for Eclipse at the
Research In Motion Developer Video Library Web site: *http://
supportforums.blackberry.com/t5/Java-Development/tkb-p/java_dev%40tkb?
labels=video*.

Provisioning Options for BlackBerry Devices

To provision the application to BlackBerry devices, you can automatically push the
application to the device or send a link to device users so they can install it when desired. For
small deployments or evaluation purposes, device users can install the application using
BlackBerry Desktop Manager.

Once installed on the device, the application appears in Downloads. However, device users
can move it to a different location. If device users reinstall the application from a link or URL,
or using Desktop Manager, the BlackBerry device remembers the installation location.

Provisioning Method	Purpose	Description
BlackBerry Enterprise Server (BES) Over-the-Air (OTA)	Enterprise installations	When the BlackBerry device activates, it automatically pairs with the BES and downloads the application. See *http://www.blackberry.com/btsc/search.do?cmd=displayKC&docType=kc&external-Id=KB03748* for step-by-step instructions.
OTA: URL/link to installation files	Enterprise installations	The administrator stages the OTA files in a Web-accessible location and notifies BlackBerry device users via an e-mail message with a link to the JAD file.
Desktop Manager	Personal installation	Installs the application when the BlackBerry device is synced via a computer.

BES Provisioning for BlackBerry

BlackBerry devices that are connected to a production environment using relay server can use
BlackBerry Enterprise Server (BES) to provision supported device types.

See the following sections in *System Administration* for details on how to perform
BlackBerry provisioning and deployment:

- *System Administration > Device Provisioning > Afaria Provisioning and Mobile Device
  Management*.
- *System Administration > Device Provisioning > BES Provisioning for BlackBerry*
  - *Provisioning Prerequisites for BlackBerry*

• *System Administration > Device Provisioning > Setting up Push Synchronization for Replication Synchronization Devices*

BlackBerry Desktop Manager Provisioning

You can deploy BlackBerry applications to physical devices through BlackBerry Desktop Manager.

The generated code is compiled against the BlackBerry RAPC compiler to output the following COD (.cod), Application Loader Files (.alx), and Java Application Descriptor (.jad) files. File requirements depend on application and installation type:

Required files include:

• CommonClientLib.cod
• MessagingClientLib.cod
• MocaClientLib.cod
• sup_json.cod

CHAPTER 5 **Glossary: Sybase Unwired Platform**

Defines terms for all Sybase Unwired Platform components.

administration perspective – Or administration console. The Unwired Platform administrative perspective is the Flash-based Web application for managing Unwired Server. *See* Sybase Control Center.

administrators – Unwired Platform users to which an administration role has been assigned. A user with the "SUP Administrator" role is called a "platform administrator" and a user with the "SUP Domain Administrator" role is called a "domain administrator". These administration roles must also be assigned SCC administration roles to avoid having to authenticate to Sybase Control Center in addition to Unwired Server:

• A domain administrator only requires the "sccUserRole" role.
• A platform administrator requires both the "sccAdminRole" and "sccUserRole" roles.

Adobe Flash Player – Adobe Flash Player is required to run Sybase Control Center. Because of this player, you are required to run Sybase Control Center in a 32-bit browser. Adobe does not support 64-bit browsers.

Advantage Database Server® – A relational database management system that provides the messaging database for Sybase Unwired Platform. *See* messaging database.

Afaria® – An enterprise-grade, highly scalable device management solution with advanced capabilities to ensure that mobile data and devices are up-to-date, reliable, and secure. Afaria is a separately licensed product that can extend the Unwired Platform in a mobile enterprise. Afaria includes a server (Afaria Server), a database (Afaria Database), an administration tool (Afaria Administrator), and other runtime components, depending on the license you purchase.

application – In Unwired Server (and visible in Sybase Control Center), and application is the runtime entity that can be directly correlated to a native or mobile workflow application. The application definition on the server establishes the relationship among packages used in the application, domain that the application is deployed to, user activation method for the application, and other application specific settings.

