Funambol
Developer's Guide
Version 8.7

Last revised: July 29, 2010

Table of Contents

1. Introduction

This document is intended for developers who aim to develop synchronization services based on the Funambol platform.

This development guide gives a deep insight of the server internals and design, providing guidance to anyone aiming to take advantage of the full range of possibilities that the platform provides.

1.1. Document structure

Chapters 2-9 present an overview of the Funambol architecture and internal workings, including a quick start guide to developing Funambol connectors. Chapters 11-13 describe in detail the SyncSource, Officer and Web Services APIs. Chapter 14 explains how to localize Funambol Clients. Chapter 15 is a short introduction to the Funambol SDK and chapter 16 presents the Funambol Device Simulator tool.

1.2. Audience

This document is addressed to anybody wanting to extend the Funambol platform or simply looking for detailed information on the Funambol architecture.

1.3. Funambol licensing

All Funambol software and software developed using Funambol APIs or SDKs are licensed under AGPL V3 (Affero General Public License) unless separate arrangements have been made with Funambol to reach an explicit commercial agreement.

For more information on AGPL V3, see [10].

1.4. Comments and feedback

The Funambol team wants to hear from you! Please access our community portal at https://www.forge.funambol.org/participate and submit your questions, comments, feedbacks or testimonials.

2. Funambol development

The following sections present several concepts related to how Funambol can be used to develop mobile applications. Before digging into the details of Funambol development, it is useful to describe some basic concepts of data synchronization since this is the basis for many Funambol applications and services.

2.1. Data synchronization

All mobile devices – handheld computers, mobile phones, pagers, laptops – need to synchronize their data with the server where the information is stored. This ability to access and update information on the fly is key to the pervasive nature of mobile computing. Yet, today, almost every device uses a different technology for performing data synchronization.

Data synchronization is helpful in respect to many areas:

- Propagating updates between a growing number of applications
- Overcoming the limitations of mobile devices and wireless connections
- Maximizing user experience minimizing data access latency
- Keeping scalability of the infrastructure in an environment where the number of devices (clients) and connections tends to increase considerably
- Understanding the requirements of mobile applications, providing a user experience that helps and is not an obstacle for mobile tasks

Data synchronization is the process of making two sets of data look identical (Figure 1). This involves many concepts, the most important are:

- ID handling
- Change detection
- Modification exchange
- Conflict detection
- Conflict resolution
- Slow and fast synchronization

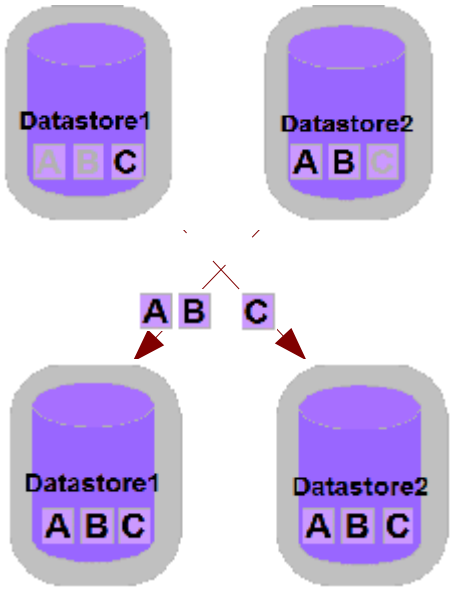


Figure 1: Data synchronization process

2.1.1. ID handling

At a first look, ID handling seems a pretty straightforward process and of no interest. Instead, ID handling is an important aspect of the synchronization process and it is not trivial. Each piece of data is usually uniquely identifiable by a subset of its content fields; for example, in the case of a contact entry, the concatenation of first name and last name uniquely selects an entry in your directory. In other cases, the ID is represented by a particular field specifically introduced for that purpose. This may be the case, for example, of a Sales Force Automation mobile application, where an order is identified by an order number or reference. The way an item ID is generated is not determinable a priori and it is application and device specific.

In an enterprise system, however, data is stored in a centralized database, shared by all users; each single item is known by the system with a unique global ID. In some cases, two sets of data (i.e. the order on the client and the order on the server) represent the same information (*the* "order" made by

the customer) but they differ. What could be done to reconcile client and server IDs in order to make the information consistent? Many approaches can be chosen:

- Clients and server agree on a *ID scheme* (a convention on how to generate IDs must be defined and used);
- Each client generates globally unique IDs (GUIDs) and the server accepts client-generated IDs;
- The server generates globally unique IDs (GUIDs) and each client accepts those IDs;
- Client and server generate their own IDs and a mapping is kept between the two. Client side IDs are called Local Unique IDentifiers (LUID) and server side IDs are called Global Unique IDentifiers (GUID). The mapping between local and global identifiers is referred as LUID-GUID mapping.

2.1.2. Change detection

Change detection is the process of identifying the data that was modified after a particular point in time (i.e. the last synchronization). This is usually achieved making use of additional information such as timestamps and state information. For example, a possible database enabled for an efficient change detection is the one shown in Table 1.

ID	***First name***	***Last name***	***Telephone***	***State***	***Last_update***
12	John	Doe	+1 650 5050403	N	2008-04-02 13:22
13	Mike	Smith	+1 469 4322045	D	2008-04-01 17:32
14	Vincent	Brown	+1 329 2662203	U	2008-03-21 17:29

Table 1 - A database enabled for efficient change detection

However, sometimes legacy databases do not provide the information needed to accomplish an efficient change detection. Therefore, the matter becomes more complicated and alternative methods must be adopted (based on content comparison, for instance).

2.1.3. Modification exchange

A key component of a data synchronization infrastructure is the way modifications are exchanged between client and server. This involves the definition of a synchronization protocol that client and server have to use to initiate and carry on a synchronization session. In addition to the exchange modification method, a synchronization protocol must also define a set of supported modification commands. The minimal set of modification commands is represented by the following:

- Add
- Replace
- Delete

2.1.4. Conflict detection

Let's suppose two users synchronize their local contacts database with a central server in the morning, before going to the office. After syncing, they have exactly the same contacts on their PDAs. Let's now suppose that they change the telephone number of the same “John Doe” entry, but for some reason with a different number (maybe, one of the two made a mistake). What will happen when the next morning they will synchronize again? Which one of the two new versions of the John Doe record should be taken and stored into the server? This condition is called “conflict” and the server has the duty of identifying and resolving it.

The simplest way to do detect a conflict is by the means of a synchronization matrix (see Table 2).

Database A → *↓ Database B*	*New*	*Deleted*	*Updated*	*Synchronized/Unchanged*	*Not Existing*
New	C	C	C	C	B
Deleted	C	X	C	D	X
Updated	C	C	C	B	B
Synchronized/Unchanged	C	D	A	=	B
Not Existing	A	X	A	A	X

Table 2 - The synchronization matrix

Because both users synchronize with the central database, we can consider what happens between the server database and one of the client databases at a time: let's call Database A the client database and Database B the server database. The symbols in the synchronization matrix have the following meaning:

- **X** : nothing to do
- **A** : item A replaces item B
- **B** : item B replaces item A
- **C** : conflict
- **D** : delete the item from the source(s) containing it

2.1.5. Conflict resolution

Once a conflict arises and it is detected, a proper action must be taken. Different policies can be applied:

- User decides: the user is notified of the conflict condition and decides what to do; this strategy, like the following “Client wins” is a bit problematic in a server centric synchronization solution: each user may have the same right to modify an item and one user could not be able to decide whether his/her modification should win over the other ones
- Client wins: the server silently replaces conflicting items with the ones sent by the client
- Server wins: the client has to replace conflicting items with the ones from the server
- Timestamp based: the last modified (in time) item wins
- Last/first in wins: the last/first arrived item wins
- Do not resolve

2.1.6. Full and fast synchronization

There are many modes to carry on the synchronization process. The main distinction is between fast and full synchronization. Fast synchronization involves only the items changed since the last synchronization between two devices. Of course, this is an optimized process that relies on the fact that, some time in the past, the devices where fully synchronized; this way, the state at the beginning of the sync operation is well known and sound. When this requisite is not met (because, for instance, the mobile device has been reset and lost the timestamp of the last synchronization), a *full synchronization* must be performed. In this case, the client sends its entire database to the server, which compares it with its local database and returns the modifications that must be applied to be up to date again.

Both fast and slow synchronization modes can be performed in one of the following manners:

- Client to server: the server updates its database with client modifications, but sends no server-side modifications.
- Server to client: the client updates its database with server modifications, but sends no client-side modifications.

- Two-way: client and server exchange their modifications and both databases are updated accordingly.

3. Funambol architecture

3.1. System architecture

The system architecture of a Funambol deployment includes the logical components illustrated in Figure 2. Note that, even though each logical component is represented as a single box in the figure, all of them can be deployed in a redundant configuration to increase availability and share load.

3.1.1. Roles and responsibilities

This section describes the roles and the main responsibilities of the components illustrated in Figure 2.

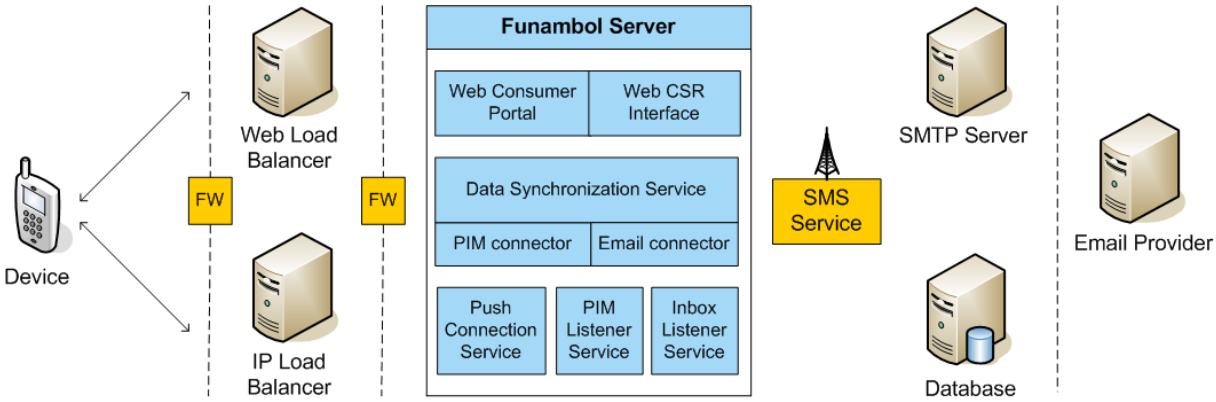


Figure 2: Funambol system architecture

Device

A device can be any physical device or client software application that can communicate with the Funambol Server via SyncML for PIM synchronization or TCP/IP for push email. Examples of such devices are:

- mobile phones with a native SyncML client
- mobile phones with a Java ME platform fulfilling basic requirements, running the Funambol Java ME Email Client
- Windows Mobile phones, running the Funambol Windows Mobile Sync Client
- desktop devices running the Funambol Outlook Sync Client and/or the Funambol iPod Sync Client
- desktop devices running the Community Sync Client
- devices running other Funambol Sync Clients (e.g. iPhone, BlackBerry, etc...)

Devices are the main interface through which users access Funambol. Responsibilities of the device/client software include:

- providing the user UI
- initiating the communication with the server
- hosting the local data (PIM/Email, ...)
- collecting/detecting the change log

The communication between the device and the Funambol Server is based on the TCP/IP protocol.

Web load balancer

SyncML is an application protocol transported over HTTP. This means that SyncML requests can be considered common HTTP traffic. A web load balancer (see Figure 2) can therefore be used to balance the incoming load amongst different nodes of a Data Synchronization Service cluster. In this respect the nodes in the cluster all perform the same function and can be used interchangeably for each SyncML request.

The main responsibilities of the HTTP load balancer include:

- providing the front-end of the Funambol system
- distributing the SyncML requests amongst the nodes of the Data Synchronization Service cluster
- detecting failures on the nodes of the cluster, redirecting traffic to the active nodes if one of the nodes fails

Note: the HTTP load balancer is not provided as part of the default installation or deployment. Many different solutions, both hardware and software, can be adopted and any organization may have different best practices already in place. A common solution is to use Apache + mod_jk.

IP load balancer

Unlike SyncML, the protocol used by connection-oriented push is not transported over HTTP but it is a pure TCP/IP communication. Since the number of connections that connection-oriented push must support is generally very high, an IP load balancer (see Figure 2) is usually needed. This component works similarly to the web load balancer, but at the Transport level (Layer-4 load balancing).

The main responsibilities of the IP load balancer include:

- providing the front-end of the Funambol system for connection-oriented push requests
- distributing connection-oriented push requests amongst the nodes of the Push Connection Service cluster
- detecting failures on the nodes of the cluster, redirecting traffic to the active nodes if one of the nodes fails

Note: the IP load balancer is not provided as part of the default installation or deployment. Many different solutions, both hardware and software, can be adopted and any organization may have different best practices already in place. A common solution is to use Linux Virtual Server.

Funambol Server

The Funambol Server is the core of the Funambol push-email service and of PIM synchronization. As illustrated in Figure 2, it comprises several components, detailed in the following sections.

Data Synchronization Service

The role of the Data Synchronization (DS) Service is to provide the synchronization services and to communicate directly to the devices using SyncML. The main responsibilities of the Data Synchronization Service are:

- hosting the synchronization engine
- accepting and serving synchronization requests
- handling low level device information
- synclet technology
- providing an interface towards the back-ends
- providing a remote administration interface
- providing connection-less push

Note: the Data Synchronization Service is the only service that Funambol Server cannot run without.

Email connector

The email connector plays a key role in the Funambol architecture, since it allows the server to communicate with different email servers (note that each user can be attached to a different email server). The email connector consists of two main components: the connector itself and the Inbox Listener Service.

The email connector is deployed together with the synchronization engine. It has the following responsibilities:

- searching the user email cache for which messages should be downloaded on the client
- filtering out unwanted messages or content (e.g., retrieve headers only)
- processing emails
- filtering emails
- sending outgoing messages

PIM connector

This connector is the counterpart of the email connector for PIM synchronization. The PIM connector consists of two main components: the connector itself and the PIM Listener Service.

The PIM connector is deployed together with the synchronization engine. It has the following responsibilities:

- searching the user email cache to determine the messages to be downloaded onto the client
- using various techniques to enable the user to filter out unwanted emails (e.g. Retrieve and sync headers only)
- processing emails
- sending outgoing emails

Portal

The Funambol Portal implements the main interface through which users and administrators interact with the Funambol platform over the Internet. The portal component consists of:

- a **web-based consumer portal**, through which users can sign up for the service, access their data and profile, manage their contacts and calendar, setup their mobile device and email account
- a **web-based customer service representative (CSR) interface**, which allows an operator to access users information and perform maintenance of user accounts

Push Connection Service

The Push Connection Service is a separate process from the Data Synchronization Service and Portal process and is responsible for the implementation of the connection-oriented push technology.

The main responsibilities of the Push Connection Service are:

- accepting and keeping open connection-oriented push connections from devices
- delivering push notifications to the attached devices

PIM Listener Service

The PIM Listener Service is a separate process from the Data Synchronization Service and Portal process; it has the following responsibilities:

- polling the user PIM database regularly to check for updates
- triggering an action to the Data Synchronization Service if the user has PIM changes

Inbox Listener Service

The Inbox Listener Service is a separate process from the Data Synchronization Service and Portal process; it has the following responsibilities:

- creating and maintaining the user inbox cache
- polling the user inbox regularly to check for new emails
- updating the user inbox cache
- triggering an action to the Data Synchronization Service if the user has new email

Note: the user inbox cache does not contain any sensitive information, but only information about when a message has been received. This is necessary in order to filter out the less recent emails during email download.

SMS Service

This is the service used to send SMS messages to user devices. The Funambol platform uses an external SMS gateway for this, which translates the HTTP-based messages sent by the server into SMS messages, and injects them into the network servicing the target user.

SMS messages are used to:

- send users the download link for Funambol clients
- perform OTA configuration of SyncML settings directly on the user device
- start a synchronization in case TCP-based push is not possible (SMS push)

Note: the SMS Service is not provided out of the box. Funambol, by default, supports SubitoSMS. Other SMS service providers are easily configurable.

SMTP server

This is the server used by Funambol Carrier Edition to send emails to external recipients.

Note: this server is used for service related emails (e.g. invitation or activation emails) and to send user emails for users who are not using a public email service.

Database

This is the database server. Funambol supports the following database systems:

- PostgreSQL
- MySQL (requires Funambol version 7.0 or later)
- Hypersonic (only on Funambol Community Edition)

Email provider

This component is not really part of the Funambol software; it represents the users' email servers. Currently, the supported mail protocols are:

- IMAP
- POP
- Google IMAP
- AOL IMAP

3.2. The synchronization engine

The following sections provide more details on a particular component, the Data Synchronization Service, and describe the architecture of the synchronization engine, which is a key component that can be extended to implement specific needs.

The synchronization engine is the component that implements the synchronization logic; this means:

- identify the sources and the destinations of the data sets to be synchronized
- identify the data that needs to be updated/added/deleted
- determine how updates must be applied
- detect conflicts
- resolve conflicts

In other words, the synchronization engine is the core of any data synchronization server. The basic framework interfaces and classes are grouped in the package *sync4j.framework.engine*.

3.3. Execution flow of a request

The execution flow of an OMA DS request is illustrated in Figure 3.

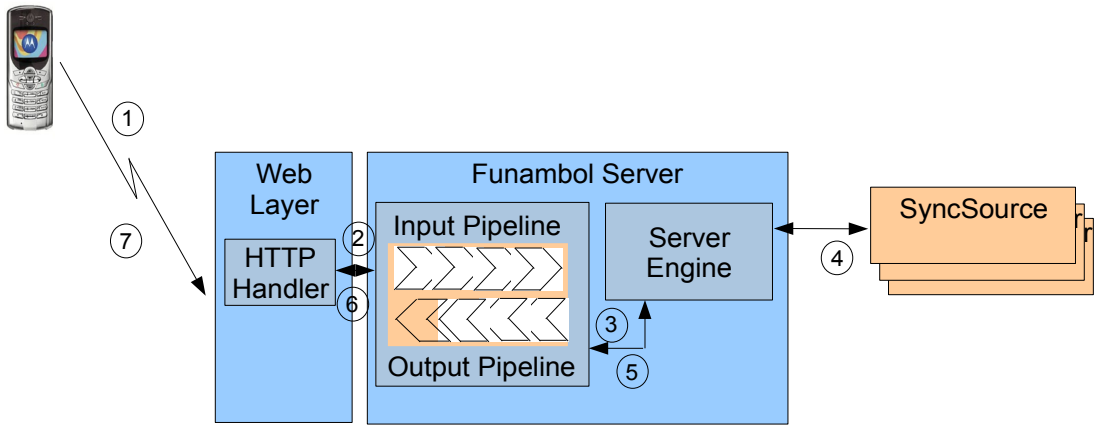


Figure 3: Execution flow of an OMA DS request

A synchronization session starts with the client device sending a first SyncML message to the server. The request then follows the flow described below:

1. When a new request comes from the client, the HTTP handler takes care of it. After some processing, for example the transformation of the binary message into a more manageable form or the association of the incoming message to an existing synchronization session, the HTTP handler passes the request to the synchronization server.
2. The message first goes through the input message processing pipeline (see later) according to the application needs.
3. The manipulated message comes out of the input pipeline and goes into the server engine for the synchronization processing.
4. When needed, the server engine calls the services of the external (and custom) SyncSources in order to access the real data stores.
5. After processing the incoming message, the server engine builds the response message, which goes through the output message processing pipeline for post-processing.
6. The response message is then returned to the HTTP handler, which packs the SyncML message into the HTTP response and sends it back to the device.

4. The synchronization process

The synchronization process is logically accomplished in three steps:

1. Preparation
2. Synchronization
3. Finalization

The Funambol engine goes through these steps coordinating their execution, but delegates most of the synchronization logic to an auxiliary class, implementation of the *SyncStrategy* interface.

There are many types of synchronization; the ones specified by the SyncML protocol are:

Sync Type	***Description***
Two-way sync (fast)	A normal sync type in which the client and the server exchange information about modified data in these devices. The client sends the modifications first.
Slow sync	A form of two-way sync in which all items are compared with each other on a field-by-field basis. In practice, this means that the client sends all its data from a database to the server and the server does the sync analysis (field-by-field) for this data and the data in the server.
One-way sync from client only	A sync type in which the client sends its modifications to the server but the server does not send its modifications back to the client.
Refresh sync from client only	A sync type in which the client sends all its data from a database to the server (i.e., exports). The server is expected to replace all data in the target database with the data sent by the client.
One-way sync from server only	A sync type in which the client gets all modifications from the server but the client does not send its modifications to the server.
Refresh sync from server only	A sync type in which the server sends all its data from a database to the client. The client is expected to replace all data in the target database with the data sent by the server.
Server Alerted Sync[1]	A sync type in which the server to alerts the client to perform sync. That is, the server informs the client to starts a specific type of sync with the server.

Table 3 - Sync modes defined by SyncML

The first two are the most important, since the others are derivation of slow and fast sync modes.
In a slow synchronization, the client sends all its items to server, which compare them with the server database and then it sends back the modification that the client has to apply in order to be in sync again. In the case of slow sync, the sources to be synchronized must be fully compared in order to reconstruct the right image of the data on both synchronization endpoints. The way the sets of items are compared is implementation specific and can vary from comparing just the item keys or the entire content of an item.

In a two-way fast synchronization, a data source is queried only for new, deleted or updated items since a given point in time (and for a given user). In this case, the status (deleted/updated/new) and the modification timestamp of the items can be checked in order to decide when a deeper comparison is necessary.

The following sections describe in more detail each phase of the synchronization process and other key aspects of the synchronization engine architecture.

1 The SyncML specification does not say anything about how the server alerted sync should be achieved, therefore each product can implement it in a different and not interoperable way. As per nowadays, only few devices are known to support this feature.

4.1. Preparation

The preparation phase is the process of analyzing the differences between two or more sources of data (called *SyncSource*s) with the goal of obtaining a list of sync operations that applied to the sources involved in the synchronization, will make the databases look identical (Figure 4).

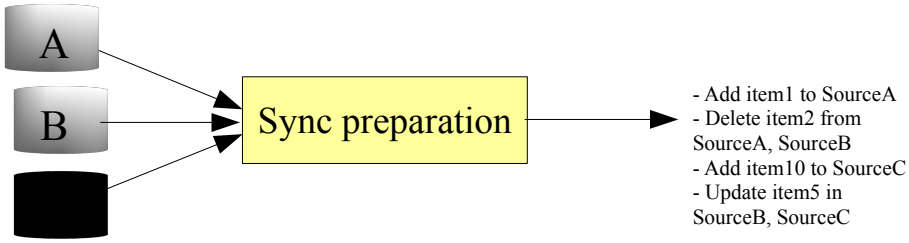


Figure 4: Preparation phase

4.2. Modifications detection

Modifications detection is based on the sets of items represented in Figure 5, applying the modifications matrix of Table 4. You can think of A as the client data source and B as the server data source.

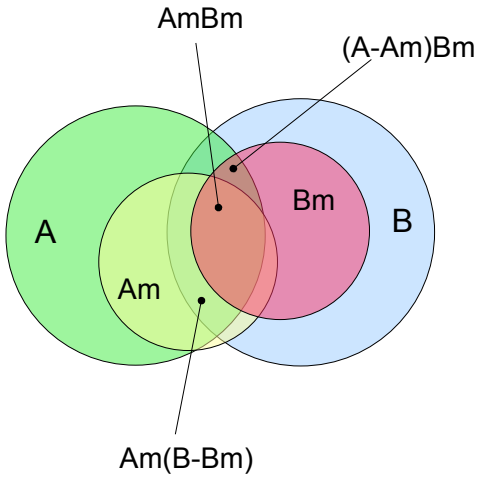


Figure 5: Synchronization items sets

- **A** – Items belonging to source A (as known via LUID-GUID mapping)
- **B** – Items belonging to source B
- **Am** – Modified items belonging to source A
- **Bm** – Modified items belonging to source B
- **AmBm** – Items modified both in source A and B (intersection between Am and Bm)
- **(A-Am)Bm** – Items unmodified in A, but modified in B
- **Am(B-Bm)** – items unmodified in B, but modified in A

Note that A is the server view of the A source: it contains the items mapped in the server as they are defined in the LUID-GUID mapping. If, for example, the client sends a new item that has never been mapped, this item will be in Am, but not in A. In order to be sure that the new item is not equal to some existing item in B, it must be looked up in B. If an item in B represents the same item as in Am, A is virtually augment of such item, so that at the end, Am will be a sub-set of A.

