Funambol Java SDK Developer Guide

Funambol Inc. java-scrum-team

Funambol Java SDK Developer Guide

by Funambol Inc. java-scrum-team

1.0.0

Abstract

Funambol Java SDK is a library meant to be used in applications that synchronize data on Java platforms.

Table of Contents

Chapter 1. Introduction

This documentation is a guide for developers that are willing to use the Funambol Java SDK to build applications. The Funambol Java SDK is a collection of components that can be used to build synchronization applications. There are several different modules in this library.

- common - contains platform abstraction classes and utilities
- sync - contains basic classes used by all available synchronization engines
- syncml - contains a full fledged SyncML stack
- sapisync - contains a full fledged synchronization stack based on Funambol SAPI technology
- pim - contains utilities for PIM handling (e.g. vcard parsers and formatters)
- client - contains high level components that can be reused to build a synchronization client

The library can be used on four different platforms:

- Java Standard Edition
- Android
- BlackBerry
- Java Micro Edition

Some parts of the library are only available on some platforms, but the core (syncml and SAPI engines) are available on all platforms. Building the library requires different tools and steps depending on the target platform. See here for building instructions.

Chapter 2. Library internals

This chapter describes some of the library internals and it is targeted for developers that need to understand how the library code is organized and works. Users willing to just use the library can safely skip this chapter.

Code Layout

This module contains code to abstract platforms and various utilities used across the library. The module does not depend on any library module and all the other modules depend on this one.

The code layout of this module is not straightforward as in mosto Java project. The reason is simply that this library is multiplatforms and therefore there are classes that depend on a specific platform and cannot be used on others. Therefore there is not a single source directory, but multiple directories. In order to build for a certain platform it is necessary to include the right mix of source files. Directories are structured this way:

- src/main/java - platform independent sources
- src/main/java-se - sources that work on all SE platforms (SE and Android)
- src/main/java-se-android - sources that work only on Android
- src/main/java-se-se - sources that work only on SE platform (not on Android)
- src/main/java-me - sources that work on all ME platforms (ME and BlackBerry)
- src/main/java-se-bb - sources that work only on BlackBerry
- src/main/java-se-me - sources that work only on ME platform (not on BlackBerry)

This layout gives the possibility to have several implementation for the same class. There are at least two interesting examples described below.

FileAdapter is an abstraction of a file. SE and ME have two different ways of handling files. The library provides two different implementation of FileAdapter, one for SE and the other for ME. The source code is stored in the java-se and java-me directories. The SE version is used both on Java SE and Android. Similarly the ME version is used both on Java ME and BlackBerry.

NetworkStatus is an abstraction that can be used to detect the network status. In this case there are four different implementations. Therefore there are four versions of this class, stored in the directories: java-se-android, java-se-se, java-me-bb and java-me-me.

The build for each platform is configured to pick the proper directories. For example the Android build uses the following three directories

- src/main/java
- src/main/java-se
- src/main/java-se-android

Common Module Description

This section describes the main features of the common module.

com.funambol.platform

This component contains classes aimed at abstracting basic platform dependent concepts. For example:

- FileAdapter - abstract files
- HttpConnectionAdapter - abstract http connections
- SocketAdapter - abstract socket connections

All these implementations have the same interface and can be used the same way on any platform.

com.funambol.util

This component contains classes aimed at providing utilities. For example:

- StringUtil - utilities to manipulate strings
- ConnectionManager - a factory for network connections (HttpConnectionAdapter and SocketAdapter)
- NetworkStatus - a utility to detect the network status

All these implementations have the same interface and can be used the same way on any platform.

com.funambol.storage

This component contains classes that abstract storage implementations.

- StringKeyValueStore - a key value store
- StringKeyValueStoreFactory - a factory for key value stores

On a given platform there can be several available implementations of StringKeyValueStore. The factory returns the default one, but users may decide to user others when convenient.