APNS – Apple Push Notification Service.

application connection – A unique connection to the application on a device.

application connection template – a template for application connections that includes application settings, security configuration, domain details, and so forth.

application node – In Sybase Control Center, this is a registered application with a unique ID. This is the main entity that defines the behavior of device and backend interactions.

application registration – The process of registering an application with Sybase Unwired Platform. Registration requires a unique identity that defines the properties for the device and backend interaction with Unwired Server.

artifacts – Artifacts can be client-side or automatically generated files; for example: `.xml`, `.cs`, `.java`, `.cab` files.

availability – Indicates that a resource is accessible and responsive.

BAPI – Business Application Programming Interface. A BAPI is a set of interfaces to object-oriented programming methods that enable a programmer to integrate third-party software into the proprietary R/3 product from SAP®. For specific business tasks such as uploading transactional data, BAPIs are implemented and stored in the R/3 system as remote function call (RFC) modules.

BLOB – Binary Large Object. A BLOB is a collection of binary data stored as a single entity in a database management system. A BLOB may be text, images, audio, or video.

cache – The virtual tables in the Unwired Server cache database that store synchronization data. *See* cache database.

cache group – Defined in Unwired WorkSpace, MBOs are grouped and the same cache refresh policy is applied to their virtual tables (cache) in the cache database

cache partitions – Partitioning the cache divides it into segments that can be refreshed individually, which gives better system performance than refreshing the entire cache. Define cache partitions in Unwired WorkSpace by defining a partition key, which is a load argument used by the operation to load data into the cache from the enterprise information system (EIS).

cache database – Cache database. The Unwired Server cache database stores runtime metadata (for Unwired Platform components) and cache data (for MBOs). *See also* data tier.

CLI – Command line interface. CLI is the standard term for a command line tool or utility.

client application – *See* mobile application.

client object API – The client object API is described in the *Developer Guide: BlackBerry Native Applications*, *Developer Guide: iOS Native Applications*, and *Developer Guide: Windows and Windows Mobile Native Applications*.

cluster – Also known as a server farm. Typically clusters are setup as either runtime server clusters or database clusters (also known as a data tier). Clustering is a method of setting up redundant Unwired Platform components on your network in order to design a highly scalable and available system architecture.

cluster database – A data tier component that holds information pertaining to all Unwired Platform server nodes. Other databases in the Unwired Platform data tier includes the cache, messaging, and monitoring databases.

connection – Includes the configuration details and credentials required to connect to a database, Web service, or other EIS.

connection pool – A connection pool is a cache of Enterprise Information System (EIS) connections maintained by Unwired Server, so that the connections can be reused when Unwired Server receives future requests for data.

For proxy connections, a connection pool is a collection of proxy connections pooled for their respective back-ends, such as SAP Gateway.

connection profile – In Unwired WorkSpace, a connection profile includes the configuration details and credentials required to connect to an EIS.

context variable – In Unwired WorkSpace, these variables are automatically created when a developer adds reference(s) to an MBO in a mobile application. One table context variable is created for each MBO attribute. These variables allow mobile application developers to specify form fields or operation parameters to use the dynamic value of a selected record of an MBO during runtime.

data change notification (DCN) – Data change notification (DCN) allows an Enterprise Information System (EIS) to synchronize its data with the cache database through a push event.

data refresh – A data refresh synchronizes data between the cache database and a back-end EIS so that data in the cache is updated. *See also* scheduled data refresh.

data source – In Unwired WorkSpace, a data source is the persistent-storage location for the data that a mobile business object can access.

data tier – The data tier includes Unwired Server data such as cache, cluster information, and monitoring. The data tier includes the cache database (CDB), cluster, monitoring, and messaging databases.

data vault – A secure store across the platform that is provided by an SUP client.

deploy – (Unwired Server) Uploading a deployment archive or deployment unit to an Unwired Server instance. Unwired Server can then make these units accessible to users via a client application that is installed on a mobile device.