Another important aspect to point out is that the entire data sets A and B can be considerably big. Therefore, when possible, it is important to deal with the smallest possible sets of items instead of doing a full item-per-item comparison.

The preparation phase is slightly different depending on the type of the synchronization. In the case of a slow synchronization, all items in the sources must be compared looking for differences that will be translated into synchronization operations. This type of process does not depend on previous synchronizations and, in fact, it is used to fully recreate a database as if no synchronizations have ever taken place. This is achieved resetting the LUID-GUID mapping before starting the modification detection process.

On the contrary, when a fast synchronization is performed, it is assumed that the involved sources rely on a previous data synchronization, so that only the changes since the time of the last synchronization need to be considered.

The algorithm used in the preparation phase is as follows:

Given the sources to be compared, suppose A and B, the goal of the algorithm is to produce an array of operations, in which each element represents a particular synchronization action, i.e. create the item X in the source A, delete the item Y from the source B, etc. Sometimes, it is not possible to decide the action to perform, thus a conflict operation is generated. A conflict might be solved by something external the synchronization process, for instance by a user action. In order to create the operation set, each item in the source A is compared with each item in the source B (to be intended as the selected items depending on the synchronization type).

To determine which operation should be generate the synchronization matrix defined in Table 4 is used.

Database A → ***↓ Database B***	***New***	***Deleted***	***Updated***	***Synchronized/Unchanged***	***Not Existing***
New	C	C	C	C	B
Deleted	C	X	C	D	X
Updated	C	C	C	B	B
Synchronized/ Unchanged	C	D	A	=	B
Not Existing	A	X	A	A	X

Table 4 - Synchronization matrix

Where:

- **A** : item A replaces item B
- **B** : item B replaces item A
- **C** : conflict
- **D** : delete the target item
- **X** : do nothing

When a client item should be updated or inserted on the server, but the server does not have a mapping for it, a deeper comparison must be performed. In fact, the new/updated item could have the same content of an existing item on the server; this could even turn into a conflict. For example, suppose that a client tries to insert a new appointment with ID “xyz” at 20041029T1400Z in the meeting room “OceanSide”. Even if there is no matching item on the server, if “OceanSide” is already busy at the same time, this could be considered a conflict.

In order to detect such situations, the synchronization engine will ask for items similar to the one that is trying to add/update. Those similar items are called twins. Note that we used by choice similar and not equal. This is because how much an item should look like an existing item in order to be considered a twin may be implementation specific. Each source should be able to find twin items accordingly to its own logic.

4.3. Synchronization

The synchronization step is the phase where the operations prepared in the previous step are executed. Executing an operation means applying the required modification to the involved SyncSources. This is done by the synchronization engine.

4.4. Finalization

The third and last step is intended for cleaning up purposes. In addition, usually in this phase the client sends the LUID-GUID mapping resulted in the synchronization just performed.

5. Extending Funambol

The Funambol platform can be extended in many areas in order to integrate Funambol into existing systems and environments. Figure 6 illustrates the most common integration user cases, and the Funambol modules involved:

- **Officer**: integrating with an external authentication and authorization service;
- **SyncSource**: integrating with an external data source
- **Synclet**: adding pre or post processing to a SyncML message
- **Admin WS**: integrating with an external management tool

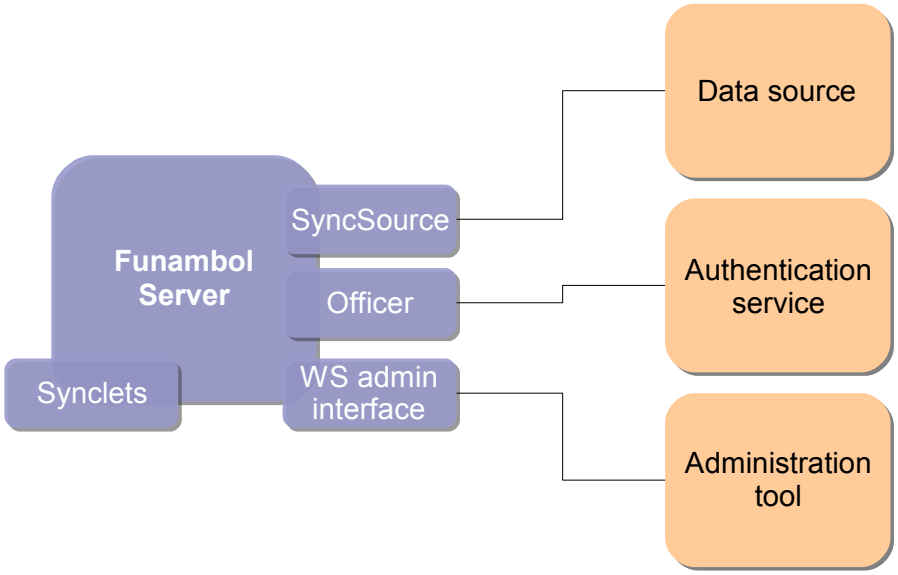


Figure 6: Integration components

Funambol extension are distributed and deployed as Funambol modules. This section describes the structure of a Funambol module, while the following sections detail each of the scenarios above.

A Funambol module represents the means by which developers can extend the Funambol Server. A module is a packaged set of files containing classes, installation scripts, configuration files, initialization SQL scripts, components and so on, used by the installation procedure to embed an extensions into the server core.

For more information on how to install Funambol modules see [3].

5.1. Building a Funambol module

A Funambol module is a zip package named following the convention:

```
<module-name>-<major-version>.<minor-version>.s4j
```

Where *<module-name>* is the name of the module without spaces and with small caps only and *<major/minor-version>* are the major and minor version numbers. A new version of a module with minor version number change must be backward-compatible, while changes in the major version number imply that a migration may be required.

The package must have the structure illustrated in Figure 7.

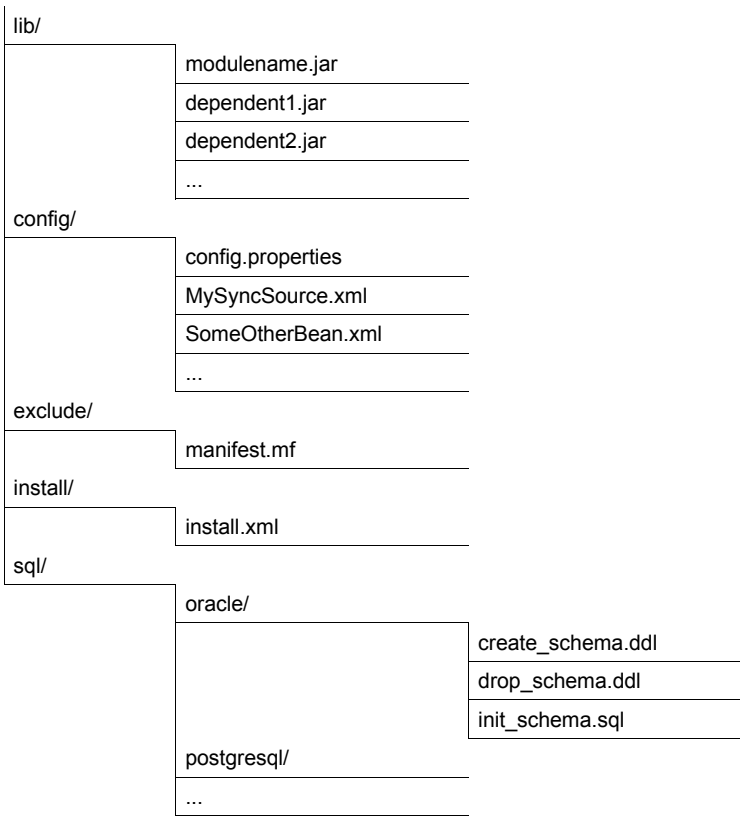


Figure 7: Module package structure

In the following, entries ending with a '/' represent directories and filenames in italic are given just as examples (in a real package they will be replaced with real filenames).

The module classes are packaged in a main jar file called <*modulename*>*.jar*.

Configuration files are stored under the package directory *config*, creating subdirectories as needed.

Note: even if it is not mandatory, usually SyncSource instance configuration files are stored under a subtree in the form <*module-id*>/<*connector-id*>/<*sourcetype-id*>, which is the convention used by the Funambol Administration Tool when creating a new SyncSource instance.

The directory *install* contains *install.xml,* an Apache ant script called when the module is installed; this is the hook where a module developer can insert module specific installation tasks. Installation specific files can be organized in subdirectories under *install*.

If the module requires a custom database schema, the scripts to create, drop and initialize the database are stored under the *sql/*<*database*> directory, where <database> is the name of the DBMS as listed in the *install.properties* file.

Finally, the *exclude* directory is used to store files that will be temporarily used by the installation procedure, but that will not be (automatically) copied. These can be used by the module installation script for any purpose.

5.2. Modules, connectors, listeners and SyncSource types

As already mentioned, a module is a container for anything related to one or more server extensions which are used by the engine to communicate and integrate with external systems. These extensions are usually specific to the backend that must be integrated. A specific case of such extension is when the main purpose is to connect to an external data source, in which case the module is called *connector.* In other words, a connector is an extension of the server, intended to support the synchronization of a particular data source.

In order to access the data source, the connector must provide a so called “SyncSource type”. A SyncSource type represents the template from which an instance of a SyncSource can be created. For example, the *FileSystemSyncSource* type made available by the Funambol is the means used by the server to synchronize data stored in the file system. However, it does not represent a particular

directory to synchronize; to synchronize a specific directory (for instance */data/contacts*) a real SyncSource instance must be created and configured with the desired directory. You can think of a SyncSource type as a class and of a SyncSource as an instance.

An additional (but optional) component that a connector can provide is called *listener*. This component is in charge to detect changes in the backend so that the server can trigger a device to synchronize the changes.

Note: Funambol provides out-of-the-box a module called Foundation that contains basic functionalities, libraries and classes that all custom modules require. This module must not be removed by any Funambol installation.

5.2.1. Registering modules, connectors and SyncSource types

Modules, connectors and SyncSource types are registered filling the following database tables:

- *fnbl_module* for module information
- *fnbl_connector* for connector information
- *fnbl_sync_source_type* for SyncSource type information
- *fnbl_connector_source_type* for connector-SyncSource type associations
- *fnbl_module_connector* for module-connector association

Note: The last two tables are used to create the hierarchy module-connector-SyncSource type that you can see in the Funambol Administration Tool.

As an example, let's consider the foundation module registration. When Funambol is installed, the foundation module is installed too. It brings a connector called *FunambolFoundationConnector*, which, in turn, contains the SyncSource types *PIM Contact SyncSource*, *PIM Calendar SyncSource*, *FileSystem SyncSource* and *SIF SyncSource* (Figure 8).

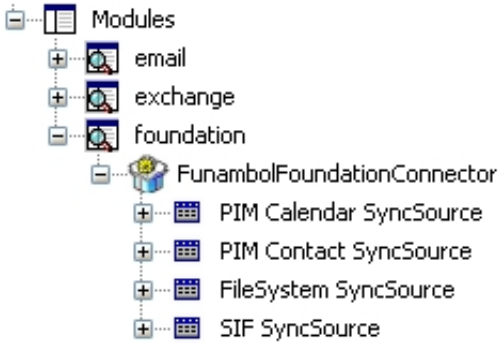


Figure 8: Foundation Connector Module in the Administration Tool

This hierarchy is obtained with the following SQL commands:

1. Module registration:

```
insert into fnbl_module (id, name, description)
values('foundation','foundation','Foundation');
```

2. SyncConnector registration:

```
insert into fnbl_connector(id, name, description)
values('foundation','FunambolFoundationConnector','Funambol Foundation Connector');
```

3. The Foundation Connector belongs to the foundation module:

```
insert into fnbl_module_connector(module, connector)
values('foundation','foundation');
```


4. The SyncSource Type registration:

```
insert into fnbl_sync_source_type(id, description, class, admin_class)

values('contact-foundation','PIM Contact
SyncSource','com.funambol.foundation.engine.source.PIMContactSyncSource','com.funambol
.foundation.admin.PIMContactSyncSourceConfigPanel');

insert into fnbl_sync_source_type(id, description, class, admin_class)

values('calendar-foundation','PIM Calendar
SyncSource','com.funambol.foundation.engine.source.PIMCalendarSyncSource','com.funambo
l.foundation.admin.PIMCalendarSyncSourceConfigPanel');

insert into fnbl_sync_source_type(id, description, class, admin_class)

values('fs-foundation','FileSystem
SyncSource','com.funambol.foundation.engine.source.FileSystemSyncSource','com.funambol
.foundation.admin.FileSystemSyncSourceConfigPanel');

insert into fnbl_sync_source_type(id, description, class, admin_class)

values('sif-fs-foundation','SIF
SyncSource','com.funambol.foundation.engine.source.SIFSyncSource','com.funambol.founda
tion.admin.SIFSyncSourceConfigPanel');
```

5. Finally, the SyncSource type belongs to the Foundation Connector:

```
insert into fnbl_connector_source_type(connector, sourcetype)

values('foundation','contact-foundation');


insert into fnbl_connector_source_type(connector, sourcetype)

values('foundation','calendar-foundation');


insert into fnbl_connector_source_type(connector, sourcetype)

values('foundation','fs-foundation');


insert into fnbl_connector_source_type(connector, sourcetype)

values('foundation','sif-fs-foundation');
```

Note: two classes are specified for each SyncSource type registration: the class (for instance *com.funambol.foundation.engine.source.FileSystemSyncSource*), which actually implements the SyncSource interface and the *admin_class*, which instead is used to create a new SyncSource instance and to configure it in the Funambol Administration Tool.

In the following section, we will see how these two classes are developed.

Note: in this guide, SyncSource and SyncSource Type are often treated as synonyms, even if they are in the template-instance relationship seen before.

6. Getting started on connector development

6.1. Introduction

This chapter describes how to create a connector that extends the functionality of the Funambol Data Synchronization (DS) Server.

As better described in chapter 5, a Funambol extension is delivered in the form of a module, which consists of a packaged set of files, including classes, configuration files, server beans and initialization SQL scripts. All these contents are deployed into the Funambol Data Synchronization Service to provide access to a specific back-end (e.g. a database, a REST based service, a web services API, etc.). In general, a module can be viewed as a container for anything related to server extensions. When a module provide access to a specific backend, it is called *connector*.

The following terms and concepts will be used in this document:

Module: a container for anything related to one or more server extensions which are used by the engine to integrate with external systems.

Connector: a particular type of module, with the purpose of connecting to an external data source; in other words, a connector is an extension of the server intended to support the synchronization of a particular data source.

SyncSource: a key component of a connector that defines the way a set of data is made accessible to the Funambol Data Synchronization Service for synchronization. A SyncSource type represents the template from which an instance of a SyncSource can be created. For example, the *FileSystemSyncSource* type defines how data stored in the file system can be accessed by the Funambol Data Synchronization Service; however, it does not represent a specific directory to be used for synchronization, and in order to synchronize a specific directory an instance of *FileSystemSyncSource* must be created and configured with the desired directory.

Synclet: a pre or post processing unit that can process a message before it gets into the synchronization engine or just after it is going out from it.

This chapter will guide you through the development, packaging, installation and testing of a module. The module contains a simple SyncSource and Synclet which produce some logging. Once you are familiar with this tutorial you can see real-world examples like the OpenXchange connector [8] or the Exchange connector [9]. Plus, many people have developed many modules and connectors that are available to the public. See [5] for more information.

6.2. Getting started

The following connector development quick-start section assumes a working knowledge of Java, Maven and SQL.

For more detailed information about Funambol development see the next sections of this developer's guide.

In order to follow this guide you need:

- Funambol Data Synchronization Service installed and running
- Funambol Software Development Kit 8.0.x [7]
- Java 2 SDK version 1.5.x or above
- Apache Maven [4]

Optionally, you may want to download a Maven plug-in for your preferred IDE (see http://mevenide.codehaus.org).

Download the software and install it in a directory of your choice; we will assume the following prefix:

- *$FUNAMBOL_HOME*: the directory where the bundle has been installed (e.g.: */opt/Funambol*)
- *$FUNAMBOL_SDK_HOME*: the directory where the Funambol SDK has been installed (e.g.: */opt/Funambol/tools/sdk*
- *$JAVA_HOME*: the directory where Java is installed (e.g.: */opt/jdk1.5.0_10*)
- *$MAVEN_HOME*: the directory where Apache Maven is installed (e.g.: */opt/apache-maven-2.0.8*)
- *$USER_HOME*: the home directory of the operating system user you are on (e.g.: */home/ste*, *c:\Users\ste*).

Note: Basic knowledge of Apache Maven, its terminology and principles are assumed, as the following sections use terms from the Apache Maven world without explaining them in details.

After installing Maven, you need to configure it so that it points to the Funambol public Maven repository (*m2.funambol.org/repositories*). To do so, copy the file *$FUNAMBOL_SDK_HOME/docs/settings.xml* under *$USER_HOME/.m2.*

6.3. Overview

We will develop the sample module following these steps:

1. Create the connector project
2. Install the module
3. Create a SyncSource instance
4. Test the module with a SyncML client

6.4. Create the connector project

The easiest way to create a connector project is by running the following Maven command:

```
mvn archetype:generate -DarchetypeGroupId=funambol
-DarchetypeArtifactId=funambol-module-archetype -DarchetypeVersion=8.0.0
-DgroupId=acme -DartifactId=acmeconnector
-DarchetypeRepository=http://m2.funambol.org/repositories/artifacts
-Dversion=1.0.0
```

You will be prompted for an answer; type “Y”.

This command will download and create a skeleton application ready to be built which contains:

1. a normal SyncSource
2. a mergeable SyncSource
3. an input/output synclet
4. all configuration and SQL files

All of the above will be generated in a Maven project located in directory named *acmeconnector*. The content of the directory is illustrated in Figure 9.

Note: In order to create a Funambol 8.0.2 project, you need to use the flag *-DarchetypeVersion=8.0.2* on the command line. For a Funambol 6.5 project, use *-DarchetypeVersion=6.5.2.*

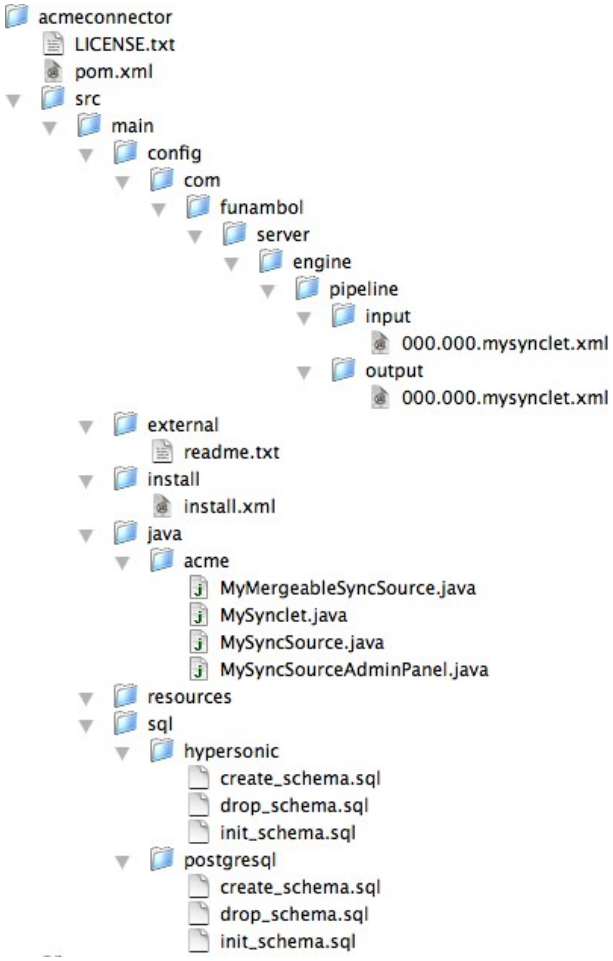


Figure 9: Module source directory structure

The following table explains the function of each file:

File	***Description***
LICENSE.txt	AGPL license file
pom.xml	Maven project file for the connector
src/main/config/com/funambol/server/engine/pipeline/input/000.000.mysynclet.xml	Sample input synclet configuration
src/main/config/com/funambol/server/engine/pipeline/output/000.000.mysynclet.xml	Sample output synclet configuration
src/main/external/readme.txt	Readme for the content of this directory
src/main/install/install.xml	Module installation file
src/main/java/acme/MyMergeableSyncSource.java	Sample mergeable SyncSource
src/main/java/acme/MySynclet.java	Sample synclet
src/main/java/acme/MySyncSource.java	Sample SyncSource
src/main/java/acme/MySyncSourceAdminPanel.java	Sample administration panel for both MySyncSource and MyMergeableSyncSource
src/main/sql/*/create_schema.sql	SQL scrip to create the database tables required by the module
src/main/sql/*/drop_schema.sql	SQL scrip to drop the database tables required by the module
src/main/sql/*/init_schema.sql	SQL scrip to initialize the database tables required by the module

Take a moment to explore and open each file.

Note: the skeleton project creates both a normal SyncSource and a mergeable SyncSource; in this section we will only consider the latter.

6.4.1. MyMergeableSyncSource type

The SyncSource type is the primary component of the connector. The source code created by the artifact is very simple and it only writes some logging info so that we can trace its execution. However in a real case this is where the code necessary to integrate an external data source will go.

MyMergeableSyncSource inherits most of its behavior from *MySyncSource*; open it into an editor or in your IDE and go through it. *MySyncSource* defines three properties that we are going to be able to set through the Funambol Administration Tool. These are: *myString*, *myInt* and *myMap*. Getter and setter methods are provided. Note also that the class implements all methods of the SyncSource interface writing a log entry. Each method has also a description of what it does and what the developer should add.

6.4.2. MySynclet

In addition to the SyncSource types described earlier, the archetype project contains also a sample input and output synclet: *MySynclet*. Open it into an editor or in your IDE and go through it.

Note: If you don't need to write a synclet, you can skip this section.

The first thing to note is that it implements both an input and an output synclet, which means that the synclet will be called for both incoming and outgoing messages. The synclet is very simple and once more it just logs the message in the *funambol.myconnector* logger.

6.4.3. MySyncSourceAdminPanel

We want to be able to create a new *MySyncSource* and configure it from the Funambol Administration Tool. This is possible thanks to the class *MySyncSourceAdminPanel*. Open it into an editor or in your IDE and go through it.

MySyncSourceAdminPanel inherits from *SourceManagementPanel*, which is a class of the Admin framework. *SourceManagementPanel* is a JPanel, therefore it has all methods of a swing panel. The *init()* method creates all widgets that we want to display in the Funambol Administration Tool and adds them to the panel. These widgets are:

- source name
- data types supported (e.g. text/vcard)
- data type versions supported (e.g. 2.1)
- source URI
- myString
- myInt
- myMap entry

In addition, it adds to the panel a JButton to add a newly created SyncSource or to save the values of an existing SyncSource. An important aspect to note is that here is where the panel interacts with the Funambol Administration Tool to persist the changes to the server. The code that performs this task is the following:

```
confirmButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent event ) {
    try {
      validateValues();
      getValues();
      if (getState() == STATE_INSERT) {
        SyncSourceAdminPanel.this.actionPerformed(
          newActionEvent(MySyncSourceAdminPanel.this,
                         ACTION_EVENT_INSERT,
```



```
                        event.getActionCommand()));
      } else {
        MySyncSourceAdminPanel.this.actionPerformed(
          new ActionEvent(MySyncSourceAdminPanel.this,
                          CTION_EVENT_UPDATE,
                          event.getActionCommand()));
      }
    } catch (Exception e) {
      notifyError(new AdminException(e.getMessage()));
    }
  }
});
```

The key is that, when needed, the method *actionPerformed()* of the base class is called with a proper event.

The other important method is *updateForm()* where the value of the SyncSource instance are displayed in the proper fields. Again this method is called by the Funambol Administration Tool when an existing instance must be displayed.