Sync Module Description

This section describes the main features of the sync module.

com.funambol.sync

This component contains classes which model basic synchronization concepts. These classes can be used by different synchronization engines and there is also an interface that models a synchronization engine. This interface is named SyncManagerI and it models a component able to perform synchronization. Such a component generates events that can be listened by a listener and it performs operations on the local storage via a SyncSource. A SyncSource for example allows an item (e.g. vCard) to be added to the local data base or to retrieve the list of contacts that changed since the last synchronization. The main concepts abstracted by this module are:

- SyncManagerI - is the abstraction of a synchronization engine
- SyncSource - is a class that allows a SyncManagerI implementation to modify a local storage

- SyncListener - is the listener of the synchronization process
- SyncReport - is the report generated at the end of a sync

This module cannot be used alone, but it provides basic concepts used by other modules (e.g. the SyncML or SAPI sync stack). For example an application can implement a SyncSource and then perform a synchronization of this source with different synchronization engines. Most of the application code is actually unaware whether the synchronization is carried via SyncML or SAPI, this is hidden in the library implementation. This abstraction provides a common framework that facilitate the usage of the library itself and also the user application code which does not need to know the details of the synchronization technology being used.

com.funambol.sync.client

This component contains classes useful to implement a SyncSource. A SyncSource is just an interface, but there are skeleton implementations that make it easier to create a new source and also provides a template to follow in the implemenation. The most important class in this category is the TrackableSyncSource which is a sync source that separates the concept of changes tracking from the rest of the its code meant to read/write the local storage.

SyncML Module Description

This section describes the main features of the syncml module.

com.funambol.syncml.protocol

This component contains classes that model the SyncML protocol. There is one class per SyncML item. For example there is a SyncML class that model the SyncML tag, a Data class for the Data tag on so on.

com.funambol.syncml.spds

This component contains the SyncML engine. This engine is capable of synchronizing data with a server using SyncML 1.2. The engine is data agnostic and it does not take care of exchanged items. It is only responsible for handling the synchronization protocol and sequence. The main features of this engine are listed hereafter.

- SyncML 1.2
- Basic and MD5 authentication
- Large objects support
- Device capabilities handling
- XML and WBXML support
- Suspend and resume support

The class SyncManager is the SyncML implementation of the interface SyncManagerI. This class is able to perform a SyncML synchronization. Users can plug their data handlers by implementing SyncSource(s). A SyncSource allows the SyncManager to execute SyncML commands. For example it allows an item (e.g. vCard) to be added to the local data base or to retrieve the list of contacts that changed since the last synchronization. The chapters that describe how to use the library provides more information on this topic.

com.funambol.syncml.client

This component contains classes useful to implement a FileSyncSource which represents files via the SyncML FileDataObject (this is the reason why such a SyncSource depends on the SyncML technology).

SapiSync Module Description

This section describes the main features of the sapisync module.

com.funambol.sapisync

This component contains an implementation of a synchronization engine based on Funambol SAPI technology. SAPI stands for Server API and they allow to get and modify the Funambol server status. See the SAPI guide for a complete description of this API. The SapiSync module has been designed and implemented to address media synchronization, altought the implementation is flexible enough to easily accomodate the sync of any type of data, as long as there is a server exposing the right APIs.

One major difference between the SyncML and the Sapi synchronization is related to where decisions are made. In SyncML decisions are taken by the server. In a SyncML synchronization the client blindly sends its items/changes to the server and then wait for server commands. It does not make any decision. A Sapi synchronization reverts this logic completely. All the logic is client side. The client is responsible to get information on what's available/changed on the server, and then decide what to do. Things like conflict resolution and twin detection are client responsability.

A user of the library may wonder if it is preferable to use the SyncML or the Sapi engine in his particular case. The two synchronization technologies have some fundamental differences that should make the decision fairly easy.

- Standardization - SyncML is a standard protocol, Sapi sync is a Funambol proprietary technology
- Availability - several SyncML servers and clients are available in the market. Sapi sync is only part of the Funambol commercial product (not the community version)
- Overall performance - Sapi sync has been designed with the goal of maximize the speed of exchaning large items
- Single item resume - Sapi sync implements resume at the level of synchronization session and single item. This means that if the exchange of a single item is interrupted, it can be resumed later. SyncML supports only resume of a synchronization session, but not at the level of single items.
- Device load - Sapi sync has all the synchronization logic client side. This implies that it needs a good hardware to run on, because it is more resources demanding than a SyncML engine. This is not a problem with today's smartphones and tablets, but it could be with older devices with very limited resources.

Sapi sync is not a protocol. There are REST APIs available on the server (SAPI) that clients can invoke in an arbitrary order to perform a synchronization. Our engine follows a flow described in the Sapi sync synchronization flow.