There is a one-to-one mapping between an Unwired WorkSpace project and a server package. Therefore, all MBOs that you deploy from one project to the same server are deployed to the same server package.

deployment archive – In Unwired WorkSpace, a deployment archive is created when a developer creates a package profile and executes the **build** operation. Building creates an archive that contains both a deployment unit and a corresponding descriptor file. A

deployment archive can be delivered to an administrator for deployment to a production version of Unwired Server.

deployment descriptor – A deployment descriptor is an XML file that describes how a deployment unit should be deployed to Unwired Server. A deployment descriptor contains role-mapping and domain-connection information. You can deliver a deployment descriptor and a deployment unit—jointly called a deployment archive—to an administrator for deployment to a production version of Unwired Server.

deployment mode – You can set the mode in which a mobile application project or mobile deployment package is deployed to the target Unwired Server.

deployment profile – A deployment profile is a named instance of predefined server connections and role mappings that allows developers to automate deployment of multiple packages from Sybase Unwired WorkSpace to Unwired Server. Role mappings and connection mappings are transferred from the deployment profile to the deployment unit and the deployment descriptor.

deployment unit – The Unwired WorkSpace build process generates a deployment unit. It enables a mobile application to be effectively installed and used in either a preproduction or production environment. Once generated, a deployment unit allows anyone to deploy all required objects, logical roles, personalization keys, and server connection information together, without requiring access to the whole development project. You can deliver a deployment unit and a deployment descriptor—jointly called a deployment archive—to an administrator for deployment to a production version of Unwired Server.

development package – A collection of MBOs that you create in Unwired WorkSpace. You can deploy the contents of a development package on an instance of Unwired Server.

device application – *See also* mobile application. A device application is a software application that runs on a mobile device.

device notification – Replication synchronization clients receive device notifications when a data change is detected for any of the MBOs in the synchronization group to which they are subscribed. Both the change detection interval of the synchronization group and the notification threshold of the subscription determine how often replication clients receive device notifications. Administrators can use subscription templates to specify the notification threshold for a particular synchronization group.

device user – The user identity tied to a device.

DML – Data manipulation language. DML is a group of computer languages used to retrieve, insert, delete, and update data in a database.

DMZ – Demilitarized zone; also known as a perimeter network. The DMZ adds a layer of security to the local area network (LAN), where computers run behind a firewall. Hosts running in the DMZ cannot send requests directly to hosts running in the LAN.

domain administrator – A user to which the platform administrator assigns domain administration privileges for one or more domain partitions. The domain administrator has a restricted view in Sybase Control Center, and only features and domains they can manage are visible.

domains – Domains provide a logical partitioning of a hosting organization's environment, so that the organization achieves increased flexibility and granularity of control in multitenant environments. By default, the Unwired Platform installer creates a single domain named "default". However the platform administrator can also add more domains as required.

EIS – Enterprise Information System. EIS is a back-end system, such as a database.

Enterprise Explorer – In Unwired WorkSpace, Enterprise Explorer allows you to define data source and view their metadata (schema objects in case of database, BAPIs for SAP, and so on).

export – The Unwired Platform administrator can export the mobile objects, then import them to another server on the network. That server should meet the requirement needed by the exported MBO.

hostability – *See* multitenancy.

IDE – Integrated Development Environment.

JDE – BlackBerry Java Development Environment.

key performance indicator (KPI) – Used by Unwired Platform monitoring. KPIs are monitoring metrics that are made up for an object, using counters, activities, and time which jointly for the parameters that show the health of the system. KPIs can use current data or historical data.

keystore – The location in which encryption keys, digital certificates, and other credentials in either encrypted or unencrypted keystore file types are stored for Unwired Server runtime components. *See also* truststore.

LDAP – Lightweight Directory Access Protocol.

local business object – Defined in Unwired WorkSpace, local business objects are not bound to EIS data sources, so cannot be synchronized. Instead, they are objects that are used as local data store on device.

logical role – Logical roles are defined in mobile business objects, and mapped to physical roles when the deployment unit that contain the mobile business objects are deployed to Unwired Server.

matching rules – A rule that triggers a mobile workflow application. Matching rules are used by the mobile workflow email listener to identify e-mails that match the rules specified by the administrator. When emails match the rule, Unwired Server sends the e-mail as a mobile workflow to the device that matches the rule. A matching rule is configured by the administrator in Sybase Control Center.