6.5. Creating and installing the connector package

To insert the created project into a Funambol module, just go into *acmeconnector* and type:

```
mvn package
```

A typical output will be as follows:

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building acme acmeconnector Module
[INFO]    task-segment: [package]
[INFO] ------------------------------------------------------------------------
[INFO] artifact org.apache.maven.plugins:maven-resources-plugin: checking for updates
from artifacts
[INFO] artifact org.apache.maven.plugins:maven-resources-plugin: checking for updates
from snapshots
[INFO] artifact org.apache.maven.plugins:maven-compiler-plugin: checking for updates
from artifacts
[INFO] artifact org.apache.maven.plugins:maven-compiler-plugin: checking for updates
from snapshots
[INFO] artifact org.apache.maven.plugins:maven-surefire-plugin: checking for updates
from artifacts
[INFO] artifact org.apache.maven.plugins:maven-surefire-plugin: checking for updates
from snapshots
[INFO] artifact org.apache.maven.plugins:maven-jar-plugin: checking for updates from
artifacts
[INFO] artifact org.apache.maven.plugins:maven-jar-plugin: checking for updates from
snapshots
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Compiling 4 source files to /Users/ste/Projects/acmeconnector/target/classes
[INFO] [resources:testResources]
```



```
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
[INFO] No sources to compile
[INFO] [surefire:test]
[INFO] No tests to run.
[INFO] [jar:jar]
[INFO]   Building   jar:   /Users/ste/Projects/acmeconnector/target/acmeconnector-1.0-
SNAPSHOT.jar
[INFO] [funambol:s4j]
[INFO] Exploding Funambol packaging...
[INFO]       Assembling       Funambol       packaging       acmeconnector       in
/Users/ste/Projects/acmeconnector/target/acmeconnector-1.0-SNAPSHOT
[INFO] Including license file /Users/ste/Projects/acmeconnector/LICENSE.txt
[INFO]
[INFO]     Including artifacts:
[INFO]     -------------------
[INFO]         x funambol:server-framework:jar:8.0.3-SNAPSHOT:compile
[INFO]         x funambol:core-framework:jar:6.5.4:compile
[INFO]         x funambol:ext:jar:6.5.2:compile
[INFO]         o org.jibx:jibx-run:jar:1.1.2fun:compile
[INFO]         o xpp3:xpp3:jar:1.1.2a-fun:compile
[INFO]         o commons-lang:commons-lang:jar:2.3:compile
[INFO]         o funambol:admin-framework:jar:6.5.2:compile
[INFO]
[INFO]     Excluded artifacts:
[INFO]     -------------------
[INFO]
[INFO]     Including jar files...
[INFO] basedir: /Users/ste/Projects/acmeconnector
[INFO] srcDir: /Users/ste/Projects/acmeconnector/src/main
[INFO]     sqlDirectory:     /Users/ste/Projects/acmeconnector/target/acmeconnector-1.0-
SNAPSHOT/sql
[INFO]    wsddDirectory:    /Users/ste/Projects/acmeconnector/target/acmeconnector-1.0-
SNAPSHOT/wsdd
[INFO]     No config files...
[INFO]     No exclude files...
[INFO]     Including install files...
[INFO]     Including sql files...
[INFO]     No wsdd files...
[INFO]
[INFO]                  Generating                  Funambol                  packaging
/Users/ste/Projects/acmeconnector/target/acmeconnector-1.0-SNAPSHOT.s4j
[INFO]   Building   jar:   /Users/ste/Projects/acmeconnector/target/acmeconnector-1.0-
SNAPSHOT.s4j
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 12 seconds
[INFO] Finished at: Sat May 17 15:42:08 CEST 2008
```



```
[INFO] Final Memory: 9M/16M
[INFO] ------------------------------------------------------------------------
```

Note: The final message "Generating Funambol packaging */Users/ste/Projects/acmeconnector/target/acmeconnector-1.0-SNAPSHOT.s4j*" tells where the package is created.

In order to install it into the server, copy *acmeconnector-1.0-SNAPSHOT.s4j* into *$FUNAMBOL_HOME/ds-server/modules* and follow the steps detailed below:

1. Make sure Funambol is up and running.
2. Using a text editor, open the *$FUNAMBOL_HOME/ds-server/install.properties* file.
3. Find the line that begins *modules-to-install*= in the Module definitions section. This line specifies, in a comma-separated list, the modules to install during installation.
4. Add *acmeconnector-1.0-SNAPSHOT* to the comma-separated list (note that you do not have to specify the .s4j filename extension).
5. Save and close i*nstall.properties*.
6. Open a command window and run the server installation script by typing the following:
   - on Windows:

```
cd $FUNAMBOL_HOME/ds-server
bin\install-modules
```

   - on Unix/Linux::

```
cd $FUNAMBOL_HOME/ds-server
bin/install-modules
```

7. Answer all questions about re-creating the modules DB schema when prompted; type 'y' when asked to install the database for the new module and 'n' for all others.

```
[echo] Funambol Data Synchronization Server will be installed on the Tomcat 5.5.x application server
[echo] Undeploying funambol...
[echo] Pre installation for modules foundation-8.0.1,acmeconnector-1.0-SNAPSHOT
[echo] foundation-8.0.1 pre-installation...
[echo] foundation-8.0.1 pre-installation successfully completed
[echo] acmeconnector-1.0-SNAPSHOT pre-installation...
[echo] acmeconnector-1.0-SNAPSHOT pre-installation successfully completed
[echo] Copying configuration files
[echo] Post installation for modules foundation-8.0.1,acmeconnector-1.0-SNAPSHOT
[echo] has.install: true
[echo] Starting custom installation...
[echo] Foundation Installation
[echo] Foundation installation successfully completed
[echo] foundation-8.0.1 installation...
[echo] Database installation for module foundation-8.0.1 on hypersonic (/opt/Funambol/ds-server)

[iterate] The Funambol Data Synchronization Server installation program can now create
[iterate] the database required by the module foundation-8.0.1 (if any is needed).
[iterate] You can skip this step if you have already a valid database created
```



```
[iterate] or the module does not require a database.

[iterate] If you choose 'y' your existing data will be deleted.

[iterate] Do you want to recreate the database?

[iterate]       (y,n)

n
```

and

```
[echo] foundation-8.0.1 installation successfully completed

[echo] has.install: true

[echo] Starting custom installation...

[echo] acmeconnector installation

[echo] acmeconnector installation successfully completed

[echo] acmeconnector-1.0-SNAPSHOT installation...

[echo] Database installation for module acmeconnector-1.0-SNAPSHOT on hypersonic
(/opt/Funambol/ds-server)

[iterate] The Funambol Data Synchronization Server installation program can now create

[iterate] the database required by the module acmeconnector-1.0-SNAPSHOT (if any is
needed).

[iterate] You can skip this step if you have already a valid database created

[iterate] or the module does not require a database.

[iterate] If you choose 'y' your existing data will be deleted.

[iterate] Do you want to recreate the database?

[iterate]       (y,n)
y
[echo] acmeconnector-1.0-SNAPSHOT installation successfully completed

[war] Warning: selected war files include a WEB-INF/web.xml which will be ignored
(please use webxml attribute to war task)

[echo] Remove output dir


BUILD SUCCESSFUL

Total time: 12 seconds
```

8. Restart the Data Synchronization Service.

The new connector is now installed.

Note: installed modules are visible in the Funambol Administration Tool, in the *Modules* section.

6.6. Creating a SyncSource

Now that the connector is installed, you can see it in the Funambol Administration Tool. To access it, start the Funambol Administration Tool by selecting *Start* | *All Programs* | *Funambol* | *Administration Tool*. The Funambol Administration Tool window appears (see Figure 10).

First of all, you will need to log into the server: on the Main Menu bar, select File | Login. The Login window displays. Verify that the fields are populated as follows, or specify the following values:

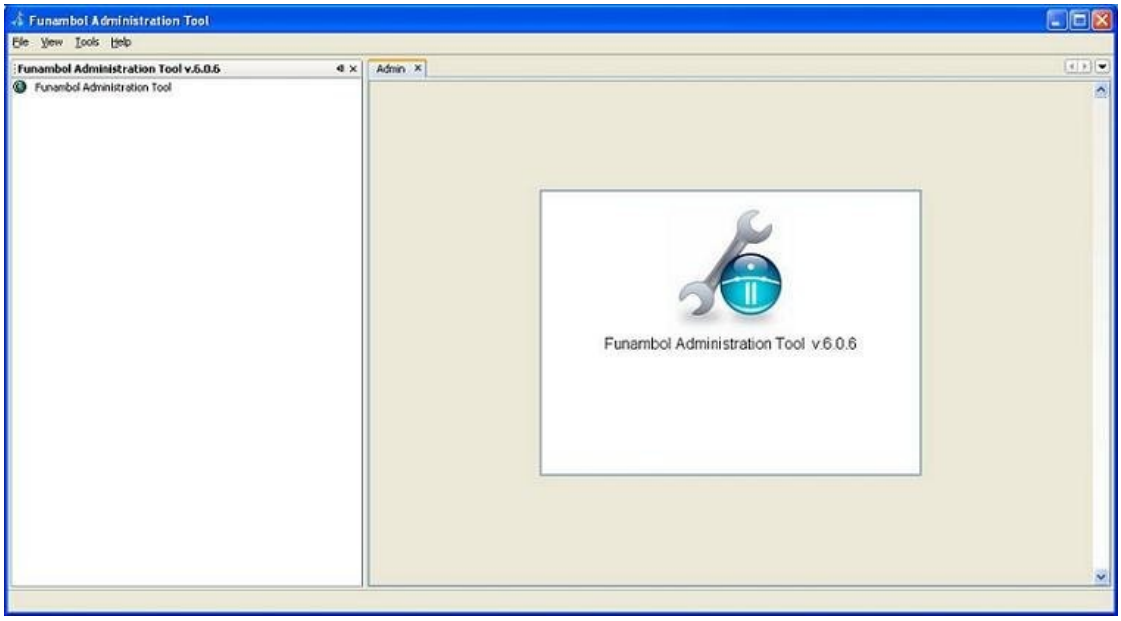


Figure 10: Funambol Administration Tool

```
Hostname/IP: <localhost> (should be your machine name)
Port: 8080
User name: admin
Password: sa
```

Click Login; the Output Window in the lower right pane should display "connected." In the left panel, expand the localhost tree as follows: localhost | Modules | acme | acmeconnector, then select MyMergeableSyncSource. The Edit My SyncSource screen appears in the upper right pane, as shown in Figure 11.

This window is used to specify configuration values. Insert the following values and press Add:

```
Source URI: acme
Name: Acme
Supported type: text/plain
Supported version: 1.0
MyString: acme connector!
MyInt: 10
MyMap entry: <acme, connector>
```

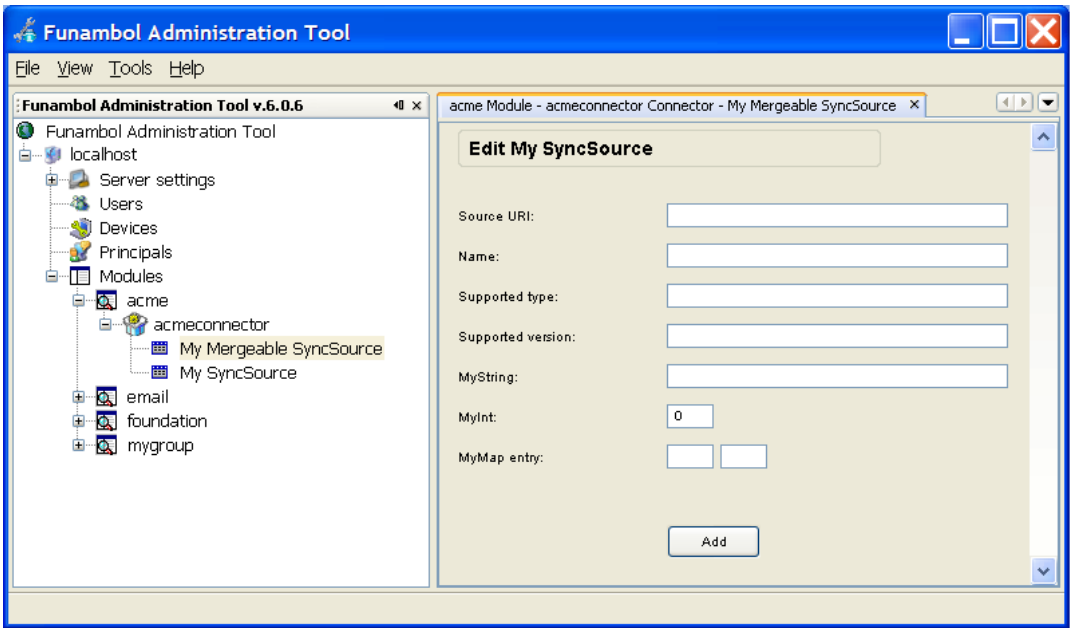



Figure 11: Edit File System SyncSource screen

6.7. Testing the connector

To test the Acme connector we will use a simple command line SyncML client which is distributed in the Funambol SDK under *$FUNAMBOL_SDK_HOME/plug-ins/cl*. Perform the following:

1. Edit the file *config/spds/sources/briefcase.properties* and set the following values:

```
name=acme
sourceClass=com.funambol.syncclient.spds.source.FileSystemSyncSource
sourceDirectory=db/briefcase
type=text/plain
sync=two-way
encode=true
sourceURI=acme
```

2. Run the command *$FUNAMBOL_SDK_HOME\plug-ins\cl\binrun.cmd* (or *$FUNAMBOL_SDK_HOME/plug-ins/cl/run.sh* if using Linux)

You will see an output similar to the following:

```
Funambol Command Line Tool ver. 8.0.1
--------------------------
2008-04-17 16:28:10:969 - # SyncClient API J2SE Log
16:28:10:970 [INFO] - Initializing
16:28:10:973 [INFO] - Sending initialization commands
16:28:11:716 [INFO] - The server alert code for acme is 201
6:28:11:718 [INFO] - Synchronizing acme
16:28:11:745 [INFO] - exchange modifications started
16:28:11:746 [INFO] - Preparing slow sync for acme
16:28:11:747 [INFO] - Detected 0 items
16:28:11:748 [INFO] - Sending modifications
16:28:11:837 [INFO] - Returned 0 new items, 0 updated items, 0 deleted items for acme
16:28:11:838 [INFO] - Mapping started
```



```
16:28:11:841 [INFO] - Sending mapping

16:28:11:853 [INFO] - Sending mapping

16:28:11:874 [INFO] - Mapping done

16:28:11:874 [INFO] - Synchronization done
```

In the server log you will be able to see your connector at work. Filtering out the lines that are not of interest for our connector, the log entries will be similar to the following text:

```
[2008-05-17 16:31:56,656] [funambol.myconnector] [INFO]
[399F31F06404433DE69A05F71D562ACD] [sc-api-j2se] [guest] [] Initializing
acme.MyMergeableSyncSource

[2008-05-17 16:31:56,657] [funambol.myconnector] [INFO]
[399F31F06404433DE69A05F71D562ACD] [sc-api-j2se] [guest] [] myString: acme connector!

[2008-05-17 16:31:56,657] [funambol.myconnector] [INFO]
[399F31F06404433DE69A05F71D562ACD] [sc-api-j2se] [guest] [] myInt: 10

[2008-05-17 16:31:56,657] [funambol.myconnector] [INFO]
[399F31F06404433DE69A05F71D562ACD] [sc-api-j2se] [guest] [] myMap: {acme=connector}

[2008-05-17 16:31:56,703] [funambol.myconnector] [INFO]
[399F31F06404433DE69A05F71D562ACD] [sc-api-j2se] [guest] [acme] Starting
synchronization: com.funambol.framework.engine.source.SyncContext@e85079

[2008-05-17 16:31:56,703] [funambol.myconnector] [INFO]
[399F31F06404433DE69A05F71D562ACD] [sc-api-j2se] [guest] [acme] getNewSyncItemKeys()

[2008-05-17 16:31:56,703] [funambol.myconnector] [INFO]
[399F31F06404433DE69A05F71D562ACD] [sc-api-j2se] [guest] [acme]
getUpdatedSyncItemKeys()

[2008-05-17 16:31:56,703] [funambol.myconnector] [INFO]
[399F31F06404433DE69A05F71D562ACD] [sc-api-j2se] [guest] [acme]
getDeletedSyncItemKeys()

[2008-05-17 16:31:56,703] [funambol.myconnector] [INFO]
[399F31F06404433DE69A05F71D562ACD] [sc-api-j2se] [guest] [acme] Committing
synchronization

[2008-05-17 16:31:56,782] [funambol.myconnector] [INFO]
[399F31F06404433DE69A05F71D562ACD] [sc-api-j2se] [guest] [acme] Ending synchronization
```

Note: in the first entries the values inserted earlier in the administration panel are displayed, meaning that the SyncSources were properly configured.

Similarly, we can see the effect of *MySynclet*. Each SyncML message has been logged; for example the first input message is something like the following text:

```
[2008-05-17 16:59:28,576] [funambol.myconnector] [INFO]
[FF4B335BA02F0581DAFF016319EDD263] [] [] []
--------------------------------------------------------------------------------[2008-
05-17 16:59:28,576] [funambol.myconnector] [INFO] [FF4B335BA02F0581DAFF016319EDD263]
[] [] [] Input message[2008-05-17 16:59:28,576] [funambol.myconnector] [INFO]
[FF4B335BA02F0581DAFF016319EDD263] [] [] []
[2008-05-17 16:59:28,585] [funambol.myconnector] [INFO]
[FF4B335BA02F0581DAFF016319EDD263] [] [] [] <?xml version="1.0" encoding="UTF-8"?
><SyncML><SyncHdr><VerDTD>1.1</VerDTD><VerProto>SyncML/1.1</VerProto><SessionID>123456
78</SessionID><MsgID>1</MsgID><Target><LocURI>http://localhost:8080/funambol/ds</LocUR
I></Target><Source><LocURI>sc-api-j2se</LocURI></Source><Cred><Meta><Type>syncml:auth-
basic</Type></Meta><Data>Z3Vlc3Q6Z3Vlc3Q=</Data></Cred><Meta><MaxMsgSize>250000</MaxMs
gSize><MaxObjSize>4000000</MaxObjSize></Meta></SyncHdr><SyncBody><Alert><CmdID>1</CmdI
D><Data>200</Data><Item><Target><LocURI>acme</LocURI></Target><Source><LocURI>acme</Lo
cURI></Source><Meta><Anchor><Last>1211034716596</Last><Next>1211036368401</Next></Anch
or></Meta></Item></Alert><Final/></SyncBody></SyncML>[2008-05-17 16:59:28,585]
[funambol.myconnector] [INFO] [FF4B335BA02F0581DAFF016319EDD263] [] [] []
--------------------------------------------------------------------------------
```

And the last output message is something like the following:

```
[2008-05-17 16:59:28,876] [funambol.myconnector] [INFO]
[FF4B335BA02F0581DAFF016319EDD263] [sc-api-j2se] [guest] []
--------------------------------------------------------------------------------[2008-
05-17 16:59:28,877] [funambol.myconnector] [INFO] [FF4B335BA02F0581DAFF016319EDD263]
```