The class SapiSyncManager is an implementation of the interface SyncManagerI. This class is able to perform a synchronization via SAPI. Another important class in this module is the SapiSyncStatus which implements the SAPI SyncReport. The SapiSyncManager needs a SyncSource to complete a synchronization. SyncSource is part of the sync module and common to both SyncML and SapiSync engines. The only difference is that the SapiSync engines expectes all items exchanged with the source to be JSONSyncItem and not plain SyncItem. This is because the SapiSync engine is not completely data

agnostic. For example it expects each item to have a unique name, or an optional remote content (i.e. the item contains an URL to the content which is stored remotely). This is so because the engine needs information on the items to be able to perform the sync in an efficient way. Because of this a SyncSource working with the sapi engine works with the SyncML engine, but not the other way round. Even though it is possible to plug a source suitable for sapi sync into the SyncML engine, its behavior may be something different than what expected because the SyncML engine for example is not able to follow remote content links.

Altough the sources are not really interchangable, from a design standpoint it is important that all sync sources are represented by the same interface regardless of the engine used to sync. This simplifies the overall design and facilitate the learning of the API.

com.funambol.sapisync.source

This component contains classes useful to implement a SyncSource for the SAPI engine. The SAPI engine can work with any SyncSource, but for example all the Funambol media sync relies on data exchanged as JSON objects. This module contains a JSONSyncSource class that provides a basic implementation of a SyncSource that works with JSON items.

PIM Module Description

This section describes the main features of the PIM module. This module contains various utilities for handling PIM data. For example it contains parsers and formatters for vCard, vCal and iCal. All the parsers are implemented using JavaCC. These parsers decouple the parsing from the data model by using listeners. For example the vCard parser does not build a contact while parsing. It simply invokes a listener to notify when a new vCard begins or a field has been found. Formatters instead are bound to the data model. For example there is a formatter for contacts represented as JSR75 PIMItem (microedition standard model).

Since the module contains platform specific code, the directories layout is similar to the one used in the common module. It is just a bit simpler because there is not code that work only on a given platform, but it rather works on all standard edition (SE and Android) or ME ones (ME and BlackBerry).

Client Module Description

This section describes the main features of the client module. This module contains various blocks that can be used to create a rich synchronization application. In particular it contains blocks for the following features.

- Push - allows a client to receive push notifications when a sync is required for changes server side
- Updater - allows a client to check for new available versions of the product
- Source - models client sources. Each source is a sort of plugin that can be plugged into the client
- Configuration - handles the client configuration
- Sapi - handles sapi requests to the server
- Controller & UI - a set of components to create an application whose structure is like the standard Funambol clients
- Customization - a basic component that keeps track of the features that can be enabled/disabled in a client

- Localization - a basic interface that allows clients localization
- Engine - a wrapper around the SyncManager to setup/control the sync process
- Test - utilities to implement almost black box automatic tests

Modules dependencies

There are non circular dependencies among the API modules that may make a module mandatory when another one is being used. These are the dependencies:

- Common - does not any other library module
- Sync - depends on the Common module
- SyncML - depends on the Common and Sync modules
- SapiSync - depends on the Common and Sync modules
- Client - depends on the Common, Sync, SyncML and SapiSync modules

Chapter 3. How to do things

This chapter describes how to do things with the library. It covers the most typical actions required to get a working syncrhonization in place. From a practical point of view, the very first thing to do is to build the library. This is covered here.

How to trigger a SyncML synchronization

The synchronization engine is part of the syncml module and the entry point class is the SyncManager. The following code snippet shows how to setup and invoke the sync process.

```
import com.funambol.syncml.SyncManager;
import com.funambol.syncml.DeviceConfig;
import com.funambol.sync.SyncConfig;

// Creates the basic sync configuration
SyncConfig  config  = new SyncConfig();
// Set credentials
config.setSyncUrl("http://my.funambol.com/sync");
config.setUserName("foo");
config.setPassword("bar");

// Creates the basic device configuration
DeviceConfig devConfig = new DeviceConfig();
devConfig.setDevID("deviceid-xxxxxxx");

MySyncSource ss = new MySyncSource();

SyncManager manager = new SyncManager(config, devConfig);

try {
    manager.sync(ss);
} catch (Exception e) {
    Log.error(TAG_LOG, "Exception while synchronizing", e);
}
```

This example shows how to start the synchronization process. The whole process is triggered by the invokation of the *sync* method. The example also assumes that the user has his own SyncSource, named MySyncSource. The description of how a sync source can be created is in a subsequent section.