MBO – Mobile business object. The fundamental unit of data exchange in Sybase Unwired Platform. An MBO roughly corresponds to a data set from a back-end data source. The data can come from a database query, a Web service operation, or SAP. An MBO contains both concrete implementation-level details and abstract interface-level details. At the implementation-level, an MBO contains read-only result fields that contain metadata about the data in the implementation, and parameters that are passed to the back-end data source. At the interface-level, an MBO contains attributes that map to result fields, which correspond to client properties. An MBO may have operations, which can also contain parameters that map to arguments, and which determines how the client passes information to the enterprise information system (EIS).

You can define relationships between MBOs, and link attributes and parameters in one MBO to attributes and parameters in another MBO.

MBO attribute – An MBO attribute is a field that can hold data. You can map an MBO attribute to a result field in a back-end data source; for example, a result field in a database table.

MBO binding – An MBO binding links MBO attributes and operations to a physical data source through a connection profile.

MBO operation – An MBO operation can be invoked from a client application to perform a task; for example, create, delete, or update data in the EIS.

MBO relationship – MBO relationships are analogous to links created by foreign keys in a relational database. For example, the account MBO has a field called *owner_ID* that maps to the *ID* field in the owner MBO.

Define MBO relationships to facilitate:

• Data synchronization
• EIS data-refresh policy

messaging based synchronization – A synchronization method where data is delivered asynchronously using a secure, reliable messaging protocol. This method provides fine-grained synchronization (synchronization is provided at the data level—each process communicates only with the process it depends on), and it is therefore assumed that the device is always connected and available. *See also* synchronization.

messaging database – The messaging database allows in-flight messages to be stored until they can be delivered. This database is used in a messaging based synchronization environment. The messaging database is part of the Unwired Platform data tier, along with the cache, cluster, and monitoring databases.

mobile application – A Sybase Unwired Platform mobile application is an end-to-end application, which includes the MBO definition (back-end data connection, attributes, operations, and relationships), the generated server-side code, and the client-side application code.

Mobile Application Diagram – The Mobile Application Diagram is the graphical interface to create and edit MBOs. By dragging and dropping a data source onto the Mobile Application Diagram, you can create a mobile business object and generate its attribute mappings automatically.

Mobile Application Project – A collection of MBOs and client-side, design-time artifacts that make up a mobile application.

mobile workflow packages – Mobile workflow packages use the messaging synchronization model. The mobile workflow packages are deployed to Unwired Server, and can be deployed to mobile devices, via the Unwired Platform administrative perspective in Sybase Control Center.

monitoring – Monitoring is an Unwired Platform feature available in Sybase Control Center that allows administrators to identify key areas of weakness or periods of high activity in the particular area they are monitoring. It can be used for system diagnostic or for troubleshooting. Monitored operations include replication synchronization, messaging synchronization, messaging queue, data change notification, device notification, package, user, and cache activity.

monitoring database – A database that exclusively stores data related to replication and messaging synchronization, queues status, users, data change notifications, and device notifications activities. By default, the monitoring database runs in the same data tier as the cache database, messaging database and cluster database.

monitoring profiles – Monitoring profiles specify a monitoring schedule for a particular group of packages. These profiles let administrators collect granular data on which to base domain maintenance and configuration decisions.

multitenancy – The ability to host multiple tenants in one Unwired Cluster. Also known as hostability. *See also* domains.

node – A host or server computer upon which one or more runtime components have been installed.

object query – Defined in Unwired WorkSpace for an MBO and used to filter data that is downloaded to the device.

onboarding – The enterprise-level activation of an authentic device, a user, and an application entity as a combination, in Unwired Server.

operation – *See* MBO operation.

package – A package is a named container for one or more MBOs. On Unwired Server a package contains MBOs that have been deployed to this instance of the server.