```
[sc-api-j2se] [guest] [] Output message[2008-05-17 16:59:28,877]
[funambol.myconnector] [INFO] [FF4B335BA02F0581DAFF016319EDD263] [sc-api-j2se] [guest]
[]                                                    [2008-05-17 16:59:28,877]
[funambol.myconnector] [INFO] [FF4B335BA02F0581DAFF016319EDD263] [sc-api-j2se] [guest]
[] <?xml version="1.0" encoding="UTF-8"?
><SyncML><SyncHdr><VerDTD>1.1</VerDTD><VerProto>SyncML/1.1</VerProto><SessionID>123456
78</SessionID><MsgID>5</MsgID><Target><LocURI>sc-api-
j2se</LocURI></Target><Source><LocURI>http://localhost:8080/funambol/ds</LocURI></Sour
ce><RespURI>http://localhost:8080/funambol/ds;jsessionid=FF4B335BA02F0581DAFF016319EDD
263</RespURI></SyncHdr><SyncBody><Status><CmdID>1</CmdID><MsgRef>5</MsgRef><CmdRef>0</
CmdRef><Cmd>SyncHdr</Cmd><TargetRef>http://localhost:8080/funambol/ds</TargetRef><Sour
ceRef>sc-api-
j2se</SourceRef><Data>200</Data></Status><Final/></SyncBody></SyncML>[2008-05-17
16:59:28,877] [funambol.myconnector] [INFO] [FF4B335BA02F0581DAFF016319EDD263] [sc-
api-j2se] [guest] []
--------------------------------------------------------------------------------
```

If you have reached this point, you have probably successfully developed a Funambol connector: congratulations! You can now go in more details in the *acmeconnector* project or start your own.

6.8. Debugging

In order to debug the connector, follow these steps:

1. Start Tomcat using the debug options for the JVM. If you are using the Funambol package, simply edit the file *$FUNAMBOL_HOME/bin/funambol-server* uncommenting the *JPDA_OPTS* line, which you can easily find if you search for “debug”. This will start Tomcat enabling it for remote debugging.
2. Connect to the running JVM. If you use Eclipse, you can create a debug profile to attach to Tomcat; note that the debug port given in the *JPDA_OPTS* line is 8787, not eclipse's default port, i.e. 8000, so either change the *JDPA_OPTS* or the Eclipse port.
3. You can now start a sync session and stop on breakpoints.

7. Developing a SyncSource

A SyncSource is the means a set of data is made available to the synchronization engine. Therefore, in order to synchronize any type of data (files, database tables, calendar events and so on), there must be a proper SyncSource able to extract and store the data from and to the real data store.

Goal of the Funambol platform is to provide a collection of SyncSources for the most common uses (i.e. files), but new SyncSources can be independently developed and plugged in the synchronization engine so that the server will be able to process synchronization requests targeted to virtually any data source.

7.1. The SyncSource interface and related classes

The core of the SyncSource architecture is the interface *sync4j.framework.engine.source.SyncSource*. This interface does not make any assumption on the type of data being synchronized, so that its concrete implementations are completely free to access their own underlying storage.

A SyncSource is identified by a *sourceURI* and usually a *name;* the former is the URI that a SyncML client must specify as target in order to synchronize this particular SyncSource; the latter is a human-friendly name used for displaying purposes only. Note that they must be both unique within a Funambol installation.

A SyncSource is also associated to a type, in the form of a MIME type that represents the type of data handled by the source.

The most important methods defined by the SyncSource interface are: (see chapter 11 for details)

Method	***Description***
beginSync()	This is the first SyncSource method the sync engine calls and it is used to specify who is going to synchronize and which type of synchronization is requested.
endSync()	This is the latest SyncSource method the sync engine calls and it may be used to perform finalization tasks.
getUpdatedSyncItems	Called to retrieve the updated SyncItems for the given principal since the given point in time. For a definition of what a Principal is, please see Section 7.1.1 below.
getUpdatedSyncItemKeys	Called to retrieve the SyncItemKey of the updated items for the given principal since the given point in time.
getNewSyncItems	Called to retrieve the new SyncItems for the given principal since the given point in time.
getNewSyncItemKeys	Called to retrieve the SyncItemKey of the new items for the given principal since the given point in time.
getDeletedSyncItems	Called to retrieve the deleted SyncItems for the given principal since the given point in time.
getDeletedSyncItemKeys	Called to retrieve the SyncItemKey of the deleted items for the given principal since the given point in time.
getAllSyncItems	Called to retrieve all the SyncItems for the given principal since the given point in time.
setSyncItem	Called to insert or update the given item.
setSyncItems	Called to insert or update the given items.
removeSyncItem/s	Called to remove the given item(s).
getSyncItemFromTwin	Called to find items that represent the same information as the given item. It is used in conflict detection and during slow sync to associate a client item with a server item which is similar enough.

Table 5 - SyncSource: most important methods

Funambol provides others two subtypes of the *SyncSource* interface: *MergeableSyncSource* and *FilterableSyncSource*.

com.funambol.framework.engine.source.MergeableSyncSource: this interface should be used when you want that conflicting data are merged when possible. The merge is performed when a conflict is

detected in order to avoid loss of information. For example, let's assume that the same contact has been defined on both the client adding a mobile phone, and the server, adding an email address. At the next sync a conflict will be detected. If the contact is generated by a *MergeableSyncSource*, the sync engine asks the SyncSource to merge the conflicting items. In the example, the result of the merge will be a contact with both the new mobile number and email address. This contact will be stored on the server and sent back to the client.

Method	***Description***
mergeSyncItem(serverKey: SyncItemKey, clientItem: SyncItem): boolean	Called when a conflict must be resolved merging the items. Server side, the merge result must be persistent in the underlying datastore. If the item on the client must be updated, this method must return true and put the new content in the given clientItem.

com.funambol.framework.engine.source.FilterableSyncSource: this interface should be used when the SyncSource supports SyncML 1.2 filtering.

Method	***Description***
getSyncItemStateFromId(itemKey: SyncItemKey): char	Called to retrieve the state of an item. This method is required because the server cannot know directly the state of an item not inside the filter criteria.
isSyncitemInFilterClause(item: SyncItem): boolean	Called to know if an item satisfies the filter.
isSyncitemInFilterClause(itemKey: SyncItemKey): boolean	Called to know if the item specified with the given key satisfies the filter.

7.1.1. SyncContext

The *begin()* method of SyncSource is called when a new sync is started. It requires just a *SyncContext* as input parameter. A *SyncContext* contains the following information:

- *principal* (of type *com.funambol.framework.security.Principal*)
- *syncMode* (of type int)
- *filter* (of type *com.funambol.framework.filter.Filter*)
- *since* (of type java.sql.Timestamp)
- *to* (of type java.sql.Timestamp)

A *principal* is composed of a *Sync4jUser* and a *Sync4jDevice* because the same user may use different devices (or the same device could be used by different users).

syncMode represents the synchronization type performed. Can be one of:

syncMode	***synchronization type***
200	TWO-WAY
201	SLOW
202	ONE_WAY_FROM_CLIENT
203	REFRESH_FROM_CLIENT
204	ONE_WAY_FROM_SERVER
205	REFRESH_FROM_SERVER

filter represents the filter required by the client. If a filter is specified and the SyncSource is a *FilterableSyncSource*, the expected behavior of the SyncSource is:

- the methods *getAllSyncItemKey()*, *getNewSyncItemKeys()*, *getDeletedSyncItemKeys()* and *getUpdatedSyncItemKeys()* return only the items inside the filter

- the methods *getSyncItemKeysFromTwin*, *getSyncItemStateFromId* ignore the filter.

*Since i*s the last successfully completed synchronization session start time.

to represents the current synchronization start time.

7.1.2. SyncItem

Items returned by a SyncSource are encapsulated in f*unambol.framework.engine.SyncItem* objects. *SyncItem* defines the following methods:

Method	*Description*
getKey	Returns the item key.
getParentKey	Returns the item's parent key (hierarchic sync)
getState	Returns the item state.
setState	Sets the item state.
getContent	Returns the item content
setContent	Sets the item content
getFormat	Returns the item format
setFormat	Sets the item format
getType	Returns the item type
setType	Sets the item type
getTimestamp	Returns the timestamp of the last change of the item state and it is used in the synchronization process, in order to determine the operation to be performed on the sources.
setTimestamp	Sets the item timestamp
getSyncSource	Returns the SyncSource the item belongs to.

Table 6 - SyncItem methods

Note: when a SyncSource creates a *SyncItem*, it must always provide a value for the content and for the timestamp.

7.1.3. Twin items

An item X is a twin of an item Y when from the SyncSource point of view, X and Y represent the same information.

For example, two event items, both at the same time and in the same place may be considered the same appointment, even if the associated note is different. Or two contacts may be considered the same person if they have the same first, middle and last name.

Because the SyncSource is the only entity with knowledge about how data are stored and represented in the data source, the SyncSource is the only component that can select the twins of a given item. The synchronization engine does it calling *getSyncItemsFromTwin()*.

Twin items are necessary in two circumstances:

- during full sync
- during conflict detection

During full sync, the client sends all its items and the server has to discover which operation the client should apply in order to make its data set look identical to the one stored on the server. Because the two devices are supposed to be out of sync, the server cannot relay on LUID-GUID mappings. In this case, for each item given by the client, the server must search for twin items in the backend data source. If a twin is not found, the new item is added. If a twin is found, the server will consider to have such client item and won't send back a command for it. If the SyncSource being synchronized is mergeable SyncSource, the synchronization engine will ask the SyncSource instance to merge the client and server items and will send back the merged item to the client.

The same process is valid during conflict detection. When during fast sync a client sends a new item, the server should first check if this item is conflicting with something else. Therefore, the server calls the SyncSource's *getSyncItemsFromTwin()*. If this method returns something, a conflict is detected and handled accordingly. If no twin is found the item can be added.

The SyncSource developer should apply the more appropriated comparison logic according to the type of data the SyncSource menages. Optionally, the SyncSource can disable this type of conflict with no impact on the calling module, by implementing this method as a stub and always returning a failure to find a twin item.

One final note about twin computation is that it should be kept as simple as possible, since it involves database searches that may have a big impact on performance.

7.1.4. The Funambol Administration Tool configuration panel

The Funambol Administration Tool makes it possible to perform many administration tasks, including the configuration of the Funambol Data Synchronization Service, loggers and SyncSources (for more information, please see [3]).

After logging into the Funambol Administration Tool and connecting to the server, users will see something similar to the window in Figure 12. Selecting an existing SyncSource, a configuration panel is displayed on the right side of the window.

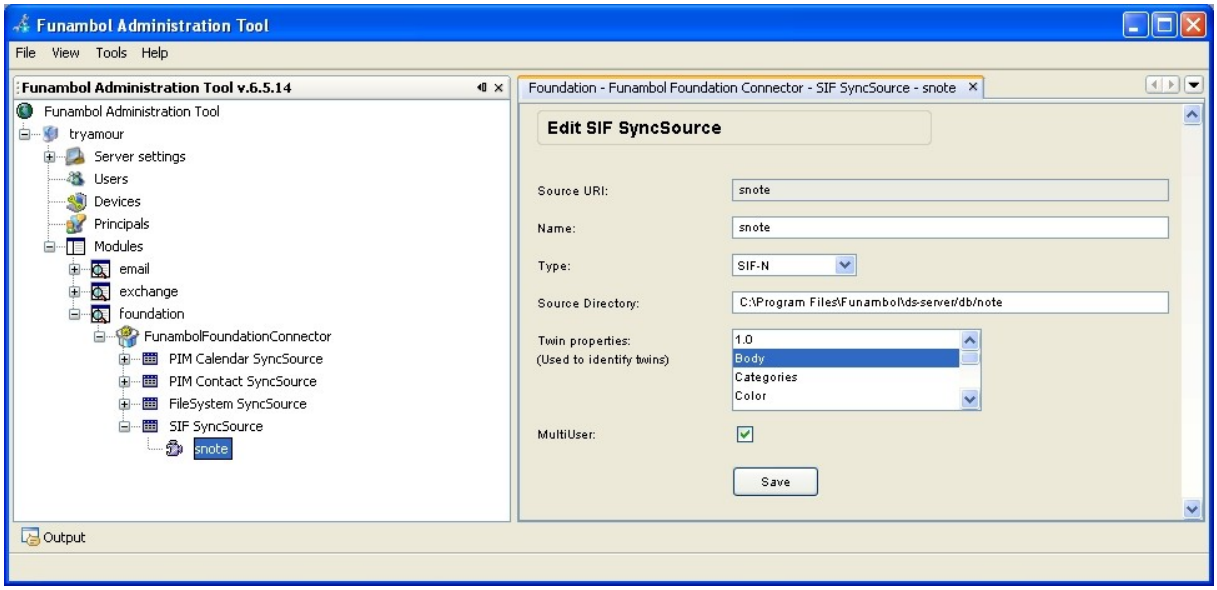


Figure 12: SyncAdmin showing connectors, modules and SyncSources

The Funambol Administration Tool is designed to be extended by developers so that a new custom SyncSource type can be configured into the tool through a custom management panel. The Funambol Administration Tool extension mechanism is described in the following chapter.

8. Extending the Funambol Administration Tool

The Funambol Administration Tool can be extended so that any kind of object can be configured through the UI. The extension mechanism is based on the concept that a module developer should be able to provide custom panels that an administrator can access in order to configure a specific server bean. Currently, a developer can provide management panels for the following objects:

- SyncSource types
- Connectors

Note: the classes implementing such management panels will be installed on the server as part of the module installation.

8.1. Architecture overview

The Funambol Administration Tool extension mechanism is based on the concept of *management object*. A management object is potentially anything that can be configured; however, in this guide the focus is on providing a way for a module developer to provide custom configuration panels for connectors and SyncSource types.

A management object is represented by the *com.funambol.syncadmin.mo.ManagementObject* class, which wraps a generic Object instance together with a management path (i.e. the path of the server bean instance in the configuration path). “Configuring a management object” means mainly setting its properties based on the input from a user. To allow this, the abstract class *com.funambol.syncadmin.ui.ManagementObjectPanel* abstracts a JPanel whose role is to configure a given management object.

Two specialized subclasses of *ManagementObjectPanel* are available to developers to configure SyncSource types and connectors. These are *SourceManegementPanel* and *ConnectorManagementPanel*. These two classes are still meant not to be directly instantiated and are therefore abstract.

A module developer can provide a configuration panel to a SyncSource type or a connector just extending *SourceManagementPanel* or *ConnectorManagementPanel* with a class showing a suitable data entry form for the object. When the module is loaded by the admin, those panel classes are downloaded from the server and instantiated into the Funambol Administration Tool when the user selects a SyncSource type or a connector.

The user can perform the following high-level actions on a custom panel:

- Adding a new object
- Updating an object
- Deleting an object

Such actions are notified to the interested classes in the same way swing events are commonly notified. Figure 13 illustrates the sequence diagram of the calls between different components of the Funambol Administration Tool or server when a user performs different operations on the connector / SyncSource type nodes.

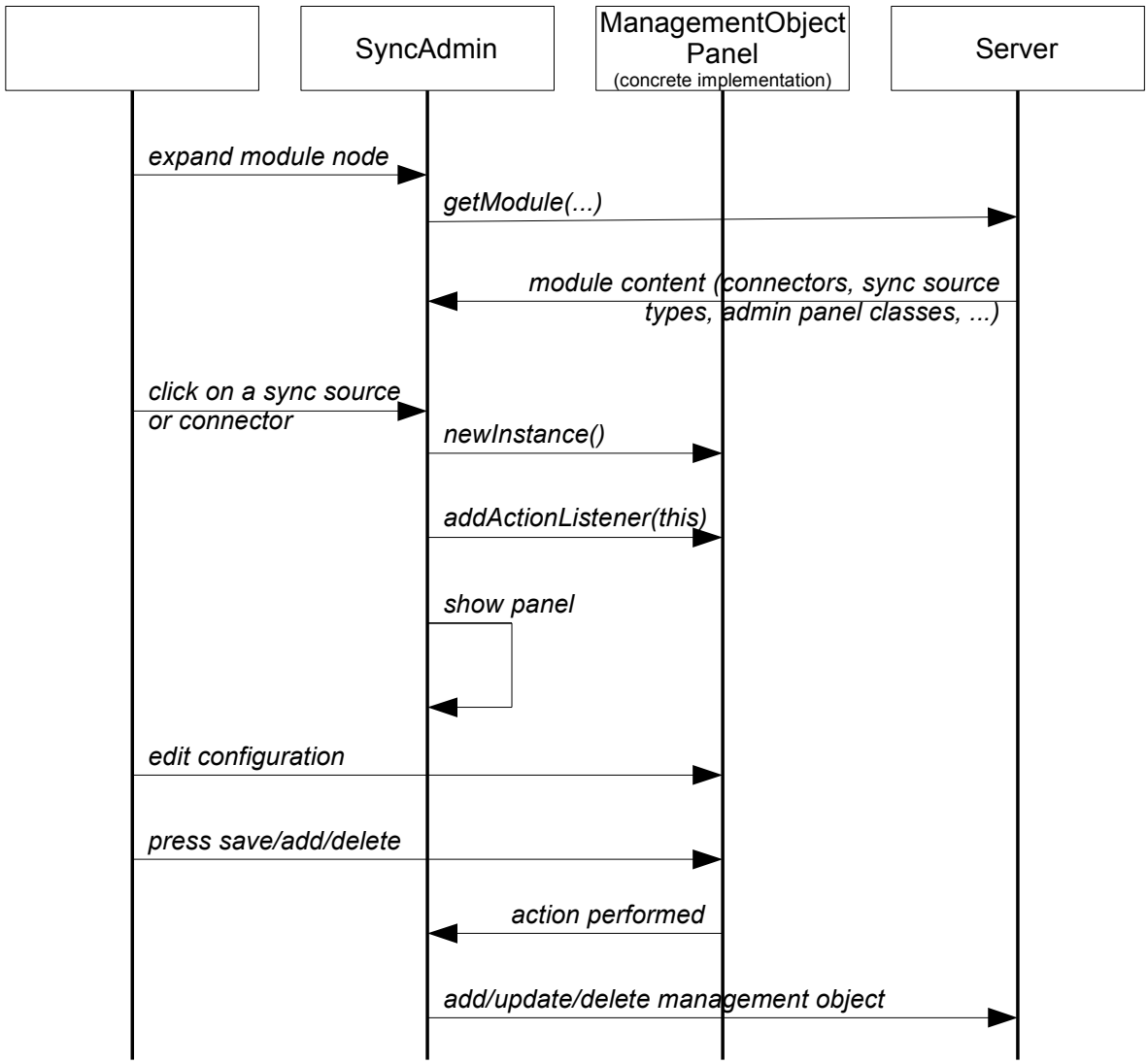


Figure 13: Admin-ManagementObjectPanel collaboration diagram

8.2. ManagementObject and subclasses

The key concept around extending the Funambol Administration Tool with custom configuration panels is the *management object*. A management object can be any object we want to be able to manipulate through of the Funambol Administration Tool. In particular, any server beans can be considered a management object.

A management object is an association between a Java object instance and a management path; the management path represents the name of such instance in the context of a tree-like configuration repository.

In the Admin framework a management object is represented by the class *com.funambol.admin.mo.ManagementObject*. This has the interface of the table below.

Method	***Description***
Constructors	
ManagementObject (Object object, String path)	Creates a new management object from the instance and the instance pathname
Public methods	
Object getObject()	Returns the object instance

Method	Description
void setObject(Object o)	Sets the object instance
String getPath()	Returns the management object pathname
void setPath(String path)	Sets the management object pathname

Two specializations of *ManagementObject* are provided as described in the following: *SyncSourceManagementObject* and *ConnectorManagementObject*. These are better described in the following sections.

8.2.1. com.funambol.admin.mo.SyncSourceManagementObject

This management object represents a SyncSource instance. It associates the instance to the module / connector / SyncSourceType IDs.

The interface of this class is shown in the table below.

Method	Description
Constructors	
SyncSourceManagementObject(SyncSource source, String moduleId, String connectorId, String sourceTypeId)	Creates a new SyncSource management object from the instance and its module/connector/type IDs
Public methods	
String getModuleId()	Returns the module ID
void setModuleId(String ID)	Sets the module ID
String getConnectorId()	Returns the connector ID
void setConnectorId(String ID)	Sets the connector ID
String getSourceTypeId()	Returns the source type ID
void setSourceTypeId(String ID)	Sets the source type ID
String getTransformationsRequired	Returns the engine transformations required (for instance 'b64')
setTransformationsRequired(String transformations)	Sets the required transformations

8.2.2. com.funambol.admin.mo.ConnectorManagementObject

This management object represents a connector configuration object. It associates the instance to the module / connector IDs. Plus, the management path is created as:

```
moduleId + '/' + connectorId + '/' + connectorName + ".xml"
```

The interface of this class is shown in the table below.

Method	Description
Constructors	
ConnectorManagementObject(Object obj, String moduleId, String connectorId, String connectorName)	Creates a new connector management object from the instance and its module/connector/type IDs. obj represents the configuration object, retrieved from the server as a server bean with the pathname generated as above.
Public methods	
String getModuleId()	Returns the module ID
void setModuleId(String ID)	Sets the module ID
String getConnectorId()	Returns the connector ID

Method	Description
void setConnectorId(String ID)	Sets the connector ID

8.3. ManagementObjectPanel and subclasses

ManagementObjectPanel is an abstract class that provides no user interaction widgets. It must be extended by concrete implementations in order to do something useful.

To notify actions to the framework, *ManagementObjectPanel* is a *java.awt.event.Action subject* (referring to the *Observer* pattern). This means that any class interested in knowing about (observing) what is happening in a management panel shall implement the *java.awt.ActionListener* interface and register itself to the panel, calling *addActionListener()*. In the Admin framework such interested classes are represented by the controllers (*SyncSourcesController*, *ConnectorController*, and so on).

The *ActionEvent* generated and notified by *actionPerformed()* to all listeners shall have the following characteristics:

- the panel instance as source
- one of the following event IDs:
  1. ACTION_EVENT_INSERT
  2. ACTION_EVENT_UPDATE
  3. ACTION_EVENT_DELETE
- an arbitrary value as command string

For example, a custom management panel to configure a connector may have a “save” button, through which the user can persist changes. When the button is pressed, the panel should notify the action to the listeners. This can be done with a code similar to the following:

```
public class ConfigPanel
extends ManagementObjectPanel {
...
public ConfigPanel() {
  ... widgets creation and initialization ...
  saveButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event ) {
      try {
        ... values validation ...
        ... management object update ....
        ConfigPanel.this.actionPerformed(
                              new   ActionEvent(ConfigPanel.this,   ACTION_EVENT_UPDATE,
event.getActionCommand())
        );
      } catch (Exception e) {
        notifyError(new AdminException(e.getMessage()));
      }
    }
  });
}
...
}
```

ManagementObjectPanel has the following interface.

Method	Description
Abstract methods	
void updateForm()	Tells the panel that it has to update the UI with the values in the management object.
Public methods	
void setManagementObject(ManagementObject)	Sets the object under editing by this management panel
ManagementObject getManagementObject()	Returns the edited object
void addActionListener(ActionListener)	Registers the given action listener
void removeActionListener(ActionListener)	Unregisters the given action listener
void notifyError(AdminException)	Called when an error related to the panel should be displayed
Protected methods	
void actionPerformed(ActionEvent)	Notifies an action event to all listeners.

Two subclasses of a *ManagementObjectPanel* are provided by the Funambol administration framework. They are described in the following sections.

8.3.1. SourceManagementPanel

This class specializes a *ManagementObjectPanel* to a particular type of management objects: SyncSources. This base class is designed to be extended by concrete implementations to configure new and existing SyncSources. The additions to the base class are:

- it handles management objects of type *com.funambol.framework.engine.source.SyncSource*
- it defines two states for the management panel: *insert* and *update*

The former is achieved providing a *getSyncSource()* methods that returns the SyncSource instance under editing. The latter is used to differentiate the case where the user is creating a new SyncSource (therefore the SyncSource instance under editing does not exist yet on the server) from the case where the user is editing an existing SyncSource. The administration framework will take care of changing the status from INSERT to UPDATE.

SourceManagementPanel has the following interface.

Method	Description
Public methods	
void setState(int)	Sets the panel state.
int getState()	Returns the panel state
SyncSource getSyncSource()	Returns the SyncSource instance under editing

8.3.2. ConnectorManagementPanel

This class represents the base class for the management panel of a connector. It does not adds much the its base class, but it just provides a shortcut to easily gather the server bean instance used to configure the connector.

ConnectorManagementPanel has the following interface.

Method	Description
Public methods	
Object getConfiguration()	This is a shortcut to getManagementObject().getObject().

9. Configuring Funambol components

One of the Funambol design goal is to provide a framework that can be used to implement any kind of synchronization service, extending existing modules or plugging in new modules. All this require a lot of configuration information and possibly an easy way to add configuration for new extensions. Configuration files should be easy to understand, access and change.

Funambol uses mainly two configuration techniques:

- System properties
- Server JavaBeans

In the following sections these two types of configuration are described in details.

9.1. System properties

The only system property directory used by the Funambol Data Synchronization Service is *funambol.home* which must point to the directory where Funambol is installed (commonly referenced as *$FUNAMBOL_HOME*).

This property is specified at JVM invocation time using the *-D* option. On many systems, it is sufficient to set the *JAVA_OPTS* environment variable in order to get it included into the JVM launching command.

9.2. Server JavaBeans

Many Funambol components are configured as *server JavaBeans*. Server JavaBeans are JavaBeans used server-side. The idea is to store a bean configuration as the serialized form of the bean itself. This way, a bean can be instantiated, configured and serialized to persist its configuration. Later, the bean can be deserialized in memory as a properly configured instance.

Funambol makes use of the standard Java facility to serialize objects into XML (and to deserialize them from XML). This is achieved by using the classes *java.beans.XMLEncoder* and *java.beans.XMLDecoder*. Since configuration files created with such encoder/decoder are easy to use, read and write, they can be created and modified manually with a simple text editor, without the need of a dedicated GUI. An additional advantage of this approach is that server JavaBeans are not requested to implement *java.io.Serializable* because *XMLEncoder* does not require it.

This is an example of a server JavaBean:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.1_01" class="java.beans.XMLDecoder">
 <object class="sync4j.framework.server.store.PersistentStoreManager">
  <void property="jndiDataSourceName">
    <string>java:/jdbc/sync4j</string>
  </void>
  <void property="stores">
   <array class="java.lang.String" length="2">
     <void index="0">
      <string>sync4j.server.store.SyncPersistentStore</string>
     </void>
     <void index="1">
      <string>sync4j.server.store.EnginePersistentStore</string>
     </void>
```



```
    </array>
   </void>
  </object>
</java>
```

In order to help server JavaBeans handling, Funambol uses the factory class *com.funambol.framework.tools.beans.BeanFactory,* which in turn makes use of a customized class loader; the class loader handles configuration files in a so called *config path*, in the same way a common class loader handles classes in the classpath.

The XML syntax uses the following conventions:

Each element represents a method call.

- The "object" tag denotes an expression whose value is to be used as the argument to the enclosing element.
- The "void" tag denotes a statement which will be executed, but whose result will not be used as an argument to the enclosing method.
- Elements which contain elements use those elements as arguments, unless they have the tag: "void".
- The name of the method is denoted by the "method" attribute.
- XML's standard "id" and "idref" attributes are used to make references to previous expressions - so as to deal with circularities in the object graph.
- The "class" attribute is used to specify the target of a static method or constructor explicitly; its value being the fully qualified name of the class.
- Elements with the "void" tag are executed using the outer context as the target if no target is defined by a "class" attribute.
- Java's String class is treated specially and is written <string>Hello, world</string> where the characters of the string are converted to bytes using the UTF-8 character encoding.

Although all object graphs may be written using just these three tags, the following definitions are included so that common data structures can be expressed more concisely:

- The default method name is "new".
- A reference to a Java class is written in the form <class>javax.swing.JButton</class>.
- Instances of the wrapper classes for Java's primitive types are written using the name of the primitive type as the tag. For example, an instance of the Integer class could be written: <int>123</int>. Java's reflection is internally used for the conversion between Java's primitive types and their associated "wrapper classes".
- In an element representing a nullary method whose name starts with "get", the "method" attribute is replaced with a "property" attribute whose value is given by removing the "get" prefix and decapitalizing the result.
- In an element representing a monadic method whose name starts with "set", the "method" attribute is replaced with a "property" attribute whose value is given by removing the "set" prefix and decapitalizing the result.
- In an element representing a method named "get" taking one integer argument, the "method" attribute is replaced with an "index" attribute whose value the value of the first argument.
- In an element representing a method named "set" taking two arguments, the first of which is an integer, the "method" attribute is replaced with an "index" attribute whose value the value of the first argument.
- A reference to an array is written using the "array" tag. The "class" and "length" attributes specify the sub-type of the array and its length respectively.

9.2.1. The configuration path

Server JavaBeans are looked for in the configuration path, which is analogous to the class path for classes lookup. This is implemented reading the serialization files from a custom class loader, *com.funambol.framework.config.ConfigClassLoader*. This class loader (which is instead our server beans loader), is configured to read objects from the configuration path. The config path is built appending "*/config*" to the *funambol.home* system property value. For example, if *funambol.home* is set to "*/opt/Funambol/ds-server*", the config path would be "*/opt/Funambol/ds-server/config*".

9.2.2. Lazy initialization

When a server bean is deserialized from its XML form, the classloader that loads the serialization file calls the empty constructor first and then it sets the bean property values using the *setXXX()* methods provided by the class. However, some classes may need additional operations to be performed in order to properly initialize (after *setXXX()* methods are called). To support this *lazy initialization* approach, these classes can implement *com.funambol.framework.tools.beans.LayInitBean*, which defines a separate *init()* method. When Funambol loads a *LazyInitBean*, after bean instantiation (or deserialization) and configuration (calling the setter methods), it calls the bean's *init()* method, giving the bean the opportunity to complete its initialization.