How to trigger a SAPI synchronization

The sapi synchronization engine is part of the sapisync module and the entry point class is the SapiSyncManager. The following code snippet shows how to setup and invoke the sync process.

```
import com.funambol.sapisync.SapiSyncManager;
import com.funambol.sync.SyncConfig;
import com.funambol.sync.DeviceConfigI;
```

```
// Creates the basic sync configuration
SyncConfig  config  = new SyncConfig();
// Set credentials
config.setSyncUrl("http://my.funambol.com/sync");
config.setUserName("foo");
config.setPassword("bar");

MyDeviceConfig devConfig = new MyDeviceConfig();

MySyncSource ss = new MySyncSource();

SapiSyncManager manager = new SapiSyncManager(config, devConfig);

try {
    manager.sync(ss);
} catch (Exception e) {
    Log.error(TAG_LOG, "Exception while synchronizing", e);
}
```

This example shows how to start the synchronization process. The whole process is triggered by the invokation of the *sync* method. The example also assumes that the user has his own SyncSource, named MySyncSource. The description of how a sync source can be created is in the next section. MyDeviceConfig is a simple implementation of the DeviceConfigI interface which provides some basic information about the device that perform the synchronization (such as a device id that the server can use to identify the device performing a synchronization).

How to Write a SyncSource

A sync source is the handler for the data to be synchronized. The SyncML stack is data agnostic and the source is responsible for data manipulation. For example a sync source for contacts will perform actions like:

- add one contact into the local database, given its vCard representation
- update one contact into the local database, given its vCard representation
- delete one contact from the local database
- retrieve the list of all existing items
- retrieve the list of items that has changed since the last sync

Altough these are the main functionalities provided, a SyncSource is actually required to do other things. We will see the other ones afterward, now we can describe some of the methods listed so far.

```
import com.funambol.sync.SyncSource;
import com.funambol.sync.SyncItem;
import com.funambol.sync.SyncException;

public class MySyncSource implements SyncSource {

    public Vector applyChanges(Vector items) throws SyncException {
        // items is a collection of SyncItem
```

```
        for(int i=0;i<items.size();++i) {
            SyncItem item = (SyncItem)items.elementAt(i);
            if (item.getState() == SyncItem.STATE_NEW) {
               String id = add(item);
               item.setKey(id);
            } else if (item.getState() == SyncItem.STATE_UPDATED) {
                updateItem(item);
            } else {
                deleteItem(item);
            }
        }
        return items;
    }
}
```

This first code snippet shows the method that the SyncEngine invokes when a bunch of commands are received from the server. First of all it is important to observe that the total set of items is usually broken into small subsets, according to the SyncML max msg size. Therefore the method applyChanges can be invoked several times during the same sync session. This method process the incoming items and based on their state (new,update,delete) triggers the appropriate action. In case of an add the SyncSource is responsible for setting the unique local identifier for the newly create item. This is done by setting the item key. In SyncML each item may have different ids on different devices. When a new item is received, the server attaches its GUID which is the server identifier for the item. Once the item has been locally created, the item has also a LUID which is the local identifier for the same item. The SyncML engine will send a mapping information to the server to inform it about the GUID/LUID association. The sync source is responsible for setting the item LUID and this is done via the setKey() method.

The implementation of the add method is also interesting. The code snippet here below describes it.

```
private String add(SyncItem item) throws SyncException {
    try {
        String vCard = item.getContent();
        Contact c = parseVCard(vCard);
        String id = storeContact(c);
        return id;
    } catch (Exception e) {
        throw new SyncException(...);
    }
}
```

This method contains several key concepts. Items received from the server are represented in a serialized form. In this case since we are talking about contacts, the item is likely to be represented by a vCard. Such a vCard is contained in the SyncItem. The vCard must be parsed and a Contact object that represents this item must be created. The Contact object is something which is not part of the synchronization process. It is actually most likely a platform dependent concept. For example in a BlackBerry environment contacts are represented by PIMItem (JSR75). The method parseVCard will then return a PIMItem that can be stored into the local address book. The Java SDK PIM module contains utility for parsing and processing items.

Another very interesting method is the getNextNewItem. This method is invoked by the manager to retrive the next new item to send to the server. The method shall return a SyncItem whose content is (at least in our example) a vCard representing a contact that was added after the last sync occurred. It is clear that this is the inverse of the add method, therefore it implies the ability to convert a local contact into a vCard (formatting). But even more interestingly it implies another sync source responsability: tracking changes. When a sync is about to start the sync source must have a mechanism to detect what changed since the

last syncheonization. The source must be able to provide added, updated and deleted items. Typically a sync source initializes the list of added/updated/deleted items when the beginSync method is first invoked (this is at the very beginning of the sync).