palette – In Unwired WorkSpace, the palette is the graphical interface view from which you can add MBOs, local business objects, structures, relationships, attributes, and operations to the Mobile Application Diagram.

parameter – A parameter is a value that is passed to an operation/method. The operation uses the value to determine the output. When you create an MBO, you can map MBO parameters to data-source arguments. For example, if a data source looks up population based on a state abbreviation, the MBO gets the state from the user, then passes it (as a parameter/argument) to the data source to retrieve the information. Parameters can be:

- Synchronization parameters – synchronize a device application based on the value of the parameter.
- Load arguments – perform a data refresh based on the value of the argument.
- Operation parameters – MBO operations contain parameters that map to data source arguments. Operation parameters determine how the client passes information to the enterprise information system (EIS).

personalization key – A personalization key allows a mobile device user to specify attribute values that are used as parameters for selecting data from a data source. Personalization keys are also used as operation parameters. Personalization keys are set at the package level. There are three type of personalization keys: Transient, client, server.

They are most useful when they are used in multiple places within a mobile application, or in multiple mobile applications on the same server. Personalization keys may include attributes such as name, address, zip code, currency, location, customer list, and so forth.

perspective – A named tab in Sybase Control Center that contains a collection of managed resources (such as servers) and a set of views associated with those resources. The views in a perspective are chosen by users of the perspective. You can create as many perspectives as you need and customize them to monitor and manage your resources.

Perspectives allow you to group resources ways that make sense in your environment—by location, department, or project, for example.

physical role – A security provider group or role that is used to control access to Unwired Server resources.

Problems view – In Eclipse, the Problems view displays errors or warnings for the Mobile Application Project.

provisioning – The process of setting up a mobile device with required runtimes and device applications. Depending on the synchronization model used and depending on whether or not the device is also an Afaria client, the files and data required to provision the device varies.

pull synchronization – Pull synchronization is initiated by a remote client to synchronize the local database with the cache database. On Windows Mobile, pull synchronization is supported only in replication applications.

push synchronization – Push is the server-initiated process of downloading data from Unwired Server to a remote client, at defined intervals, or based upon the occurrence of an event.

queue – In-flight messages for a messaging application are saved in a queue. A queue is a list of pending activities. The server then sends messages to specific destinations in the order that

they appear in the queue. The depth of the queue indicates how many messages are waiting to be delivered.

relationship – *See* MBO relationship.

relay server – *See also* Sybase Hosted Relay Service.

resource – A unique Sybase product component (such as a server) or a subcomponent.

REST web services – Representational State Transfer (REST) is a style of software architecture for distributed hypermedia systems such as the World Wide Web.

RFC – Remote Function Call. You can use the RFC interface to write applications that communicate with SAP R/3 applications and databases. An RFC is a standalone function. Developers use SAP tools to write the Advanced Business Application Programming (ABAP) code that implements the logic of a function, and then mark it as "remotely callable," which turns an ABAP function into an RFC.

role – Roles control access to Sybase Unwired Platform resources. *See also* logical role and physical role.

role mapping – Maps a physical (server role) to a logical (Unwired Platform role). Role mappings can be defined by developers, when they deploy an MBO package to a development Unwired Server, or by platform or domain administrators when they assign a security configuration to a domain or deploy a package to a production Unwired Server (and thereby override the domain-wide settings in the security configuration).

RSOE – Relay Server Outbound Enabler. An RSOE is an application that manages communication between Unwired Server and a relay server.

runtime server – An instance of Unwired Server that is running. Typically, a reference to the runtime server implies a connection to it.

SAP – SAP is one of the EIS types that Unwired Platform supports.