9.3. How to configure a standard component

Making a change to a configuration bean is as easy as editing a text file. Let's take as example the configuration file for the *DBOfficer* component. The configuration bean full path is *com/funambol/server/security/DBOfficer.xml* (please note that this path is relative to the config path) and its content is below:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.0" class="java.beans.XMLDecoder">
 <object class="sync4j.server.security.DBOfficer">
  <void property="clientAuth">
   <string>syncml:auth-basic</string>
  </void>
  <void property="serverAuth">
   <string>none</string>
  </void>
 </object>
</java>
```

The object element specifies which Java class will be instantiated and the property element sets the corresponding instance property. Therefore, to change the preferred client authentication type, it is sufficient to edit the file, change the *clientAuth* property and save. The next time this bean will be used, the new configuration value will be picked up.

9.4. How to create a custom configurable object

With this technique, any Java object can be configured, from a simple Java class to a very complex Java object tree. For example, this configures a String object:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.2" class="java.beans.XMLDecoder">
 <string>This is a String!</string>
</java>
```

A more interesting example is given, for instance, by the class *com.funambol.framework.config.LoggingConfiguration*. The class looks like the following:

```
public class LoggingConfiguration {
```



```
    // ------------------------------------------------------------ Private data
    private ArrayList loggers;
    // ------------------------------------------------------------ Constructors
        /** Creates a new instance of LoggingConfiguration */
    public LoggingConfiguration() {
    }
    /**
     * Getter for property loggers.
     *
     * @return Value of property loggers.
     */
    public ArrayList getLoggers() {
        return loggers;
    }
    /**
     * Setter for property loggers.
     *
     * @param loggers New value of property loggers.
     */
    public void setLoggers(ArrayList loggers) {
        this.loggers = loggers;
    }
}
```

A possible configuration file for such a class could be:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.2_04" class="java.beans.XMLDecoder">
 <object class="sync4j.framework.config.LoggingConfiguration">
  <void property="loggers">
   <object class="java.util.ArrayList">
    <!--
         funambol
    -->
    <void method="add">
     <object class="com.funambol.framework.config.LoggerConfiguration">
      <void property="append">
       <boolean>true</boolean>
      </void>
      <void property="count">
       <int>1</int>
      </void>
      <void property="fileOutput">
       <boolean>true</boolean>
      </void>
      <void property="level">
       <string>INFO</string>
      </void>
```



```
      <void property="limit">
       <int>100</int>
      </void>
      <void property="name">
       <string>funambol</string>
      </void>
      <void property="pattern">
       <string>logs/ds-server.log</string>
      </void>
     </object>
    </void>
    <!--
         funambol.engine
    -->
    <void method="add">
     <object class="com.funambol.framework.config.LoggerConfiguration">
      <void property="append">
       <boolean>true</boolean>
      </void>
      <void property="count">
       <int>1</int>
      </void>
      <void property="inherit">
       <boolean>true</boolean>
      </void>
      <void property="level">
       <string>INFO</string>
      </void>
      <void property="limit">
       <int>100</int>
      </void>
      <void property="name">
       <string>funambol.engine</string>
      </void>
      <void property="pattern">
       <string>logs/ds-server.engine.log</string>
      </void>
     </object>
    </void>
   </object>
  </void>
 </object>
</java>
```

Note: see later how to automatically create such a file.

9.5. How to get a configured instance

Configuration beans are accessed through the singleton *com.funambol.framework.config.Configuration* object. For example, to instantiate a configured *LoggingConfiguration* instance, use the code below.

```
Configuration c = Configuration.getConfiguration();

LoggingConfiguration logging =
c.getBeanInstanceByName("sync4j/server/logging/logging.xml");
```

9.5.1. Tips and tricks

It is not necessary to write a configuration file by hand from scratch. To write a bean instance for the first time use the *com.funambol.framework.tools.beans.BeanFactory*'s *saveBeanInstance()* method to save a configured instance into a file. For example:

```
Jbutton b = new Jbutton(“press me”);

BeanFactory.saveBeanInstance(b, new File("button.xml");
```

The result is the following:

```
<?xml version="1.0" encoding="UTF-8"?>

<java version="1.4.2_04" class="java.beans.XMLDecoder">

 <object class="javax.swing.JButton">

  <string>press me</string>

 </object>

</java>
```


10. Customizing message processing

This section explains how to extend Funambol customizing the processing of incoming and outgoing messages.

10.1. Overview

The OMA DS protocol is an XML-based client-server protocol. This means that each OMA DS message is an XML document and a response message always follows a request message.

In the Funambol implementation, an OMA DS message reaching the server goes through some transformations. These may be:

- XML level transformations, processing the message in its native XML representation, and
- message transformations, acting on a Java representation of the message

Figure 14 helps explaining this process.

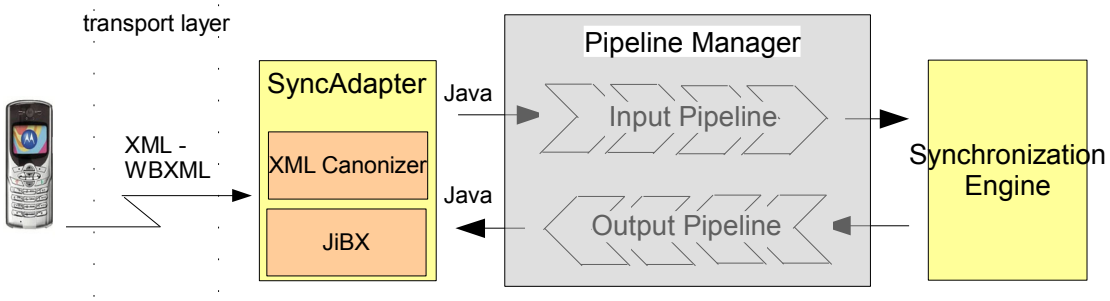


Figure 14: Message processing architecture

In order to save bandwidth and processing power, OMA DS messages can be also WBXML encoded. Regardless of how the message is encoded, its content is first delivered to a *SyncAdapter* component by the transport layer. The *SyncAdapter* first transforms the message into XML if it was WBXML encoded and then the XML message is reduced to a “canonical” form in order to get rid of device specific singularities. XML canonization is the standard XML level transformation executed by the system.

Even when in the canonical XML form, the message is still hard to manipulate, since XML needs to be parsed. Plus, each component that needs to access any of the OMA DM message elements would have to parse the XML again, with a big impact on performance. For these reasons, the canonic XML message is transformed into an object tree that represents exactly the message.

After an incoming message has been transformed into an object tree, it passes through the input message processing pipeline before it gets to the synchronization engine. This gives the opportunity of further processing the message when it is in a more manageable representation. In a similar way, a response message going out from the engine, passes through the output message processing pipeline before getting transformed to its XML (and then WBXML) representation.

The input and the output pipelines are completely customizable, so that custom message pre and post processing can be easily added to the system.

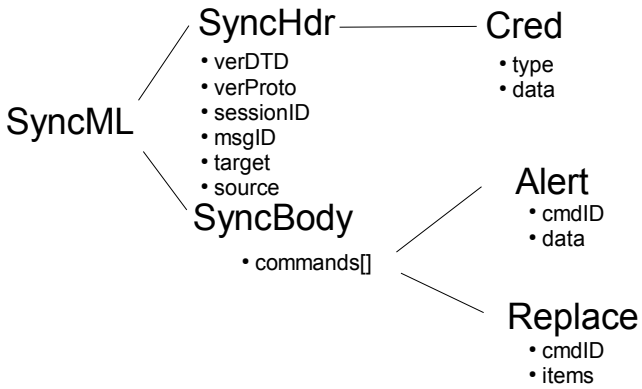
Input and output message processing components are also called *synclets*.

10.2. Preprocessing an incoming message

To preprocess an incoming message we have to create an input processor component (input synclet) and to configure the pipeline manager accordingly. This is described below..

10.2.1. Creating an input synclet

An input synclet is a class that implements the *com.funambol.framework.engine.pipeline.InputMessageProcessor* interface. This interface defines just one method: *preProcessMessage(MessageProcessingContext context, SyncML msg)*. context is a parameter that is shared amongst all the synclets (both input and output) involved in the message processing. msg is the object tree representing the SyncML message. The object tree is composed of instances of classes in the *com.funambol.framework.core* packages and represents a hierarchical view of the message.



For example, the synchronization message below will be translated in the object hierarchy of Figure 15.

```
<SyncML>
<SyncHdr>
<VerDTD>1.1</VerDTD>
<VerProto>SyncML/1.1</VerProto>
<SessionID>12345678</SessionID>
<MsgID>2</MsgID>
<Target><LocURI>http://localhost</LocURI>
</Target><Source><LocURI>syncml-phone</LocURI></Source>
<Cred>
 <Meta><Type>syncml:auth-basic</Type></Meta>
 <Data>Z3Vlc3Q6Z3Vlc3Q=</Data>
</Cred>
</SyncHdr>
<SyncBody>
<Alert>
<CmdID>1</CmdID>
<Data>200</Data>
<Item>
<Target><LocURI>test</LocURI></Target>
<Source><LocURI>test</LocURI></Source>
<Meta>
<Anchor>
<Last>234</Last>
<Next>276</Next>
```



```
</Anchor>
</Meta>
</Item>
</Alert>
<Final/>
</SyncBody>
</SyncML>
```

An example of an input synclet is the following.

```
package com.foo.synclet;
import com.funambol.framework.logging.*;
import com.funambol.framework.core.*;
import com.funambol.framework.engine.pipeline.*;
import com.funambol.framework.tools.SyncMLUtil;

public class LoggingSynclet
implements InputMessageProcessor {
// ------------------------------------------------------------ Private data
private static final FunambolLogger log = FunambolLoggerFactory.getLogger("engine");
// ------------------------------------------------------------ Public methods
    /**
     * Logs the input message and context
     *
     * @param processingContext the message processing context
     * @param message the message to be processed
     *
     * @throws Sync4jException
     */
    public void preProcessMessage(MessageProcessingContext processingContext,
                                  SyncML                    message)
    throws Sync4jException {
       if (log.isLoggable(Level.INFO)) {
            log.info("------------------------------------------------------------");
            log.info("Input message processing context");
            log.info("");
            log.info(processingContext.toString());
            log.info("------------------------------------------------------------");
            log.info("Input message");
            log.info("");
            log.info(Util.toXML(message));
            log.info("------------------------------------------------------------");
            //
            // Sets the device ID to foo
            //
            message.getSyncHdr().getSource().setLocURI("foo");
        }
    }
}
```


This example synclet just logs to the *funambol.engine* logger the input message; also, it modifies the message setting the device ID to “foo”.

One very important thing to keep in mind when developing a synclet (input or output) it is that in the current Data Synchronization Service implementation just one instance is used to serve all the requests and so no instance/class variables must be used to store its status. Indeed the *MessageProcessingContext* must be used.

10.2.2. Configuring an input synclet

The input synclet so created is configured and deployed in the Data Synchronization Service adding a server side JavaBeans like the following in the *ds-server/config/com/funambol/server/engine/pipeline/input* directory.

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.5.0" class="java.beans.XMLDecoder">
 <object class="com.foo.synclet.LoggingSynclet" />
 </object>
</java>
```

Note: any XML file in the *ds-server/config/com/funambol/server/engine/pipeline/input* directory is used to create an input synclet, so this directory should be under strict control by the system administrator. The synclet loading and execution is done in alphabetic order, so if more than one synclet is used, a naming convention is an important part of processing control.

10.3. Postprocessing an outgoing message

To postprocess an outgoing message we have to create an output processor component (output synclet) and to configure it accordingly. This is described below.

10.3.1. Creating an output synclet

An output synclet is a class that implements the *com.funambol.framework.engine.pipeline.OutputMessageProcessor* interface. This interface defines just one method: *postProcessMessage(MessageProcessingContext context, SyncML msg).*

The concepts behind the output message processing are the same as per input processing.

An example of an output synclet is the class shown below. This synclet injects a Get command into the outgoing message to request client capabilities:

```
package com.foo.synclet;
public class AddGetSynclet
implements OutputMessageProcessor {
    // ------------------------------------------------------------ Constants
    // ---------------------------------------------- OutputMessageProcessor
    public void postProcessMessage(MessageProcessingContext processingContext,
                                   SyncML                   message          )
    throws Sync4jException {
        AbstractCommand[] commands = message.getBody().getCommands();
        AbstractCommand[] newCommands = new AbstractCommand[commands.length+1];
        Meta meta = new Meta();
        meta.setType(“application/vnd.syncml-devinf+xml”);
        Item item = new Item(
            new Target(“/devinf11”),
            null,
            null,
```



```
            null,
            false
        );
        Get get = new Get(
            new CmdID(newCommands.length),
            false,
            null,
            null,
            meta,
            new Item[] { item }
        );
        System.arraycopy(commands, 0, newCommands, 0, commands.length);
        newCommands[commands.length] = get;
    }
}
```

10.3.2. Configuring an output synclet

As for the input synclet, the output synclet so created is configured and deployed in the Data Synchronization Service adding a server side JavaBeans like the following in the *ds-server/config/com/funambol/server/engine/pipeline/output* directory.

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.5.0" class="java.beans.XMLDecoder">
 <object class="com.foo.synclet.AddGetSynclet" />
 </object>
</java>
```

Note: any XML file in the *ds-server/config/com/funambol/server/engine/pipeline/output* directory is used to create an input synclet, so this directory should be under strict control by the system administrator. The synclet loading and execution is done in alphabetic order, so if more than one synclet is used, a naming convention is an important part of processing control.

10.3.3. The MessageProcessingContext

Funambol uses the *MessageProcessingContext* mentioned above to store the following properties:

Property name	*Type*	*Scope*	*Description*
funambol.session.id	java.lang.String	Session	Session ID of the current session
funambol.session.messageType	java.lang.String	Session	'WBXML' or 'XML' accordingly to the type of the messages in the current session
funambol.request.parameters	java.util.Map	Request	Parameters passed on the query string of the URL used at the transport level
funambol.request.headers	java.util.Map	Request	Headers passed in the request at the transport level.

10.3.4. How to stop message processing

If needed, an input or output pipeline component can ask the manager to stop further processing of the message. The synclet does this by throwing a *com.funambol.framework.pipeline.StopProcessingException* exception.

11. SyncSource API

The SyncSource identifies the remote database that will be synchronized with the client database; it wraps a set of items to be synchronized. Any type of data (files, database tables, calendar events and so on) can be synchronized, but there must be a proper SyncSource for each type, capable of extracting and storing the data for a real data store.

The goal of Funambol is to provide a collection of SyncSources for the most common uses (for instance, files), although SyncSources can be independently developed and plugged in the synchronization engine, allowing synchronization requests targeted to virtually any database or type of data.

11.1. SyncSource class

The SyncSource class provides an implementation of the main methods; then, the derived classes *MergeableSyncSource* and the *FilterableSyncSource* implement additional specific methods.

Figure 16 shows the class diagram of a *DummySyncSource* that implements *MergeableSyncSource.*

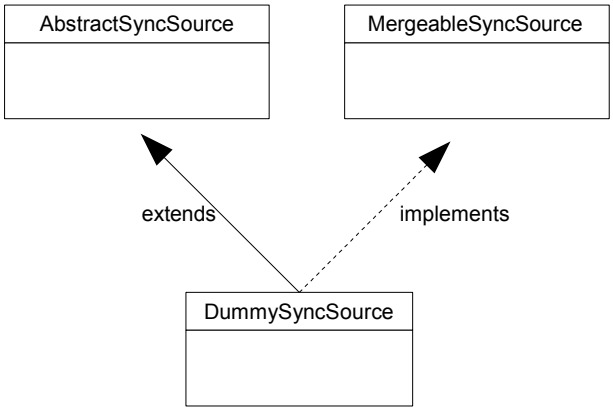
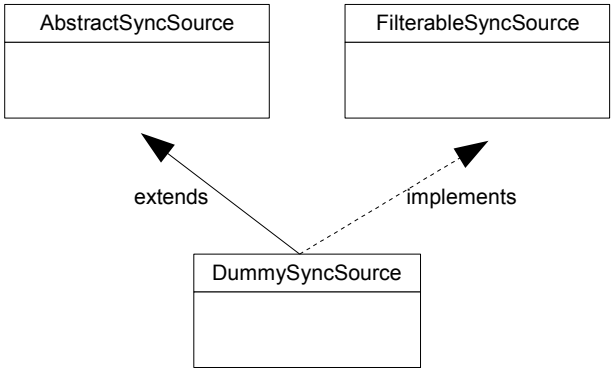


Figure 17 shows the class diagram of a *DummySyncSource* that implements *FilterableSyncSource.*



11.1.1. Methods list

Note: All of these methods throw an exception: *SyncSourceException*.

Method	*Return*	*Description*
beginSync(SyncContext context)	void	Init method of the synchronization process
endSync()	void	final method of the synchronization process
getAllSyncItemKeys()	SyncItemKey[]	Get all the IDs of the items in the back end
getSyncItemFromId(SyncItemKey syncItemKey)	SyncItem	Get the item associated to the given key from the Back end system
getNewSyncItemKeys(Timestamp since, Timestamp to)	SyncItemKey[]	Get the IDs of the new items from the given since timestamp to the given to timestamp
getUpdatedSyncItemKeys(Timestamp since, Timestamp to)	SyncItemKey[]	Get the IDs of the updated items from the given since timestamp to the given to timestamp
getDeletedSyncItemKeys(Timestamp since, Timestamp to)	SyncItemKey[]	Get the IDs of the deleted items from the given since timestamp to the given to timestamp
getSyncItemKeysFromTwin(SyncItem syncItem)	SyncItemKey[]	Get the IDs of the items that match with the given item
addSyncItem(SyncItem syncItem)	SyncItem	Called by the synchronization engine to add the given item to the back end. syncItem contains the item to add. The SyncSource developer can use SyncSource methods to extract the real item content and perform the appropriate actions on the back end.
updateSyncItem(SyncItem syncItem)	SyncItem	Called by the synchronization engine to update the given item to the back end. syncItem contains the item to add. The SyncSource developer can use SyncSource methods to extract the real item content and perform the appropriate actions on the back end.
removeSyncItem(SyncItemKey syncItemKey, Timestamp ts, boolean softDelete)	void	Called by the synchronization engine to delete the given item from the back end. syncItem contains the key of the item to delete.
setOperationStatus(String operation, int statusCode, SyncItemKey[] keys)	void	This call notifies the status code of the given operation performed on the items with the given keys.

SyncItem is a class in the package: import *com.funambol.framework.engine.SyncItem*. The methods available in this interface are reported in the following table:

Method	*Description*
getKey	Returns the item key.
getParentKey	Returns the item's parent key (hierarchic sync).
getState	Returns the item state. (see below)
setState	Sets the item state. (see below)
getContent	Returns the item's content.
setContent	Sets the item's content.
getFormat	Returns the item's encoding. (“b64” for SIF, nothing in most other cases)
setFormat	Sets the item's encoding. (“b64” for SIF, nothing in most other cases)
getType	Returns the item's MIME type.
setType	Sets the item's MIME type.
getTimestamp	Returns the timestamp of the last change of the item state and it is used in the synchronization process, in order to determine the operation to be performed on the sources.
setTimestamp	Sets the item timestamp.
getSyncSource	Returns the SyncSource the item belongs to.

The *key* and *parentKey* properties are *com.funambol.framework.engine.SyncItemKey* objects identifying the item or the parent item. The parent item is used in the case of hierarchical synchronization where a parent/child structure is synchronized (e.g. email synchronization). In many cases a *SyncItemKey* just encapsulates a string.

The *state* (*getState*, *setState*) property represents the state of the item and can be one of the following values:

Value	*Description*
SyncItemState.NEW	The item is new.
SyncItemState.UPDATED	The item has been updated.
SyncItemState.DELETED	The item has been deleted.
SyncItemState.SYNCHRONIZED	The items has been already synchronized.
SyncItemState.UNKNOWN	The item is in a unknown state.
SyncItemState.NOT_EXISTING	The item is not existing yet.
SyncItemState.CONFLICT	The item is in a conflict state.
SyncItemState.PARTIAL	The item contains only a portion of the content.

Note that only NEW, UPDATED and DELETED items are available for use by a SyncSource. The other states are only used and processed by the synchronization engine. The associated Timestamp property is the timestamp of the last change.

Content is the item content returned encapsulated into a object. The MIME type of the content is represented by the property *type* (*getType*, *setType*). The following table lists some examples of MIME types supported by some of our SyncSources:

MIME type	*Content*
text/x-vcalendar	A calendar (event or task) item in the vCalendar 1.0 format.
text/calendar	A calendar (event or task) item in the iCalendar (i.e. vCalendar 2.0) format.
text/x-s4j-sife	An event in the SIF-E format.
text/x-s4j-sift	A task in the SIF-T format.
text/x-vcard	A contact in the vCard format.
text/x-s4j-sifc	A contact in the SIF-C format.
text/plain	A plain-text note.
text/x-s4j-sifn	A note in the SIF-N format.
application/vnd.omads-email+xml	An electronic mail message in the format defined by the Open Mobile Alliance.
application/vnd.omads-folder+xml	An electronic mailbox folder in the format defined by the Open Mobile Alliance.
…	...

As a developer you do not usually need to develop your own implementation of *SyncItem*; Funambol provide a default implementation that should be sufficient in most cases. This is *com.funambol.framework.engine.SyncItemImpl*. See the source code of one of the Funambol connector for examples of how to use this class.

11.2. Mergeable SyncSource methods

The *MergeableSyncSource* class is an extension of a SyncSource that allows to merge the content of two items when a conflict is detected in order to avoid a loss of whatever information. For instance, while synchronizing the contacts, the server could find out that the same contact was updated on the server and on the client. An example could be that a new email address has been added in the client, while a telephone number was added in the server image of the record. In this case, the server can merge the server contact information and the client contact information, keeping both the new email address and phone number.

11.2.1. Methods list

Note: all the methods throw a exception: *SyncSourceException*

Method	*Return*	*Description*
mergeSyncItems(SyncItemKey syncItemKey, SyncItem syncItem)	boolean	Called by the synchronization engine to merge the server item identified by syncItemKey with the content obtained from the client and stored in syncItem. This methods must return true if the content has been changed so that the item resulting from the merge will be sent back to the client.

11.3. Filterable SyncSource methods

The *FilterableSyncSource* class is an extension of a SyncSource that allows the handling of filters during the sync process. Two type of filters can be used by a device:

- Record filter: permits to specify the records to sync (for example, only today's emails)
- Field filter: permits to specify criteria on the fields to synchronize (for example, skip the contact's photo)

11.3.1. Methods list

Note: all the methods throw a exception: *SyncSourceException*

Method	*Return*	*Description*
isSyncItemInFilterClause(SyncItem syncItem)	boolean	Detect if the given item matches the filter clause. Called to know if an item satisfies the filter

isSyncItemInFilterClause(SyncItemKey syncItemKey)	boolean	Detect if the item linked to the ID matches the filter clause. Called to know if the item specified with the given key satisfies the filter
getSyncItemStateFromId(SyncItemKey syncItemKey)	char	Get the “status” of the given item. Called to retrieve the status of an item. This method is required because the server isn't able to know the status of an item not inside the filter criteria.

12. Officer API

The Funambol security architecture is designed to be pluggable and is based on a very simple concept: authentication and authorization are centralized in a single dedicate component called officer.

An officer is a Java class that implements a specific interface. Concrete implementations provide adapters for external security services. For instance, a common external security service could be a database storing user profile information; a DBOfficer can be plugged into Funambol in order to perform authentication and authorization through the user information stored into the database.

12.1. Officer class

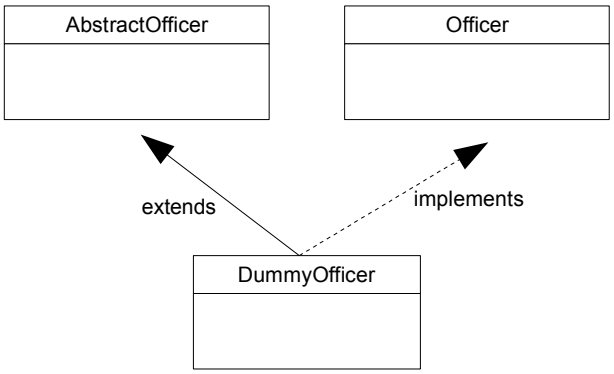


Figure 18 shows the class diagram of a *DummyOfficer*.

12.1.1. Methods list

Method	*Return*	*Description*
authenticateUser(Cred credential)	Sync4jUser	This method verifies the user on the back-end system, logging her in.
authorize(Principal principal, String resource)	Officer.AuthStatus	If required, this method authorizes the user for a specific purpose.
unauthenticateUser(Sync4jUser user)	void	If required, this method makes the user log out from the back-end system.

Cred is a class in the package: *com.funambol.framework.core.*

A credential is created on the basis of a *com.funambol.framework.core.Authentication* object, containing all the information needed to authenticate the user on a back-end system. Four types of authentication are supported:

- *none*
- *syncml:auth-basic*
- *syncml:auth-md5*
- *syncml:auth-MAC*

Principal is a class in the package: *java.security.Principal.*

The possible statuses of authorization are:

- *RESOURCE_NOT_AVAILABLE*
- *INVALID_RESOURCE*
- *NOT_AUTHORIZED*
- *AUTHORIZED*
- *PAYMENT_REQUIRED*

12.1.2. Usage example

The following code (from the Connector Testing Framework) provides a simple example of the authentication process:

```
// Context set-up

Authentication authentication =
new Authentication("syncml:auth-" + authenticationType, userName, password);

Cred credentials = new Cred(authentication);

Sync4jUser user = officer.authenticateUser(credentials);

Sync4jDevice device = new Sync4jDevice(deviceID);

Sync4jPrincipal principal = new Sync4jPrincipal(user, device);

SyncContext context = new SyncContext(principal, 200, null, "localhost", 2);
```


13. Web Services API

In order to solve the problem of application-to-application communication, Funambol provides a web services layer that allows for applications to be integrated with the Funambol Platform and to access the resources provided by the system.

The application integration becomes much more flexible because web services provide a form of communication that is not tied to any particular platform or programming language. Thanks to the web service, Funambol makes resources available over the networks in a standardized fashion.

13.1. Introduction

The Funambol Data Synchronization Service exposes the web services listed in the next paragraphs. The endpoint of the web service is: http://localhost:8080/funambol/services/admin.

In order to access the web service, the client must be authenticated. The default credentials are:

user = **admin**

password = **sa**

13.1.1. Funambol Data Synchronization Service Web Services

The following table lists the web services exposed by the Funambol Data Synchronization Service.

A clause, used as a parameter in several methods of this list, is an XML element that represents a logical clause used to filter a list of results. It is generated by a Clause object as explained in the Funambol Data Synchronization Service Architecture and Design Document (section *FilterableSyncSource*).

Method	***Parameters***	***Return***	***Description***
getServerVersion	String version	String	Returns the server version.
setServerConfiguration	ServerConfiguration config	void	Saves and apply the new server configuration.
getServerConfiguration		ServerConfiguration	Returns the server configuration.
getServerBean	String bean	String	Returns the server bean with the given name. This is used to load a configuration as explained in the Funambol Data Synchronization Service Architecture and Design Document (section *Server JavaBeans*). If the bean does not exist, an AdminException is thrown.
setServerBean	String bean, String obj	void	Sets the server bean with the given name. This is used to set up a configuration.
login	String username	void	This method is only used to check credentials. Being this WS stateless, its methods are always called on request and each of them has to pass authentication. However, for instance the SyncAdmin, has a login panel that is displayed before any access to one of the other WS methods. In order to provide to the user an immediate feedback, the SyncAdmin (or any other client can just call login()
authorizeCredential	Credential authCred, String resource	AuthorizationResponse	Checks if the user is authorized to use the given resource. The credential comprises the user name and the corresponding password.
getUsers	String clause	Sync4jUser[]	Gets all users that satisfy the parameter

			of search.
countUsers	String clause	int	Counts the number of users that satisfy a given clause.
addUser	Sync4jUser user	void	Adds a new user with their assigned role.
setUser	Sync4jUser user	void	Updates the information about a user.
deleteUser	String username	void	Deletes the user from the users list.
getDevices	String clause	Sync4jDevice[]	Gets a list of devices that satisfy a given clause.
countDevices	String clause	int	Counts the number of device that satisfy a given clause.
addDevice	Sync4jDevice d	String	Inserts a new device in the devices list and returns its newly assigned device ID.
setDevice	Sync4jDevice d	void	Updates the information about a device.
deleteDevice	String deviceID	void	Deletes the device with the given ID from the devices list.
getDevice	String deviceID	Sync4jDevice	Retrieves the device with the given ID from the devices list.
getDeviceCapabilities	String deviceID	String	Gets a device's capabilities as a SyncML node.
setDeviceCapabilities	String caps, String deviceID	Long	Sets a device's capabilities for a specific device and returns the identifier of the newly added capabilities.
getPrincipals	String clause	Sync4jPrincipal[]	Gets all principals that satisfy a given clause. A principal comprises references to a user and a device.
countPrincipals	String clause	int	Counts the number of principals that satisfy a given clause.
addPrincipal	Sync4jPrincipal p	long	Adds a new principal and returns the newly assigned principal ID.
deletePrincipal	long principalID	void	Deletes a principal from the principals list.
getRoles		String[]	Gets the list of available roles for users.
getLoggers		LoggerConfiguration[]	Returns the available configurations for the logger.
getAppenders		String	Returns an XML serialization of a map with all available Appender objects on the server.
setAppender	String appenderBean	void	Sets the given appender. The configuration of the appender is specifiied as a JavaBean.
getAppenderManagementPanel	String appenderClassName	String	Returns the class name of the management panel to be used to configure an appender with the given class name.
setLoggerConfiguration	String loggerBean	void	Saves and applies the new logger configuration. The configuration is specified as a JavaBean.
setLoggingConfiguration	LoggingConfiguration config	void	Saves and applies a new logging configuration.
getLatestDSServerUpdate		Component	Returns the latest available Data Synchronization Service update. The Component class contains information about the update: the updated component's name, version, release date, URL, short and long descriptions. See the Funambol Data Synchronization Service Architecture and Design Document (section *Automatic check for updates*).
getLastTimestamps	String clause	LastTimestamp[]	Reads all sync timestamps that satisfy the clause.

deleteLastTimestamp	long principalId, String sourceId	void	Deletes the timestamp of the last synchronization performed by a given principal with a given sync source.
countLastTimestamps	String clause	int	Counts the number of sync timestamps that satisfy a given clause.
getModulesName		Sync4jModule[]	Gets a list of all modules installed on the server.
getModule	String moduleID	String	Gets information about a module installed on the server.
addSource	String moduleId, String connectorId, String sourceTypeId, String source	void	Adds a new source into the datastore and creates the corresponding XML file with its configuration. The source must have a defined source type. The source type must refer to a connector. The connector must refer to a module.
getSync4jSources	String clause	Sync4jSource[]	Gets a list of sync sources that satisfy a given clause.
getSyncSourceClasses	String[] sourceTypesID	String[]	Returns an array with the classes used for the given source types.
deleteSource	String sourceUri	void	Removes a sync source and the corresponding configuration file.
setSource	String moduleId, String connectorId, String sourceTypeId, String source	void	Updates a specific source into the datastore and creates the corresponding XML file with its configuration. The source must have a defined source type. The source type must refer to a connector. The connector must refer to a module.
sendNotificationMessage	String deviceID, String alerts, Integer uimode	void	Sends a notification message to the given device. The *uimode* argument specifies the “user interaction mode” according to the SyncML notification standard that can have four values: 0 – unspecified, 1 – background action, 2 – inform the user, 3 – require user interaction. See OMA document about SyncML Notification Initiated Session.
sendNotificationMessages	String username, String alerts, Integer uimode	void	Sends a notification message to all devices of the principals with the given username.

Below is an example of the notification message needed by the *sendNotification* method of the WS API; the “*alert*” parameter of the *sendNotificationMessage* method listed in the previous table is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.5.0_15" class="java.beans.XMLDecoder">
  <array class="com.funambol.framework.core.Alert" length="1">
  <void index="0">
   <object class="com.funambol.framework.core.Alert">
    <void property="cmdID">
     <object class="com.funambol.framework.core.CmdID"/>
    </void>
    <void property="data">
     <int>206</int>
    </void>
    <void property="items">
     <void method="add">
      <object class="com.funambol.framework.core.Item">
       <void property="meta">
```



```
        <object class="com.funambol.framework.core.Meta">
         <void property="metInf">
          <void property="type">
           <string>application/vnd.omads-email+xml</string>
          </void>
         </void>
        </object>
       </void>
       <void property="target">
        <object class="com.funambol.framework.core.Target">
         <void property="locURI">
          <string>mail</string>
         </void>
        </object>
       </void>
      </object>
     </void>
    </void>
   </object>
  </void>
 </array>
</java>
```

For the details of the logging system used by Funambol and the usage of classes like *LoggerConfiguration* and *LoggingConfiguration*, please see the Funambol Data Synchronization Service Architecture and Design Document, section *Logging*.

The installation of the Email Module adds the web services listed in the following table. The endpoint of the email module web service is: http://localhost:8080/funambol/services/email.

Method	***Parameters***	***Return***	***Description***
getUsers	String clause	MailServerAccount[]	Gets a list of users that matches the filter clause.
getUser	String username	MailServerAccount	Gets the user with the given username
getUserFromID	long accountID	MailServerAccount	Gets the user with the given account ID.
insertUser	MailServerAccount msa	int	Adds a user and the corresponding mail server configuration.
updateUser	MailServerAccount msa	int	Updates a user and the corresponding mail server configuration.
disableUser	long accountID	int	Disables an account.
enableUser	long accountID	int	Enables an account.
markUserAsDelete	long accountID	int	Deletes the user from the *fnbl_email_account* table and sets them as 'D' in the *fnbl_email_push_registry* table.
checkAccount	MailServerAccount msa, Integer timeout	MailServerError	Checks the account on the mail server. If no answer is received before the time-out, the account is regarded as not verified.
insertPubMailServer	MailServer ms	int	Adds a public mail server configuration to the list.
deletePubMailServer	String mailServerID	int	Deletes a public mail server configuration with a given mail server ID.
updatePubMailServer	MailServer ms	int	Updates a public mail server configuration.
getPubMailServers	String clause	MailServer[]	Gets the list of the public mail servers that

			satisfy the filter clause.
getPubMailServerFromID	String mailServerID	MailServer	Gets a public mail server configuration.
getCachedInfo	String username, String protocol	SyncItemInfoAdmin[]	Get the cached information from the local inbox folder (*fnbl_email_inbox*) for the specified username using the specified mail protocol.
getImapFolders	MailServerAccount msa	Object[]	Returns all the folders names for the given IMAP account.

14. Localizing Funambol clients

This section describes the steps needed to translate the strings that are shown to users of Funambol clients into different languages. Each client uses the localization framework of the platform it is based on, so in order to localize the various Funambol clients a translator should follow different steps, depending on the target client platform.

The clients covered by this document are:

1. Windows Mobile Sync Client
2. Outlook Sync Client
3. Java ME Email Client
4. BlackBerry Sync Client
5. iPod Sync Client
6. iPhone/iPod Touch Sync Client
7. Symbian Sync Client

14.1. General considerations

The following sections provide some general considerations, common to all clients, related to their localization.

14.1.1. Strings length

In the translation process, the length of the strings is very critical. The strings on mobile devices are often shown on displays with a limited space, so in many cases the English wording already fills the available space.

There are languages, such as German, with words in average 20% longer than English words. Translating the sentences without keeping the available space in mind will probably lead to errors on the device (for example, cut words).

As a rule of thumb, the translation should take the same space on the screen as the original (space also depends on individual characters, since 'l' is thinner than 's') for single-word entries (e.g. the menu items). Messages shown in an alert box, such as error messages, allow more flexibility. This is because generally there is extra space or scrollbars available; but also in this case the increase in text should not exceed 15%.

If the translator has access to the device running the client to be translated, it can be useful in some cases to see where the word is shown on the screen to understand whether there is room for more characters.

14.1.2. Coherence among clients

In some languages, more words are needed to translate a single word in English, or the same message can be expressed using different words.

If the same word or sentence can be reused on different clients, it's a big advantage to do so. If a translation in a particular language is available for another client, check the wording used there before using a different word or sentence. If you feel that the translation is not correct, please contact the author or raise the issue on the Funambol user forum.

14.2. Windows Mobile Sync Client

The Funambol Windows Mobile Sync Client is written in C++ using the tools available in the Microsoft

Visual Studio 2005 (and later) development environment. The localization of the client should be accomplished using the tools and procedures offered by that environment.

The localization is done using Windows String Tables. To easily support different languages on the same package, there are separate Visual Studio projects for each language (under *localization*, currently *EN*, *DE* and *IT* are available – the Italian version has not been tested or released). The correct one will be loaded at runtime, if available, with default to English.

14.2.1. Languages currently available

Language	*DLL name*	*Status*
English	language-en.dll	Default language, officially maintained in the product.
German	language-de.dll	Used for one customer, bundled in the product.
Italian	language-it.dll	Available in the code repository but not maintained.

14.2.2. How to add a new language

Using Visual Studio 2005 you can modify messages and labels from the resource files to obtain a customized environment for a specific language.

The localization is done using Windows String Tables. To easily support different languages on the same package, there are separate Visual Studio projects for each language (under the *localization* folder). The correct one will be loaded at runtime, if available, with default to English.

To add a new language, the easiest way is to duplicate one existing language project and then translate all strings into the new language, from Visual Studio string tables. The resources are different for PocketPC (*FunLanguageppc.rc*) and for Smartphone (*FunLanguagesp.rc*) targets, so each language project has 2 different string tables. However, they mostly overlap and the actual strings are largely the same.

Below is a list of the steps to create a new language project (FR) from an existing one (EN):

- Go to the *language* directory
- Copy the *EN* folder and rename it to *FR*
- Go to the *language\FR\language\build* directory
- Rename file *FunLanguage-en.vcproj* to *FunLanguage-fr.vcproj*
- Open the file *FunLanguage-fr.vcproj* with a text editor, locate and change the line:

  ```
  Name="FunLanguage-en
  ```

  to:

  ```
  Name="FunLanguage-fr
  ```

  in the first lines of the file
- Open the *WMPlugin* solution with Visual Studio 2005
- Select *File* > *Add* > *Existing Project...* from the Visual Studio menu
- Browse and select the new project, *FunLanguage-fr.vcproj* (*)
- Right click on the *FunLanguage-fr* project and select Properties from the drop-down menu
- Click on *Configuration Properties* > *Linker* on the left menu
- Click on the *Platform:* drop-down list and select *All Platforms*
- Change the *Output File* value to:

  ```
  $(PlatformName)\$(ConfigurationName)/language-fr.dll
  ```

- Click OK

- Select *File* > *Save All* to close the Solution (*)
- Open the *WMPlugin* solution again with Visual Studio 2005
- Open the *Resource View* for the project *FunLanguage-fr*
- Under *FunLanguageppc.rc* and *FunLanguagesp.rc* there are String Tables with all strings to be translated into the new language

(*) Visual Studio may display some 'unspecified errors' at this point. This may happen because the new project configuration is not complete yet; do not worry and click OK.

The new language project will be built together with the other project of the Solution.

The final step is to include the new output file (*language-fr.dll*) in the packages for PocketPC and Smartphone.

For PocketPC

- Go to the *install\pocketpc\build* directory
- Open the build file *ppcwm5.xml*
- All required files are copied under *forge-no-checkout-release-wm5* target. Add the lines:

```
<copy todir="${dir.files}">
             <fileset   dir="../../localization/FR/language/build/${wm.targetppcwm5}/${wm.configuration-release}/">
        <include name="language-fr.dll"/>
    </fileset>
</copy>
```

- Save and close the file
- Open the file *setup-ppc-wm5.inf* and add the lines:

```
language-fr.dll=1
```

under the *[SourceDisksFiles]* section, and

```
language-fr.dll, language-fr.dll
```

under the *[CopyToProgramFiles]* section

- Save and close the file

For Smartphone

The steps are the same.

The build file to modify is *\install\smartphone\build\sphwm5.xml* and the inf file is *\install\smartphone\build\setup-sph-wm5.inf*.

14.3. Outlook Sync Client

The Funambol Outlook Sync Client is written in C++ using the tools available in the Microsoft Visual Studio 2005 (and later) development environment. The customization of the Plugin should be accomplished using the tools and procedures offered by that environment.

The localization is done using Windows String Tables. In the same project, it is possible to define different string tables, and the system will use the one corresponding to the system language or English by default. Currently, only English and a test Romanian and Italian versions (not delivered) are available.

14.3.1. Languages currently available

Language	*Status*
English	Default language, officially maintained in the product.
French	Used for one customer, not officially maintained yet.

14.3.2. How to add a new language

Using Visual Studio 2005 you can modify messages/labels from the resource file of the *OutlookPlugin* project to obtain a customized environment for a specific language:

- Open the *OutlookPlugin* solution
- Open the resource file for the desired target (in *Resource View*, under the *OutlookPlugin* project)
- Add a new string table for the desired language (right click and select *Insert copy* from the drop-down menu, then choose the language)
- Change all the messages you want to customize from the new string table
- Build the application and run it on a PC with the Windows OS of the same language

By default, the application will choose the resources corresponding to the language of the current Operating System; if some resources are not found, the default ones are used (English).

14.4. Java ME Email Client

The Funambol Java ME Email Client is written in Java for memory constrained devices based on MIDP 2.0 and CLDC 1.x.

The design goal of reducing the application size to the minimum, to run the application on as many devices as possible, also impacts the localization model.

The client also has a text help, which differs for various phone manufacturers and models.

14.4.1. Languages currently available

Language	*File name*	*Status*
English	UI/basic/res/localization/EN/language.properties	Default language, officially maintained in the product.
Italian	UI/basic/res/localization/IT/language.properties	Available in the repository but not maintained.

14.4.2. How to add a new language

The localized strings are contained in a resource file, which is processed at build time to write a Java class containing the localized strings. Currently, the application supports one language at a time, chosen at build time.

To add a new language, follow these steps:

- Copy the default resource file (see table above) into a new folder named as the capitalized language code (e.g. FR for French)
- Translate the strings
- Set the name of the property file to be used in the *build/build.properties* file
- Rebuild the application

14.4.3. How to translate the help text

For the most popular devices, the help text is customized to match the device specifics. The generic help text is contained in the file *UI/basic/res/help.txt*, and is used for phones that are not in the table below.

To translate the help, it is necessary to translate all these files and rebuild the application.

Device	File name
BlackBerry	UI/basic/res/help/Blackberry/help.txt
LG	UI/basic/res/help/LG/help.txt
Motorola	UI/basic/res/help/Motorola/help.txt
Nokia	UI/basic/res/help/Nokia/help.txt
Samsung	UI/basic/res/help/Samsung/help.txt
SonyEricsson	UI/basic/res/help/SonyEricsson/help.txt

14.5. BlackBerry Sync Client

The BlackBerry Sync Client is written in Java using Java ME on top of the RIM proprietary API.

14.5.1. Languages currently available

Language	File name	Status
English	src/xml/language.xml src/help/help.txt	Default language, officially maintained in the product.
German	N/A	Done for a customer, can be adapted and committed to the code repository (not there yet).

14.5.2. How to add a new language

The localized strings are contained in two files: an XML file that represents the general language of the file and an help file distributed as *help.hlp* and generated by the file *src/help.txt*. Currently, the application supports one language and one help file at a time.

The language and help file can be changed translating the message in the *language.xml* and *help.txt* files.

To add a new language, keeping more than one in the build environment and choose one at build time, you can follow these steps:

- Copy the default resource file (see table above) into a new one; the following name scheme is suggested: *src/xml/language_<code>.xml* (e.g. *src/xml/language_fr.xml*);
- Translate the strings in the new file
- Set the name of the property file to be used in the file:
  src/java/com/funambol/util/StaticDataHelper.java
- Change the help file with the one translated into the new language, overwriting the file *src/help/help.txt* with the new one.
- Rebuild the application.

14.6. iPod Sync Client

The Funambol iPod Sync Client is written in Java standard edition. There are two sets of strings that can be localized:

- Main application strings (see section 14.6.1)
- Windows installer strings (see section 14.6.2)

14.6.1. Main application strings localization

The localization of the main application is managed automatically by the JavaSE *ResourceBundle* component.

All the strings are contained in property files found in the following resource folder:

- *src/main/resources/com/funambol/syncclient/util*

The default language is specified in the *language.properites* file (default: English).

Further languages can be added in this folder by specifying the locale ID in the file name (e.g. *language_de.properties* represents the German language). Please refer to the JavaME *java.util.ResourceBundle* class for more details.

The following is the list of the main application languages currently available:

Language	*File name*	*Status*
English	language.properties	Default language, officially maintained in the product.
German	language_de.properties	-

14.6.2. Windows installer strings localization

All the locale files are found in the following folder:

- *src/nsis/locale*

For each language, there is a specific folder (e.g. *en-us*) which contains a *nsh* property file that is included by the NSIS installer script.

Currently, only the English language is available.

14.6.3. How to add a new language

If you want to add a new language for the main application, follow these steps:

1. Copy the default language file (*language.xml*)
2. Translate all the strings in a separated file named *language_<locale>.xml* where *<locale>* is the new language locale ID (please refer to the JavaSE j*ava.util.Locale* class documentation to retrieve a proper locale ID).

If you want to add a new language for the Windows installer, follow these steps:

1. Create a new folder under *src/nsis/locale* named using the new language locale ID (e.g. *de-de* for the German language)
2. Copy an existing *nsh*, which you can find under an existing language folder
3. Translate all the contained strings
4. Open the *ipod.nsi* file (located in the directory *src/nsis*) and add the following line:

```
!insertmacro MUI_LANGUAGE "<language-name>"
```

   Where *language-name* is the name of the new language.

5. Then, include the new *nsh* property file:

!include ".\locale\<locale-id>\language.nsh"

14.7. iPhone/iPod Touch Sync Client

The iPhone Sync Client is written in ObjectiveC and can run on the iPhone and on the iPod Touch.

The localization framework is the one provided by Apple with their Xcode development environment, and the customization is based on the Xcode resources.

14.7.1. Languages currently available

Language	*File name*	*Status*
English	UI/I10n/English.lproj/Localizable.strings	Default language, officially maintained in the product

14.7.2. How to add a new language

The strings are contained in a text file called *UI/l10n/<Language>.lproj/Localizable.strings* (for example, *UI/English.lproj/Localizable.strings*).

To add a new language, follow these steps:

- Copy the folder *English.lproj* and its contents to another folder (e.g. *French.lproj*);
- Translate the strings in the file *Localizable.strings* under the new folder
- Rebuild the application

14.8. Symbian Sync Client

The Symbian Plugin is written in C++ and the customization and localization is based on the Symbian toolchain resources.

14.8.1. Strings localization

All localized strings are defined inside a *.rls* resource file, found under the data directory of the client source tree. For each language supported there is a different file that contains all the strings in that language.

During install procedure a list of the language options is presented, to choose the desired language to use. If one of the languages supported is exactly the same as the language of the OS in use, that language will be automatically selected without prompting the user.

To add a new language (currently, only English and German are supported), follow these steps:

- Open the *Funambol.mmp* file under the *group* directory. Locate the *LANG* statement and add a new language code; for example, change:

  ```
  LANG EN DE
  ```

  to:

  ```
  LANG EN DE FR
  ```

- Open the *customization.h* header file, under the *inc* directory. Add an *#if defined* macro to include the strings resource file for the new language supported; for example:

  ```
  #if defined (LANGUAGE_FR) #include "strings_fr.rls" #endif"
  ```

- Create the new strings resource *.rls* file under the *data* directory, copying it from an existing *.rls* file and translating all the desired definitions (e.g. create the file *strings_fr.rls*)
- Open the *Funambol_gcce.pkg* file under the *sis* directory. Locate the languages statement beginning with "*&*" and add the new supported language code; for example, change:

  ```
  &EN,GE
  ```

  to:

  ```
  &EN,GE,FR
  ```

  Note: the codes MUST be the predefined two-letter language codes that identify the language, for example "FR" for "French".

- Add a localized string for all the following statements:
  - package header (line starts with "#")
  - localized vendor name (line starts with "%")
- Add the new localized resource files to the list of resource files to be installed; for example, change:

```
{..."resource.ren" "...resource.rde"} -"!:\...resource.rsc"
```

to:

```
{..."resource.ren" "...resource.rde" "...resource.rfr"} -"!:\...resource.rsc"
```

Note: the list of files to be installed MUST be in the same order as the list of supported languages in the “&” line.

15. Funambol Software Development Kit

The Funambol Software Development Kit (SDK) [7] is the group of tools available to develop Funambol extensions.

To install it, extract the archive *funambol-sdk-<version>.tar.gz* in a directory of choice; the directory structure shown in Figure 19 will be created.

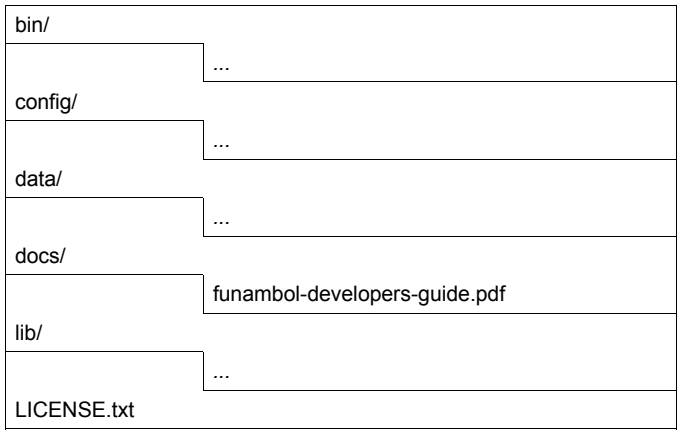


Figure 19: Funambol SDK directory structure

There are two ways to develop Funambol extensions: using your own custom development environment or using Maven [4]. Both methods will be explained in the following sections, but the preferred method (since Funambol v6.5) is using Maven.