In order to support synchronization resuming a sync source must provide some basic support. In other words the engine does not provide a self contained support for resuming, but it requires the source to provide some basic support. A source that supports resuming shall implement the ResumableSyncSource interface, which has two different sets of methods:

- methods for synchronization session resuming, such as readyToResume, exists and hasChangedSinceLastSync
- methods for single items resuming, such as getItemSize and getLuid

If a source supports single items resuming, it must also support session resuming. The converse is not necessary, a source can support only session resuming and not single items resuming. Altough a source can support resuming at different levels, the overall support of the resume feature depends on the engine being used. For example the SyncML engine supports only session resuming, while the SAPI engine supports both session and single item resuming.

In a synchronization it is possible to have large objects to be exchanged. Large objects have a precise meaning in SyncML, but here we just mean objects whose footprint is significantly big. Suppose the client receives a large item from the server. It this were passed to the source in a single step, we would need the entire object to be kept in memory. This is not acceptable, therefore the source has the possibility to work in a different way. For every item received from the server, the method createSyncItem is invoked and it returns a new SyncItem to the engine. The SyncItem has a getOutputStream method that the server uses to get a valid stream and then it writes the item content as soon as a chunk of data is received. Once the item has been fully received, the addItem is finally invoked. If the sync source handles only small items, it can safely use the default implementation of SyncItem whose output stream is ByteArrayOutputStream, so the actual content is kept in memory and then stored in the add method.

How to write a sync source in practice

The previous section described what a sync source is and what are the main methods that need to be implemented. It is possible to implement the interface completely, but usually it is a lot more convenient to extend existing partial implementation. The TrackableSyncSource is meant to be a convenient class that can be extended by other sources. This source decouples the work of handling the local data store with the task of tracking changes. The source requires a ChangesTracker which is another interface that models an observer of a data store to track its changes. The following code shows a sync source implemented deriving a TrackableSyncSource.

```
import java.util.Map;
import java.util.HashMap;
import java.util.Enumeration;

import com.funambol.sync.SyncSource;
import com.funambol.sync.SourceConfig;
import com.funambol.sync.client.TrackableSyncSource;
import com.funambol.sync.client.ChangesTracker;

public MySyncSource extends TrackableSyncSource {
```

```
        private Map<String,String> items = new HashMap<String,String>();
        private int id = 0;

        public MySyncSource(SourceConfig config, ChangesTracker tracker) {
           super(config, tracker);
        }

        public int addItem(SyncItem item) throws SyncException {

            String newKey = "" + id++;
            items.put(newKey, item.getContent());
            item.setKey(newKey);

            // This needs to be invoked here, so that the TrackableSyncSource
            // can update the tracker status
            super.addItem(item);
            return SyncSource.SUCCESS_STATUS;
        }

        public void updateItem(SyncItem item) throws SyncException {
            String key = item.getKey();
            items.put(key, item.getContent());
            // This needs to be invoked here, so that the TrackableSyncSource
            // can update the tracker status
            super.updateItem(item);
            return SyncSource.SUCCESS_STATUS;
        }

        public void deleteItem(String key) throws SyncException {
            String key = item.getKey();
            items.remove(key);
            super.deleteItem(key);
            return SyncSource.SUCCESS_STATUS;
        }

        public void deleteAllItems() throws SyncException {
            items.clear();
            super.deleteAllItems();
        }

        protected Enumeration getAllItemsKeys() {
            return items.keys();
        }

        protected int getAllItemsCount() throws SyncException {
            return items.size();
        }

        protected SyncItem getItemContent(SyncItem item) throws SyncException {
            SyncItem ret = new SyncItem(item);
            ret.setContent(items.get(item.getKey()));
            return ret;
        }
    }
```

This is an example of a simple SyncSource that keeps its items in memory. As stated before, the sync engine sends commands in batch (via the applyChanged method) but the TrackableSyncSource serializes them and invoke the proper addItem, updateItem and deleteItem. If a source needs to process all the items together, then it can override the applyChanges method.