SCC – Sybase Control Center. A Web-based interface that allows you to administer your installed Sybase products.

schedule – The definition of a task (such as the collection of a set of statistics) and the time interval at which the task must execute in Sybase Control Center.

scheduled data refresh – Data is updated in the cache database from a back-end EIS, based on a scheduled data refresh. Typically, data is retrieved from an EIS (for example, SAP) when a device user synchronizes. However, if an administrator wants the data to be preloaded for a mobile business object, a data refresh can be scheduled so that data is saved locally in a cache. By preloading data with a scheduled refresh, the data is available in the information server when a user synchronizes data from a device. Scheduled data refresh requires that an administrator define a cache group as "scheduled" (as opposed to "on-demand").

security configuration – Part of the application user and administration user security. A security configuration determines the scope of user identity, authentication and authorization

---

checks, and can be assigned to one or more domains by the platform administrator in Sybase Control Center. A security configuration contains:

- A set of configured security providers (for example LDAP) to which authentication, authorization, attribution is delegated.
- Role mappings (which can be specified at the domain or package level)

security provider – A security provider and it's repository holds information about the users, security roles, security policies, and credentials used by some to provide security services to Unwired Platform. A security provider is part of a security configuration.

security profile – Part of the Unwired Server runtime component security. A security profile includes encryption metadata to capture certificate alias and the type of authentication used by server components. By using a security profile, the administrator creates a secured port over which components communicate.

server connection – The connection between Unwired WorkSpace and a back-end EIS is called a server connection.

server farm – *See also* cluster. Is the relay server designation for a cluster.

server-initiated synchronization – *See* push synchronization.

SOAP – Simple Object Access Protocol. SOAP is an XML-based protocol that enables applications to exchange information over HTTP. SOAP is used when Unwired Server communicates with a Web service.

solution – In Visual Studio, a solution is the high-level local workspace that contains the projects users create.

Solution Explorer – In Visual Studio, the Solution Explorer pane displays the active projects in a tree view.

SSO – Single sign-on. SSO is a credential-based authentication mechanism.

statistics – In Unwired Platform, the information collected by the monitoring database to determine if your system is running as efficiently as possible. Statistics can be current or historical. Current or historical data can be used to determine system availability or performance. Performance statistics are known as key performance indicators (KPI).

Start Page – In Visual Studio, the Start Page is the first page that displays when you launch the application.

structured data – Structured data can be displayed in a table with columns and labels.

structure object – Defined in Unwired WorkSpace, structures hold complex datatypes, for example, a table input to a SAP operation.

subscription – A subscription defines how data is transferred between a user's mobile device and Unwired Server. Subscriptions are used to notify a device user of data changes, then these updates are pushed to the user's mobile device.

Sybase Control Center – Sybase Control Center is the Flash-based Web application that includes a management framework for multiple Sybase server products, including Unwired Platform. Using the Unwired Platform administration perspective in Sybase Control Center, you can register clusters to manage Unwired Server, manage domains, security configurations, users, devices, connections, as well as monitor the environment. You can also deploy and MBO or workflow packages, as well as register applications and define templates for them. Only use the features and documentation for Unwired Platform. Default features and documentation in Sybase Control Center do not always apply to the Unwired Platform use case.

Sybase Control Center *X.X* Service – Provides runtime services to manage, monitor, and control distributed Sybase resources. The service must be running for Sybase Control Center to run. Previously called Sybase Unified Agent.

Sybase Hosted Relay Service – The Sybase Hosted Relay Service is a Web-hosted relay server that enables you to test your Unwired Platform development system.

Sybase Messaging Service – The synchronization service that facilitates communication with device client applications.

Sybase Unwired Platform – Sybase Unwired Platform is a development and administrative platform that enables you to mobilize your enterprise. With Unwired Platform, you can develop mobile business objects in the Unwired WorkSpace development environment, connect to structured and unstructured data sources, develop mobile applications, deploy mobile business objects and applications to Unwired Server, which manages messaging and data services between your data sources and your mobile devices.