In the following sections, it is assumed that the developer is familiar with the following concepts:

- Java development
- Funambol architecture
- Maven (optional)

15.1. Obtaining and building the source code

Funambol is open source! The best source of information and documentation is the source code, which is available on the Funambol public code repository [6].

The instructions on how to download and build Funambol version 8.0 can be found here:

- https://core.forge.funambol.org/wiki/BuildingFunambolV8

The instructions on how to download and build older versions of Funambol can be found here:

- Funambol version 7.1: https://core.forge.funambol.org/wiki/HOWTOBuildCapri
- Funambol version 7.0: https://core.forge.funambol.org/wiki/BuildingFunambolV7
- Funambol version 6.5: https://core.forge.funambol.org/wiki/BuildingFunambolV6.5
- Funambol version 6.0: https://core.forge.funambol.org/wiki/BuildingFunambolV6

15.2. Developing with a custom environment

This section describes how to develop a Funambol extension using just an editor and the Java Development Kit (JDK).

When developing in a custom environment there is no any additional requirement other than generating a s4j package as described earlier; how to do it is left to the developer. One important thing to note is that in order to compile your classes you have to make sure to have the following framework jars in the build classpath:

Name	*Download URL*
Funambol version 8.0	
core-framework-<version>.jar	http://m2.funambol.com/repositories/artifacts/funambol/core-framework/8.0.1/core-framework-8.0.1.jar
server-framework-<version>.jar	http://m2.funambol.com/repositories/artifacts/funambol/server-framework/8.0.0/server-framework-8.0.0.jar
ds-server-<version.>jar	http://m2.funambol.com/repositories/artifacts/funambol/ds-server/8.0.0/ds-server-8.0.0.jar
admin-framework-<version>.jar	http://m2.funambol.com/repositories/artifacts/funambol/admin-framework/8.0.0/admin-framework-8.0.0.jar
Funambol version 7.0	
core-framework-<version>.jar	http://m2.funambol.com/repositories/artifacts/funambol/core-framework/7.0.0/core-framework-7.0.0.jar
server-framework-<version>.jar	http://m2.funambol.com/repositories/artifacts/funambol/server-framework/7.0.3/server-framework-7.0.3.jar
ds-server-<version.>jar	http://m2.funambol.com/repositories/artifacts/funambol/ds-server/7.0.2/ds-server-7.0.2.jar
admin-framework-<version>.jar	http://m2.funambol.com/repositories/artifacts/funambol/admin-framework/6.5.2/admin-framework-6.5.2.jar

These libraries can be found under the directory lib of the Funambol SDK.

For an example on how to develop a connector, see chapter 6, “Error: Reference source not found“.

15.3. Developing with Maven

Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information. For more information about Maven, see [4]. In the next section it is assumed that the reader is already familiar with Maven and its concepts.

Even if many Funambol version 6.5 components have been put under the Maven build system already, only with Funambol version 7 and later Maven has become the default build tool for Funambol components. All Funambol artifacts are now represented, built and deployed as Maven artifacts. Funambol has also set up a public maven repository where all components are published. Funambol maintains two repositories: *artifacts*, for released artifacts and *snapshots*, for snapshots; the latter keeps the snapshots for the last 10 days. The repository can be accessed through the URL http://m2.funambol.org/repositories

15.3.1. Maven configuration

Before using Maven, the repository above must be added to your maven environment. Do to so, edit your *settings.xml* file (either under maven or your user home) and add the following sections:

```
<repositories>
  <repository>
    <id>artifacts</id>
    <url>http://m2.funambol.org/repositories/artifacts</url>
  </repository>
  <repository>
    <id>snapshots</id>
    <url>http://m2.funambol.org/repositories/snapshots</url>
```

```
  </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>artifacts</id>
    <url>http://m2.funambol.org/repositories/artifacts</url>
  </pluginRepository>
  <pluginRepository>
    <id>snapshots</id>
    <url>http://m2.funambol.org/repositories/snapshots</url>
  </pluginRepository>
</pluginRepositories>
```

15.3.2. Creating a new module

The best and quickest way to create a Funambol module with Maven is using the funambol-module-archetype:

```
mvn archetype:create -DarchetypeGroupId=funambol
-DarchetypeArtifactId=funambol-module-archetype -DarchetypeVersion=6.5.0
-DgroupId=<your group> -DartifactId=<your artifact>
-DarchetypeRepository=http://m2.funambol.org/repositories/artifacts
-Dversion=<version>
```

For example:

```
mvn archetype:create -DarchetypeGroupId=funambol
-DarchetypeArtifactId=funambol-module-archetype -DarchetypeVersion=6.5.0
-DgroupId=yourgroup -DartifactId=yourconnector
-DarchetypeRepository=http://m2.funambol.org/repositories/artifacts
-Dversion=1.0.0
```

This creates a new Maven project for *yourgroup:yourconnector* in a directory called *myconnector*. This projects contain a skeleton of a SyncSource and of a input and output Synclet.

Note: the command line above creates a Funambol version 6.5 module specifying all the required dependencies. You don't need to worry about which jars you have to download and add to the classpath, as Maven will do all this for you. To create a Funambol version 8.0 module, just use the version 8.0.0 of funambol-maven-archetype:

```
mvn archetype:create -DarchetypeGroupId=funambol
-DarchetypeArtifactId=funambol-module-archetype -DarchetypeVersion=8.0.0
-DgroupId=yourgroup -DartifactId=yourconnector
-DarchetypeRepository=http://m2.funambol.org/repositories/artifacts
-Dversion=1.0.0
```

15.3.3. Building the module

To build the module created as described below, just go inside the newly created directory and run maven. For example:

```
mvn package
```


15.4. The Funambol Connector Testing Framework

The Funambol Connector Testing Framework (FCTF) is a command-line tool that can be used to test and validate any Funambol connector against any set of PIM or e-mail items, without setting up a complete test environment comprising a device, a Data Synchronization server and a back-end system.

Note: you need a running back-end system and the connector for an FCTF instance to work.

Each FCTF instance is defined by its own set of PIM and/or e-mail items and corresponding expected results, in addition to the connector to be used.

Three operation modes are available:

1. Automatic mode (default): only pre-defined sets of tests depending on the MIME type can be performed.
2. Single-test mode: only one test is specified using the *--command* keyword.
3. Batch mode: a batch file containing a series of tests is specified using the *--batch* keyword.

Automatic tests allow for different levels of certification of a connector. A connector that passes the basic automatic tests, for example, can be declared to be FCTF-certified at the basic level. The certification level can be specified on the command line after the *--testset* keyword, the default value being basic.

The batch files (and, therefore, also the items used for the individual tests) may or may not be connector-specific, since the expected results can change depending on the features of the back-end system.

15.4.1. Usage

The Funambol Connector Testing Framework has been designed to be used from the command line; the executables can be found under the *bin* directory of the work environment.

Once the work environment has been generated, the user is expected to change their working directory to *$FUNAMBOL_HOME/tools/sdk* and run the command *bin/fctf* from there.

An example of a typical command line that will launch the automatic tests at the basic certification level for a vCard sync source belonging to the Foundation standard connector is:

```
bin/fctf -s config/foundation-vcard.xml
```

bin/fctf requires at least one argument (the SyncSource); the following tables details all available options:

Argument	*Short form*	*Parameter(s)*	*Default value*	*Usage notes*
--source	-s	The path to the XML file defining the SyncSource; it usually begins with *config*.	N/A	Mandatory. It is needed to load the SyncSource that will be operated by the tool.
--testset	-t	The certification level (it must correspond to an existing subdirectory of data).	basic	Default option. If this option is selected, the FCTF will operate in the automatic-tests mode.
--command	-c	A single command, according to the grammar defined in section 15.4.1, Tests.	N/A	If this option is selected, the FCTF will operate in singe-test mode.
--batch	-b	The path to the batch command file containing the tests to be performed (one per line).	N/A	If this option is selected, the FCTF will operate in batch mode.
--datasource	-D	A list of datasource names (usually, 2: the user DB and the core DB). The first part of each name must correspond to an existing subdirectory of *config/com/funambol/server/db*, the second part to an existing *.xml* file within that subdirectory.	jdbc/fnblcore jdbc/fnbluser	

Argument	*Short form*	*Parameter(s)*	*Default value*	*Usage notes*
--authtype	-a	The type of authentication (*basic*, *md5* or *MAC*).	basic	This is used for connector whose officer uses a special authentication type.
--path	-P	The path to the data item.	data	This is used to load different sets of items without modifying the built-in work environment.
--user	-u	The username.	fctf	
--password	-p	The user's password.	the username	This is useful only when the user needs be authenticated.
--device	-d	The device ID	fctf	
--officer	-o	The path to the XML file defining the officer; it usually starts with *config*.	config/test-officer.xml	This depends on the connector to be tested.
--defaultuser	-U	The path to the XML file defining a pre-authenticated user; it usually starts with *config*.	N/A	This is useful when the officer is not yet developed or does not work properly within the FCTF context. In case the authentication process fails (i.e., returns a null user), a default user (belonging to a subclass of Sync4jUser that depends on the connector to be tested) will be unmarshalled and used instead.
--verbose	-v	No parameter.	N/A	If selected, in case of error the stack trace of the exception raised will be displayed.

Tests

The FCTF performs a pre-defined series of tests that are a combination of the following basic tests:

- removal of an item from the back-end system (*d*);
- update of an item on the back-end system (*u*);
- retrieval of an item from the back-end system (*g*);
- retrieval of an item's twin from the back-end system (*t*).

All of these tests are preceded by the addition of an item on the back-end system, and followed, if necessary, by the removal of the items. Therefore every test is independent and leaves the back-end server in a clean state, if no concurrent operations are performed in the meanwhile on the same item.

In the list above, the letter in brackets is the one used to call for an individual test. It will be followed by the item(s) file name(s). The letter *x* is also used to indicate the expected result, if applicable. The same grammar is used both in the command-line single-test mode and the batch mode:

- <*SyncSource*> *d* <*filename*> - the item is added in the back-end system and then removed;
- <*SyncSource*> *u* <*filename1*> <*filename2*> *x* <*filename3*> - the first item is added in the back-end system, it's updated with the modified content of *filename2* and then compared to the expected result in *filename3*;
- <*SyncSource*> *g* <*filename1*> *x* <*filename2*> - the first item is added in the back-end system, then retrieved and compared to the expected result in *filename2*;
- <*SyncSource*> *t* <*filename1*> <*filename2*> ... <*filenameN*> *x* <*K*> - all items but the first one are added in the back-end system, then the twins of the first one are looked for; *K* (any number from 0 to *N*) twins are expected to be found.

If one such command is called in the single-test mode, it must be specified after the *--test* keyword. If a a series of commands is launched in the batch mode, they will be listed, one per line, in the batch file.

In the automatic mode, the tests do not need to be specified using this grammar because the test set is pre-defined. For example, if the items to be tested are *item1.txt*, *item2.txt* and *item3.txt*, the following tests will be automatically generated and launched:

- g item1.txt x item1.txt
- *g item2.txt x item2.txt*
- *g item3.txt x item3.txt*
- *u item1.txt item2.txt x item2.txt*
- *u item2.txt item3.txt x item3.txt*

This will be done for every MIME type supported by the SyncSource being tested, starting from the preferred MIME type.

Test items

The items are provided as vCalendar, iCalendar, vCard, SIF-E, SIF-T, SIF-C and RFC882 data. They are in the *data* subdirectory. The MIME type of each item set is inferred from the file extension:

- *.vcs* for vCalendar (1.0)
- *.ics* for iCalendar (vCalendar 2.0)
- *.vcf* for vCard
- *.sife*, *.sift*, *.sifc* for the SIF format
- *.eml* for RFC882 data

It's important to notice that these data are not filtered by the synclets, therefore this tester cannot be used to test the behavior of different devices.

Items used in the automatic tests are grouped in different subdirectories of directory *data* according to the certification level they belong to. Each MIME type within a certification level has its own directory where items of that type will be collected. The directory subtree will reproduce the structure of MIME taxonomy. For example, vCalendar (1.0) items belonging to the Basic certification level will be found under *data/basic/text/x-vcalendar*. The file names are relevant because the items are sorted in alphabetical order when the automatic tests are generated and launched.

Test pass conditions

During automatic tests, server-generated data (in vCalendar, iCalendar, vCard, SIF-E, SIF-T, SIF-C or RFC882 format) are extracted by the back-end response message and compared with the corresponding item in the pre-defined set of the expected results. If the comparison does not outline any data loss, the tests have succeeded.

It is important to underline that the FCTF only checks for the possible loss of data between the expected result and the actual result. If the actual result contains *more* information than the expected one, that is considered as a successful comparison.

The comparison is done on a line-wise basis for vCard, vCalendar and iCalendar items, while for SIF items it is based on (non-empty) end nodes (leaves) of the XML node tree.

In this way, the SyncML message is not compared as such to an expected result, as in the more generic Funambol Test Suite. The FCTF focuses only on the PIM and e-mail data.

The same mechanism is used in the single-test and batch modes for retrieval and update tests. In the removal and twin-search cases, the expected results are just the actual deletion of the item and a positive result of the twin search. If these outcomes occur, the tests have succeeded.

15.4.2. Certifying a connector

The certification of a connector may require a few steps.

Installing the libraries

Note: This step is always necessary.

The libraries needed for the connector's SyncSources to run (for example, the connector's module) must be copied as JARs in *lib/ext*. This includes the libraries for the access to data sources, like the DB.

Installing the SyncSources

Note: This step is always necessary.

The *.xml* files containing the marshalled version of SyncSources must be copied in the *config* subdirectory. These files are usually exactly the same *.xml* files that can be found under the *config* directory in the connector's source, already prepared for usage by the Funambol Administration Tool.

Installing the officer

Note: This step is often, but not always, necessary.

Some connectors require an officer for user authentication. In this case, its marshalled version must also be copied in the *config* subdirectory. If no user authentication is needed on the back-end system, the default dummy officer (*TestOfficer*) can be used.

If the officer is not yet developed or does not work within the FCTF context, a default user can be defined. This will be unmarshalled and used when the authentication fails. In order to activate this behavior, a marshalled pre-authenticated user (i.e. the expected return value of the *authenticate* method of the connector's officer) must be created in the *config* subdirectory. This object must represent an authorized user in the back-end system.

Setting up the data source

Note: This step is often not necessary.

In the *config/com/funambol/server/db* directory and its subdirectory *jdbc* there are *.xml* files containing the marshalled version of the automatic configurer of the JDBC connections to the datasources. New files can be added or the default ones can be modified to replicate the features of the data source in use.

Setting up the data for individual (or batch) tests

Note: This step is not strictly necessary for certification, only for other tests.

Item files can be added in the *data* subdirectory. It's important to follow the file extension rules as of Test items, section 15.4.1.

Setting up the data for a custom certification

Note: this step is not necessary for pre-defined certification levels, only for custom certifications.

Custom certifications can be added under the *data* directory. In order to do that, a subdirectory must be created with the name of the custom certification level. Under that subdirectory, it is necessary to create a directory tree that follows the MIME types tree, including all MIME types that are wished to be included in the custom certification. The *data/basic* subdirectory can be used as an example. Then, item files must be created in the correct directories. The file names are relevant because the automatic tests will be generated and launched on the basis of the alphabetical order.

Launching the tests

See section 15.4.1, Usage.

15.4.3. Limitations

A key purpose of the FCTF is to test the mapping between the foundation data model and the back-end data model. This cannot be tested directly if the tool has to be an automated one because it cannot perform any direct test on the status of the back-end system. The back-end status can only be inferred by the data retrieved by the connector, but those data, in turn, are inserted there by the insert method of the connector.

If a property *A* on a client item is *incorrectly* mapped to property *b* on the server and this is "correctly" mapped back to the client as *B*, the difference between *A* and *B* may be used to detect an error in the behavior of the connector. On the contrary, if a property *A* on a client item is *correctly* mapped to property *a* on the server and this is correctly mapped back to the client as *A*, the item will pass the test. But what if a property *A* on a client item is *incorrectly* mapped to property *b* on the server and this is again *incorrectly* mapped back to the client as *A*? There is no way to tell this case from the previous one.

Original value	*Value on the server after synchronization with the client*	*Value on the client after retrieval from the server*	*Test outcome*	*Bugs?*
A	b	B	Failed	Yes
A	a	A	Passed	No
A	b	A	Passed	Yes

As a consequence, there's a class of bugs that cannot be verified with this tool, like field swaps, some time zone errors etc. Manual tests with human interaction with the back-end system are still needed to investigate these cases.

15.4.4. Error codes

Please note that a connector cannot be considered certified as long as it has not passed the automatic tests for the required certification level with the application successfully exiting without displaying any fatal warning or error.

When the application exits with an error, a brief error message is usually displayed and an error code is provided. The most likely solution to the problem corresponding to each error code is explained in this table:

Code	*Mode*	*Most likely solution*
2	any	Check that the SyncSource specified with the *-s* argument is valid and can be unmarshalled. The required libraries must be under *lib/ext*.
3	any	Check that the officer specified with the *-o* argument is valid and can be unmarshalled. The required libraries must be under *lib/ext*.
4	any	Check that the datasource name specified with the *-D* argument is correct.
5	any	Check that the datasource is available, its parameters correct and the required libraries under *lib/ext*.
6	any	Check that the datasource is up and running properly.
7	any	Check that the *.xml* files used to configure and bind the datasources are correct and in the correct locations.
99	single test	Check the grammar of the test command. It must match one of the cases listed in section 15.4.1, Tests.
102	single test	Check the mapping used by the SyncSource and debug the connector. Be sure that the expected results were correct.
103	single test	Check the mapping and twin search criterion used by the SyncSource and debug the connector. Be sure that the expected twin count were correct.
104	single test	Check that the back-end system is up and running. Debug the addition (create) function of the SyncSource.
105	single test	Debug the deletion (remove) function of the SyncSource.
106	single test	Debug the retrieval (get) function of the SyncSource.
107	single test	Debug the update (modify) function of the SyncSource.
108	single test	Debug the twin search (get twins) function of the SyncSource.
110	single test	Debug the SyncSource.
111	single test	Check that all item files mentioned in the test command exist under the default item directory (*data*) or the directory specified with the *-P* argument.
112	single test	Check that the extensions of all item files mentioned in the test command are among the supported extensions listed in section 15.4.1, Test items.

Code	*Mode*	*Most likely solution*
113	single test	Check that the twin count command ends with an integer.
202	batch	Check the mapping used by the SyncSource and debug the connector. Be sure that the expected results were correct.
203	batch	Check the mapping and twin search criterion used by the SyncSource and debug the connector. Be sure that the expected twin count were correct.
204	batch	Check that the back-end system is up and running. Debug the addition (create) function of the SyncSource.
205	batch	Debug the deletion (remove) function of the SyncSource.
206	batch	Debug the retrieval (get) function of the SyncSource.
207	batch	Debug the update (modify) function of the SyncSource.
208	batch	Debug the twin search (get twins) function of the SyncSource.
210	batch	Debug the SyncSource.
211	batch	Check that all item files mentioned in the test command exist under the default item directory (*data*) or the directory specified with the *-P* argument.
212	batch	Check that the extensions of all item files mentioned in the test command are among the supported extensions listed in section 15.4.1, Test items.
213	batch	Check that the twin count command ends with an integer.
220	batch	Check that the batch file specified with the *-b* option exists.
302	automatic tests	Check the mapping used by the SyncSource during the addition of a new item and debug the connector.
304	automatic tests	Check that the back-end system is up and running. Debug the addition (create) function of the SyncSource.
305	automatic tests	Debug the deletion (remove) function of the SyncSource.
306	automatic tests	Debug the retrieval (get) function of the SyncSource.
310	automatic tests	Debug the SyncSource.
311	automatic tests	Do not modify the automatic test files while the application is running.
312	automatic tests	Check that the extensions of all item files in the automatic tests directories are among the supported extensions listed in section 15.4.1, Test items.
320	automatic tests	Check that the directory specified with the *-a* option exists *under the default item directory (data*) or the directory specified with the *-P* argument.
352	automatic tests	Check the mapping used by the SyncSource during the update of an item and debug the connector.
354	automatic tests	Debug the addition (create) function of the SyncSource.
355	automatic tests	Debug the deletion (remove) function of the SyncSource.
356	automatic tests	Debug the retrieval (get) function of the SyncSource.
357	automatic tests	Debug the update (modify) function of the SyncSource.
360	automatic tests	Debug the SyncSource.
361	automatic tests	Do not modify the automatic test files while the application is running.

16. The Funambol Device Simulator tool

This section describes the Funambol Device Simulator, a tool that allows to test the server simulating SyncML devices; it is addressed to developers who want to run one or more existing test suites against a bundle to test their own software, or who would like to add a brand new test.

With this tool, it is possible to run one of the test suites provided, for example to test a new module you are working on, and it is also possible to add new tests.

At present, the following test suites are available:

- *phone-testsuite* (used to simulate the sync of specific devices)
- *bug-testsuite* (used to simulate bug behavior)
- *email-testsuite* (used to test the email connector)
- *general-testsuite* (used to test the general behavior of the DS Service, for instance the twin search or if the DS Service is able to handle the device capabilities)

Using this tool, you can ensure that what you are doing does not break the protocol and synclets compatibility.

Each test is made of a set of messages and some SQL scripts. The logic of this tool is based on ant scripts and customized tasks that execute simple steps, such as:

- sending messages to the server
- matching response messages from the server with the expected ones
- executing SQL scripts, for example to clean up the database before or after running the tests, or to change item statuses.

It is also possible to configure the Device Simulator tool to run a whole set of tests without having to sync with each of them manually, which could be very time-consuming.

Most of the tests that you will find in the Device Simulator follow the same schema, although you are free to customize each test:

1. in the first slow sync, items are sent from the client to the server
2. an update is executed on the database to change the status of the items
3. in the second fast sync, items are sent back from the server in order to verify if the server correctly saved the items sent by the client during the previous sync, and also if the correct output synclet was called.

A test is passed if the significant part of the messages received from the server matches the expected messages saved on the file system.

Using this tool, you can run both *xml* and *wbxml* based tests; keep in mind that they are launched with two different commands and you will need to run both of them in order to pass the whole test suite.

The reason why two different commands are required is that a test could be both *xml* and *wbxml* based, and each time you run a kind of test the test directory structure is changed and server messages are stored. So if you launch two tests, you will only find info about the last executed test.

16.1. Prerequisites

In order to use the Funambol Device Simulator, the Funambol Server software must be installed on the same host where the Funambol SDK has been unpacked.

The device to be tested must be SyncML 1.0, 1.1 or 1.2 compliant.

The scripts used to launch the tests require *ant* version 1.6.5.

Note: the Device Simulator tool works with both Funambol Community Edition version and Funambol Carrier Edition version.

16.2. Directory structure

The Funambol Device Simulator is delivered with the Funambol SDK. You will find this tool in the directory *Funambol/tools/sdk*, according to the structure showed in Table 7:

Directory	*Description*
/bin	This directory contains executable files
/config	This directory contains configuration files
/lib	This directory contains libraries (i.e. jar files)
/report	This directory contains report files (i.e. files containing info about failed/passed tests)
/data	This directory contains testcases and other files
/docs	This directory contains the documentation

Table 7: Device Simulator directory structure

16.3. Adding new tests

In this section we will describe the steps needed to add a new test.

The process to record a new test is almost the same for all SyncSources and it will be described for a generic item; any points that might change when using a different SyncSource will be highlighted.

16.3.1. Setting up the environment

In order to record a new test, you will need a Funambol Server and the device you wish to test, which must be able to reach the server.

First of all, you must set the debug level to TRUE and the server log level to ALL.

To change the debug level, edit the file *FUNAMBOL_HOME/bin/funambol-server* file and modify the following line:

```
set JAVA_OPTS=%JAVA_OPTS% -D funambol.debug=true
```

This allows to make the content of SyncML objects viewable, as it is hidden by default for privacy purposes.

The server log level can be modified using the Funambol Administration Tool: go to *Server Settings* | *Logging* | *Loggers* | *funambol* and modify the Logging level to ALL.

16.3.2. Testing the device

In order to write a test case for a device, first of all it is necessary to check that the device is synchronizing correctly against the Funambol Server and there are no blocking issues on this side.

In this step, you should run an initial sync with all the SyncSources that the device can synchronize (contacts, calendar, tasks, notes and emails), in order to create a synchronization profile on your device.