In order to instantiate MySyncSource it is necessary to create a ChangesTracker. The API provides a default implementation named CacheTracker that associates to each item in the SyncSource a fingerprint (by default MD5) and uses it to detect changes. An implementation is free to extend the CacheTracker by redefining the computeFingerprint method or to reimplement a ChangesTracker completely.

Are there sync sources ready to go?

The API has also a sync source which is ready to be used. This is the FileSyncSource which can be used to synchronize files. This is an extension of the TrackableSyncSource that uses FileAdapter to work on any platform. Using a CacheTracker allows to synchronize files with almost no effort.

```
SyncConfig  config  = new SyncConfig();
// Set credentials
config.setSyncUrl("http://localhost:8080/funambol/ds");
config.setUsername("foo");
config.setPassword("bar");
DeviceConfig devConfig = new DeviceConfig();
SyncManager manager = new SyncManager(config, devConfig);
SourceConfig sc = new SourceConfig(SourceConfig.BRIEFCASE, SourceConfig.FILE_OBJEC
sc.setEncoding(SourceConfig.ENCODING_NONE);

String sourceConfigFile = "briefcase.dat";

// If a configuration exists, then load it
try {
    FileAdapter ssConfig = new FileAdapter(sourceConfigFile);
    if (ssConfig.exists()) {
        sc.load(sourceConfigFile);
        ssConfig.close();
    }
} catch (IOException ioe) {
    System.err.println("Cannot load configuration");
}

String cacheFile = "briefcase.cache";
StringKeyValueFileStore ts = new StringKeyValueFileStore(cacheFile);
CacheTracker ct = new CacheTracker(ts);
FileSyncSource fss = new FileSyncSource(sc, ct, "./directory/");

try {
    manager.sync(fss);
    // Save the configuration
    sc.save(sourceConfigFile);
} catch (Exception e) {
    Log.error(TAG_LOG, "Exception while synchronizing", e);
}
```

Chapter 4. How to build

This chapter describes how the library can be built. The library can be succesfully built on Windows and Linux.

System Requirements

The system requirements depend on the target platform. Here below the requirements are detailed for each platform.

- For all platforms
  - Download and install Apache Ant: http://ant.apache.org/ (version 1.7 is required)
- For Java Micro Edition
  - Download and install Sun Java Wireless Toolkit: http://java.sun.com/products/sjwtoolkit/ (recommended version 2.5.2)
  - Download and install Apache Antenna: http://antenna.sourceforge.net/ (recommended version 1.0.2)
- For BlackBerry
  - Download and install Sun Java Wireless Toolkit: http://java.sun.com/products/sjwtoolkit/ (recommended version 2.5.2)
  - Download and install Apache Antenna: http://antenna.sourceforge.net/ (recommended version 1.0.2)
- For Java Standard Edition there are no additional requirements
- For Android
  - Download and install the Android SDK 2.0 or above: http://developer.android.com/sdk/index.html

Configuring the APIs

1. Unzip the package into your working directory
2. Open (create if needed) a build.properties file in ~/funambol/build/<platform>/

   where platform is:

   - me for Java MicroEdition
   - bb for BlackBerry
   - se for Java StandardEdition
   - android for Android

   [the symbol '~' refers to the user home directory. On Windows system this is generally "Documents and Settings/<user>". See the Java home.user variable for more details]
3. Fill the build.properties with the proper values, according to the platform for which the build is to be generated

- For Java Micro Edition
  - WTK home on your environment, wtk.home=<wtk-home> for example wtk.home=C:/wtk2.5.2
- For BlackBerry
  - Same settings as Java MicroEdition
  - The Blackeberry JDE home dir is required to build the API for BlackBerry; they are compliant with the API provided by RIM: bb.jdehome=<BB jde home dir> for example: bb.jdehome=C:/tools/Research In Motion/Blackberry JDE 4.2.1
- For Android
  - The Android SDK home dir is required sdk-folder=<Android SDK>
- For SE no settings are required

The APIs build on Windows and Linux. Building the BlackBerry version on Linux requires tweaking the RIM toolchain (a good starting point is: http://www.slashdev.ca/2008/04/03/blackberry-development-using-linux/).

Bulding the sources

Each module needs to be built separately at the moment. This is an example of the commands to be executed to build all modules on the Android platform.

1. cd common/build/android; ant
2. cd sync/build/android; ant
3. cd syncml/build/android; ant
4. cd sapisync/build/android; ant
5. cd pim/build/android; ant
6. cd client/build/android; ant

Each build generates a corresponding jar file into the output/android directory of each module.