Sybase Unwired WorkSpace – Sybase Unwired Platform includes Unwired WorkSpace, which is a development tool for creating mobile business objects and mobile applications.

synchronization – A synchronization method where data is delivered synchronously using an upload/download pattern. For push-enabled clients, synchronization uses a "poke-pull" model, where a notification is pushed to the device (poke), and the device fetches the content (pull), and is assumed that the device is not always connected to the network and can operate in a disconnected mode and still be productive. For clients that are not push-enabled, the default synchronization model is pull.*See also* messaging based synchronization.

synchronization group – Defined in Unwired WorkSpace, a synchronization group is a collection of MBOs that are synchronized at the same time.

synchronization parameter – A synchronization parameter is an MBO attribute used to filter and synchronize data between a mobile device and Unwired Server.

synchronization phase – For replication based synchronization packages, the phase can be an upload event (from device to the Unwired Server cache database) or download event (from the cache database to the device).

synchronize – *See also* data refresh. Synchronization is the process by which data consistency and population is achieved between remote disconnected clients and Unwired Server.

truststore – The location in which certificate authority (CA) signing certificates are stored. *See also* keystore.

undeploy – Running **undeploy** removes a domain package from an Unwired Server.

Unwired Server – The application server included with the Sybase Unwired Platform product that manages mobile applications, back-end EIS synchronization, communication, security, transactions, and scheduling.

user – Sybase Control Center displays the mobile-device users who are registered with the server.

view – A window in a perspective that displays information about one or more managed resources. Some views also let you interact with managed resources or with Sybase Control Center itself. For example, the Perspective Resources view lists all the resources managed by the current perspective. Other views allow you to configure alerts, view the topology of a replication environment, and graph performance statistics.

Visual Studio – Microsoft Visual Studio is an integrated development environment product that you can use to develop device applications from generated Unwired WorkSpace code.

Welcome page – In Eclipse, the first set of pages that display when you launch the application.

workspace – In Eclipse, a workspace is the directory on your local machine where Eclipse stores the projects that you create.

WorkSpace Navigator – In Eclipse, the tree view that displays your mobile application projects.

WSDL file – Web Service Definition Language file. The file that describes the Web service interface that allows clients to communicate with the Web service. When you create a Web service connection for a mobile business object, you enter the location of a WSDL file in the URL.

CHAPTER 6   Glossary: OData SDK and Online Data Proxy

Defines terms for OData and Online Data Proxy when used with Sybase Unwired Platform components.

cache – In the context of OData applications: a memory system component responsible for storing and accessing OData related objects in the memory of the mobile device for quick access.

collection – Resource that contains a set of entries which are structured according to the Data Object / Entity Type definition in the respective Data Model. In OData, a Collection is represented as an Atom Feed or an array of JSON objects.

mobile application – Applications that run on smartphones and other mobile devices. SUP Mobile applications make SAP content available outside the corporate firewall and connect users to SAP services that are more commonly accessed on desktop computers.

OData metadata document – OData metadata documents describe the Entity Data Model (EDM) for a given service, which is the underlying abstract data model used by OData services to formalize the description of the resources it exposes.

OData (Open Data Protocol) – Web protocol for querying and updating data. It applies and builds upon Web technologies such as HTTP, Atom Publishing Protocol (AtomPub) and JSON to provide access to information from a variety of applications.

OData for SAP – OData for SAP Products provide SAP Extensions to the OData protocol that enable users to build user interfaces for accessing the data published via OData. The interfaces require human-readable, language-dependent labels for all properties and free-text search within collections of similar entities and across (OpenSearch).

OData Schema – Defines the structure of the xml files in the OData service.

OData Service Document – A document that describes the location and capabilities of one or more Collections.

Online Data Proxy – A light-weight edition of the Sybase Unwired Platform that provides a robust mobile infrastructure for enterprise IT organizations to securely roll-out and manage the deployment of light-weight applications in a controlled and monitored approach.

SAP – SAP Business Suite applications (such as ERP, CRM, SRM, SCM, Industry Solutions and so on) consist of many technologies and components. Unless stated otherwise, the term "SAP" means a backend business application that is based on the SAP NetWeaver ABAP application server, for example ECC 6.0.

SAP NetWeaver Gateway – Enables people-centric applications to consume SAP Business Suite data through popular devices and platforms in an easy and standards-based fashion.

SAP Passport – Medium to transport technical data of a request from the client to the server. Used for collecting trace and reporting information for chains of requests (RFC, HTTP) across system borders.

Index

X

Xcode 5