Once you have checked that the device can synchronize with the Funambol Server, the automated test compilation can start.

16.4. Adding items to sync

During the first phase of the test, you will be sending items to the server, so will need to create some new items on the device. In the next paragraphs, the guidelines to follow in order to add items to the device's database will be detailed.

According to the type of item you wish to test, you will need to follow the instructions provided in one of the following paragraphs: 16.4.1 for Contacts, 16.4.2 for Events, 16.4.3 for Tasks, 16.4.4 for Notes, 16.4.5 for Emails). Then you can jump directly to paragraph 16.4.6 to finish recording the test.

16.4.1. Adding contacts to the address book

In order to test the Contact SyncSource, you will need to manually add some contacts on the device's address book. Note that the existing test cases use 5 contacts.

All available fields should be filled for each contact, including:

- first name and last name
- all available types of phone numbers: mobile, home, work, fax, etc.
- email addresses: home, work, other
- web address
- picture (if available)
- contact informations: home and work address, title, company, etc.
- birthday
- notes

In order to make the test more effective, contacts should contain special characters such as:

- symbols, for example: &, @, $, %
- language specific characters, for example: â, ã, ä, å, è, é

16.4.2. Adding events to the calendar

In order to test the Calendar SyncSource, you will need to manually add some events on the device's calendar. Note that existing test cases use 6/8 events.

All available fields should be filled for each event, including:

- subject
- location
- description
- start/end time and date
- duration

The test should take into account all event types that the device supports, including recurring events (daily, weekly, monthly and yearly) and recurring events exceptions; for this reason, all existing event types should be added in the test case.

In order to make the test more effective, events should contain special characters such as:

- symbols, for example: &, @, $, %
- language specific characters, for example: â, ã, ä, å, è, é

Note: some devices, such as Nokia and some Samsung and LG devices, can synchronize Calendar and Todos (i.e. Tasks) at the same time. In this case, refer to section 16.4.3 Adding tasks to include these items in the synchronization.

16.4.3. Adding tasks

In order to test the Task SyncSource, you will need to manually add some tasks on the device. Note that existing test cases use 2 tasks.

All available fields should be filled for each task, including:

- subject
- reminder
- priority
- start/end time and date (if available)
- duration (if available)

The test should take into account all task types that the device supports, including recurring tasks (daily, weekly, monthly and yearly) and recurring tasks exceptions; for this reason, all existing task types should be added in the test case.

In order to make the test more effective, tasks should contain special characters such as:

- symbols, for example: &, @, $, %
- language specific characters, for example: â, ã, ä, å, è, é

Note: some devices, such as Nokia and some Samsung and LG devices, cannot synchronize Todos (i.e. Tasks) separately from the rest of the calendar. In this case, refer to section 16.4.2 Adding events to the calendar to include these items in the synchronization.

16.4.4. Adding notes

In order to test the Note SyncSource, you will need to manually add some notes on the device. Note that existing test cases use 2 notes.

All available fields should be filled for each task, including:

- body
- subject (if available)
- date (if available)

In order to make the test more effective, notes should contain special characters such as:

- symbols, for example: &, @, $, %
- language specific characters, for example: â, ã, ä, å, è, é

16.4.5. Adding emails

In order to test the Email SyncSource, you have to send an email to the account that you want to sync; in this way, you'll have a new email on your mail server to sync to the client.

The email can contain text or HTML and can also have attachments. This is useful when performing the test using a real device and a real mail server, in order to analyze the log.

Remember that to test the Email SyncSource with the Device Simulator, you will need to change the backend using a mock JavaMail provider provided by Funambol instead of a real mail server. This JavaMail provider is a fake mail server to be used in testing for more flexibility and to be able to check on the status of emails since they are stored on the file system. For more information on the JavaMail API, see [12].

Copy *lib/mock-javamail-provider-*.jar* under the directory *$FUNAMBOL_HOME/tools/tomcat/webapps/funambol/WEB-INF/lib* before running the Funambol server, and *mock-javamail-provider.properties* under *$FUNAMBOL_HOME/config* (this properties file is used to configure the mock javamail provider).

You can find examples of mail servers to use in the test (e.g. gmail.com) under *Funambol/tools/sdk/config/email-testsuite/mailserver*, that contains the configuration for the accounts *user.devsim1* and *user.devsim2*.

The tests provided simulate:

- Synchronization using the IMAP protocol. These tests use a fake Gmail mail server (*gmail.com*) with accounts *user.devsim*1 and *user.devsim2.*
- Synchronization using the POP3 protocol. These tests use a fake server named *popuser.com* with user *user.devsim3.*

For each Gmail account there are the *inbox*, *outbox* and *sent* folders. For the *popserver.com* account only the *inbox* folder is provided, since current tests only use this server with the POP3 protocol and the POP3 protocol only supports the *inbox* folder.

Note: each fake server can be accessed with both IMAP and POP3 protocols.

The name of the filesystem folders that represent a mail server must be lower case. Therefore, a folder named *inbox* stands for the mail server's *inbox* folder, while *Inbox* or *INBOX* are not recognized by the mock JavaMail provider.

In order to simulate a new email on the mail server, you must add a *.eml* file to the *inbox* folder. The *.eml* file should contain both the body of the email and the XML code to specify the flags, since they can be synchronized as well.

Note: the Inbox Listener Service does not need to be running: to simulate its work, simply insert a record in the *fnbl_email_inbox* table. This is done to simplify the coordination of the test with the Inbox Listener Service loop.

16.4.6. Getting messages

Setting up the synchronization profile

Before starting the sync process, you will need to configure your profile on the device with the following details:

- server address, that is your server's IP address or URL (for example: *http://127.0.0.1/funambol/ds*)
- username; for automated tests, the default to be used is *guest*
- password; for automated tests, the default to be used is *guest*
- items to be synchronized (should be contacts only)
- remote contacts database, that is, *card*

Note: only the desired SyncSource will be synchronized in this step.

Starting the synchronization

Once the synchronization profile has been set up, the synchronization can start. In this phase, all items present on the device will be sent to the server.

16.4.7. Phase 1: extracting SyncML messages

At the end of the synchronization, you will need to extract the logs from the *FUNAMBOL_HOME/logs/ds-server/ds-server.log* file. In particular, it is necessary to isolate the SyncML messages that are displayed before the SYNCLET processing phase.

The SyncML message is the XML content that can be easily located after the *"Message to translate into the SyncML object"* trace, as shown in the following example:

```
[][funambol.server] [TRACE] [] [] [] [] Message to translate into the SyncML object:

<SyncML><SyncHdr><VerDTD>1.1</VerDTD><VerProto>SyncML/1.1</VerProto><SessionID>11</Ses
sionID><MsgID>1</MsgID><Target><LocURI>http://localhost:8080/funambol/ds</LocURI></Tar
get><Source><LocURI>syncml-phone</LocURI></Source><Cred><Meta><Type>syncml:auth-
basic</Type></Meta><Data>Z3Vlc3Q6Z3Vlc3Q=</Data></Cred><Meta><MaxMsgSize>10000</MaxMsg
Size></Meta></SyncHdr><SyncBody><Alert><CmdID>1</CmdID><Data>200</Data><Item><Target><
LocURI>./card</LocURI></Target><Source><LocURI>./C\System\Data\Contacts.cdb</LocURI></
```



```
Source><Meta><Anchor><Last>1</Last><Next>10</Next></Anchor></Meta></Item></Alert><Fina
l></Final></SyncBody></SyncML>
```

Each SyncML message sent from the device must be copied in a single text file which should be named *msgX.xml,* where *X* is the sequence number of the message. The set of *.xml* messages must then be copied in a directory named as the device model concatenated with the SyncSource name (for example: *N70_CARD*; see section 16.6 for the directory naming convention).

Each SyncML message sent from the server must be copied in a single text file which should be named *msgX.xml*, where *X* is the sequence number of the message. The set of *.xml* messages must then be copied in a directory named *reference* under the directory with the device model name concatenated with the SyncSource name (for example: *N70_CARD/reference*; see section 16.6 for the directory naming convention).

Creating the *header.properties* file

In the directory named as the device model, you will need to create the *header.properties* text file, which will be used by the device simulator to simulate the HTTP headers sent by the device.

The HTTP headers of the device can be found in the *ds-server.log* file and is usually composed as following:

```
> Cache-Control: no-store
> host: my.funambol.com
> accept: application/vnd.syncml+wbxml
> accept-charset: utf-8
> accept-language: en
> user-agent: Nokia SyncML HTTP Client
> content-length: 278
> content-type: application/vnd.syncml+wbxml
```

The *header.properties* file must then be filled with the following information (if present):

- *user-agent*
- *x-wap-profile*

For example, the *header.properties* file for Nokia N70 contains the following:

```
user-agent: Nokia SyncML HTTP Client
```

16.4.8. Syncing items back to the device and getting the logs

The next step of the test consists in syncing back all the items to the device, forcing an update from the server's side.

Setting up the server

First of all, the server needs to be set up to send back all items to the device. To do this, you need to perform an update on all the relevant server items.

From the command line of the database console, execute one of the following scripts according to the type of items you are working on.

Supposing you are working with contacts, the update query is:

```
update fnbl_pim_contact
set status = 'U',
last_update = (select end_sync from fnbl_last_sync
                      where principal =
                      (select MIN(id) from fnbl_principal where username='guest')
```



```
                    and sync_source = 'card'
                    );
```

Supposing you are working with calendar events, the update query is:

```
update fnbl_pim_calendar
set status = 'U',
last_update = (select end_sync from fnbl_last_sync
               where principal =
               (select MIN(id) from fnbl_principal where username='guest')
               and sync_source='cal'
               );
```

Note: for devices that have separate tasks synchronization (such as the Sony Ericsson) and use the Event SyncSource, the script to be used is the following:

```
update fnbl_pim_calendar
set status = 'U',
last_update = (select end_sync from fnbl_last_sync
               where principal =
               (select MIN(id) from fnbl_principal where username='guest')
               and sync_source='event'
               );
```

Supposing you are working with tasks, the update query is:

```
update fnbl_pim_calendar
set status = 'U',
last_update = (select end_sync from fnbl_last_sync
               where principal =
               (select MIN(id) from fnbl_principal where username='guest')
               and sync_source='task'
               );
```

Supposing you are working with notes, the update query is:

```
update fnbl_pim_note
set status = 'U',
last_update = (select end_sync from fnbl_last_sync
               where principal =
               (select MIN(id) from fnbl_principal where username='guest')
               and sync_source='note'
               );
```

This query will update the status of all items on the server and will force it to send them back to the device.

Supposing you are working with emails, to simulate a new email on your mail server you must insert a record in the *fnbl_email_inbox* table in this way:

```
insert into fnbl_email_inbox(guid, username, protocol, last_crc, invalid, internal,
messageid, headerdate, received, subject, sender, token, status)
values('I/3455-FUN-0',     'guest',     'imap',     767151784,          null,     null,
'<29343256.131242131595031.JavaMail.guest@local>',                  '20090518T123329Z',
```



```
'20090518T123329Z',                                    '',                                    '',
'2DlQn5g9F/Jifc7ZDjuKAhwIBSeJF8KxAuwQOk+PTE+eHqgVBVuxzg==', 'N');
```

This query will force the server to send them back to the device.

Starting the synchronization

The synchronization profile should be already set up by the initial sync, so you can start the new synchronization. In this phase, all the items stored on the server will be sent back to the device that will replace them. This will be also the second phase of the test.

16.4.9. Phase 2: extracting the SyncML messages

At the end of the synchronization, you will need to extract the logs from the *FUNAMBOL_HOME/logs/ds-server/ds-server.log* file. In particular, it is necessary to isolate the SyncML messages that are displayed before the SYNCLET processing phase.

Each SyncML message sent from the device must be copied in a single text file which should be named *msgY.xml*, where *Y* is the sequence number of the message. The set of *.xml* messages must then be copied in a directory named as the device model concatenated with the SyncSource name (for example: *N70_CARD*; see section 16.6 for the directory naming convention).

Each SyncML message sent from the server has to be copied in a single text file which should be named *msgY.xml*, where *Y* is the sequence number of the message. The set of *.xml* messages must then be copied in a directory named *reference* under the directory with the device model name concatenated with the SyncSource name (for example: *N70_CARD/reference*; see section 16.6 for the directory naming convention).

Note: the *Y* number of the sequence must start where the sequence reported in section 16.4.2 ended; for example, if the name of the last message of Phase 1 was *msg4.xml*, the first message of Phase 2 will be *msg5.xml*.

16.4.10. Editing SyncML messages

The set of SyncML messages collected during the previous stages of the test need to be edited in order to fit the Funambol Device Simulator standards; specifically, it is necessary to remove information related to the server and the client that were used to write them.

Removing DeviceID and server information

In the *.xml* files collected during the initial stages of the test, you need to replace the DeviceID information with a generic *syncml-phone* tag and the server information with a generic *http://localhost:8080/funambol/ds* tag.

SessionID

Each synchronization session is characterized by a unique SessionID. The SessionID used in the second fast sync should be consecutive (incremental) to the first one used in the slow sync and should be different for each test case; in this way, if a test fails, it will be possible to try to understand the error by identifying the log messages searching the specific SessionID.

Managing the SyncML content

This section presents an example of SyncML message sequence and what they should contain in order to be compliant with the Funambol Device Simulator standards:

Device	***Server***
msg1.xml Synchronization starts • credentials • put (device capabilities)	

• alert for synchronizing items (slow sync- 201)	
	msg1.xml • status (200) for device capabilities • status (200) for synchronizing items • alert for synchronizing items (slow sync- 201)
msg2.xml • status (200) for synchronizing items • sync • add command for each item (the <Data> tag contains the item encoded according to its format)	
	msg2.xml • status (200) for synchronizing items • status (201) for each added item
msg3.xml • alert (222)	
	msg3.xml • status (200) on alert • sync (no items are added/replaced/deleted by the server)
msg4.xml • status (200) for server sync request	
	msg4.xml Synchronization ends

Table 8: Example of SyncML message sequence – Phase 1

The sequence exemplified in Table 8 corresponds to Phase 1 of the test, where the device requests a slow sync and sends all its items to the server.

Any item contained in the *<Data>* tag of *msg2.xml* must be codified in one of the following formats:

- a Contact in *vcard* format
- a Note in *text/plain* format
- a Task in *vcal* format
- an Event in *vcal* format
- an Email in *application/vnd.omads-email+xml* format

Phase 2 is similar and is exemplified in the following table:

Device	***Server***
msg5.xml Synchronization starts • credentials • alert for synchronizing items (fast sync- 200)	
	msg5.xml • status (200) for synchronizing items • alert for synchronizing items (fast sync- 200)
msg6.xml • status (200) for synchronizing items	

• sync (no items are added/replaced/deleted by the device)	
	msg6.xml • status (200) for synchronizing items
msg7.xml • alert (222)	
	msg7.xml • status (200) on alert • replace command for each item (the <Data> tag contains the item encoded according to its format)
msg8.xml • status (200) for server sync request • status (201) for each replaced item	
	msg8.xml Synchronization ends

Table 9: Example of SyncML message sequence – Phase 2

Note: the number of messages could be higher depending on the size of the items and on the *MaxMsgSize* of the device. The sequence shown in Table 9 is just an example and it may not reflect what really happens in a real-world scenario.

Any item contained in the *<Data>* tag must be codified in one of the following formats:

- a Contact in *vcard* format
- a Note in *text/plain* format
- a Task in *vcal* format
- an Event in *vcal* format
- an Email in *application/vnd.omads-email+xml* format

16.4.11. Building tests

This section presents the steps needed to complete the test in terms of scripting files.

Editing the build.xml file

The *build.xml* file contains the commands that will be executed by the automated script. Below is an example of *build.xml* file:

```
<?xml version="1.0"?>

<!DOCTYPE project [
<!ENTITY test-wi SYSTEM "file:./test-wi.xml">
]>

<!-- $Id: build.xml, Exp $
  ============================================================================
    This is the driver script for QA testing, not at unit level, but at a
    protocol and client level.
  ============================================================================
-->
<project name="Funambol DS Server Test" default="start_test" basedir=".">
```



```
    <target name="start_test" depends="init_data"  />
    <target name="msg5.xml"   depends="update_item"/>
    <target name="end_test"   depends="reset_data" />

    <target name="init_data">
        <sql driver    = "${jdbc.driver}"
             url       = "${jdbc.url}"
             userid    = "${jdbc.user}"
             password  = "${jdbc.password}"
             classpath = "${jdbc.classpath}"
             onerror   = "continue"
             autocommit= "true"
             src       = "msg1.sql"
        />
        <sleep seconds="${sleep-seconds}"/>
    </target>

    <target name="update_item">
        <sql driver    = "${jdbc.driver}"
             url       = "${jdbc.url}"
             userid    = "${jdbc.user}"
             password  = "${jdbc.password}"
             classpath = "${jdbc.classpath}"
             onerror   = "continue"
             autocommit= "true"
             src       = "msg5.sql"
        />
        <sleep seconds="${sleep-seconds}"/>
    </target>

    <target name="reset_data">
        <sql driver    = "${jdbc.driver}"
             url       = "${jdbc.url}"
             userid    = "${jdbc.user}"
             password  = "${jdbc.password}"
             classpath = "${jdbc.classpath}"
             onerror   = "continue"
             autocommit= "true"
             src       = "msg-end.sql"
        />
        <sleep seconds="${sleep-seconds}"/>
    </target>
</project>
```

The *start_test* target depends on the execution of the *init_data* target, which calls the *msg1.sql* script. After initializing the database, the *msg1.xml* message is executed together with all the following messages.

Before executing *msg5.xml*, the script launches the *msg5.sql* script that updates the server's contacts, and then all the following messages are executed.

At the end, the *reset_data* target is called to restore the initial state.

Editing the msg1.sql file

The content of this scripting file depends on the type of item you are working on.

Supposing you are working with contacts:

```
--
-- Initialization data for testing
--
-- @version $Id: msg1.sql, Exp $
--
UPDATE fnbl_id SET counter=1, increment_by=1 WHERE idspace='pim.id';
DELETE FROM fnbl_client_mapping WHERE principal=
 (SELECT MIN(id) FROM fnbl_principal WHERE username='guest');
DELETE FROM fnbl_last_sync WHERE principal=
 (SELECT MIN(id) FROM fnbl_principal WHERE username='guest');
DELETE FROM fnbl_pim_address WHERE contact IN
 (SELECT id FROM fnbl_pim_contact WHERE userid='guest');
DELETE FROM fnbl_pim_contact_item WHERE contact IN
 (SELECT id FROM fnbl_pim_contact WHERE userid='guest');
DELETE FROM fnbl_pim_contact_photo WHERE contact IN
 (SELECT id FROM fnbl_pim_contact WHERE userid='guest');
DELETE FROM fnbl_pim_contact WHERE userid='guest';
```

Supposing you are working with events:

```
--
-- Initialization data for testing
--
-- @version $Id: msg1.sql, Exp $
--
UPDATE fnbl_id SET counter=1, increment_by=1 WHERE idspace='pim.id';
DELETE FROM fnbl_client_mapping WHERE principal=
 (SELECT MIN(id) FROM fnbl_principal WHERE username='guest');
DELETE FROM fnbl_last_sync WHERE principal=
 (SELECT MIN(id) FROM fnbl_principal WHERE username='guest');
DELETE FROM fnbl_pim_calendar_exception WHERE calendar in
 (SELECT id FROM fnbl_pim_calendar WHERE userid='guest');
DELETE FROM fnbl_pim_calendar WHERE userid='guest';
```

Supposing you are working with tasks:

```
--
-- Initialization data for testing
--
-- @version $Id: msg1.sql, Exp $
--
UPDATE fnbl_id SET counter=1, increment_by=1 WHERE idspace='pim.id';
```



```
DELETE FROM fnbl_client_mapping WHERE principal=
 (SELECT MIN(id) FROM fnbl_principal WHERE username='guest');
DELETE FROM fnbl_last_sync WHERE principal=
 (SELECT MIN(id) FROM fnbl_principal WHERE username='guest');
DELETE FROM fnbl_pim_calendar_exception WHERE calendar in
 (SELECT id FROM fnbl_pim_calendar WHERE userid='guest');
DELETE FROM fnbl_pim_calendar WHERE userid='guest';
```

Supposing you are working with notes:

```
--
-- Initialization data for testing
--
-- @version $Id: msg1.sql, Exp $
--
UPDATE fnbl_id SET counter=1, increment_by=1 WHERE idspace='pim.id';
DELETE FROM fnbl_client_mapping WHERE principal=
 (SELECT MIN(id) FROM fnbl_principal WHERE username='guest');
DELETE FROM fnbl_last_sync WHERE principal=
 (SELECT MIN(id) FROM fnbl_principal WHERE username='guest');
DELETE FROM fnbl_pim_note WHERE userid='guest';
```

This script deletes all items' information related to the user *guest*.

In this script it is possible to specify the charset used by the device by running the following update:

```
update fnbl_device set charset='ISO-8859-1' where id='syncml-phone';
```

Supposing you are working with emails:

```
insert into fnbl_email_folder(guid, source_uri, principal, parentid, path)
values('ROOT/I', 'mail', (select id from fnbl_principal where username='guest' and
device='fjm-1215355df35b917ee82'), 'ROOT','Inbox');
insert into fnbl_email_folder(guid, source_uri, principal, parentid, path)
values('ROOT/O', 'mail', (select id from fnbl_principal where username='guest' and
device='fjm-1215355df35b917ee82'), 'ROOT','Outbox');
insert into fnbl_email_folder(guid, source_uri, principal, parentid, path)
values('ROOT/S', 'mail', (select id from fnbl_principal where username='guest' and
device='fjm-1215355df35b917ee82'), 'ROOT','Sent');
insert into fnbl_client_mapping(principal, sync_source, luid, guid, last_anchor)
values((select id from fnbl_principal where username='guest' and device='fjm-
1215355df35b917ee82'), 'mail', 'Inbox', 'ROOT/I', '0');
insert into fnbl_client_mapping(principal, sync_source, luid, guid, last_anchor)
values((select id from fnbl_principal where username='guest' and device='fjm-
1215355df35b917ee82'), 'mail', 'Outbox', 'ROOT/O', '0');
insert into fnbl_client_mapping(principal, sync_source, luid, guid, last_anchor)
values((select id from fnbl_principal where username='guest' and device='fjm-
1215355df35b917ee82'), 'mail', 'Sent', 'ROOT/S', null);
```

This script inserts the mapping of the folders in order to avoid syncing both folders and emails. It is not necessary to clean other tables since the *email-testsuite* automatically cleans the database and the file system before running each test.

Editing the msg5.sql file

The content of this scripting file depends on the type of item you are working on.

Supposing you are working with contacts:


```
--
-- Initialization data: updating all contacts
--
-- @version $Id: msg5.sql, Exp $
--

update fnbl_pim_contact
set status = 'U',
last_update = (select end_sync from fnbl_last_sync
               where principal =
               (select MIN(id) from fnbl_principal where username='guest')
               and sync_source='card'
               );
```

Supposing you are working with events:

```
--
-- Initialization data: updating all contacts
--
-- @version $Id: msg5.sql, Exp $
--

update fnbl_pim_calendar
set status = 'U',
last_update = (select end_sync from fnbl_last_sync
               where principal =
               (select MIN(id) from fnbl_principal where username='guest')
               and sync_source='cal'
               );
```

Note: for devices that have separate tasks synchronization (such as the Sony Ericsson) and use the event SyncSource, the script to be used is the following:

```
--
-- Initialization data: updating all contacts
--
-- @version $Id: msg5.sql, Exp $
--

update fnbl_pim_calendar
set status = 'U',
    last_update = (select end_sync
                   from fnbl_last_sync
                   where principal=
                     (select MIN(id) from fnbl_principal where username='guest')
                   and sync_source='event'
                  );
```

Supposing you are working with tasks:


```
--
-- Initialization data: updating all tasks
--
-- @version $Id: msg5.sql, Exp $
--

update fnbl_pim_calendar
set status = 'U',
last_update = (select end_sync from fnbl_last_sync
               where principal =
               (select MIN(id) from fnbl_principal where username='guest')
               and sync_source='task'
               );
```

Supposing you are working with notes:

```
--
-- Initialization data: updating all notes
--
-- @version $Id: msg5.sql, Exp $
--

update fnbl_pim_note
set status = 'U',
last_update = (select end_sync from fnbl_last_sync
               where principal =
               (select MIN(id) from fnbl_principal where username='guest')
               and sync_source='note'
               );
```

This script will update the status of all items on the server.

Editing the msg_end.sql file

The content of this scripting file is standard and looks like the following:

```
--
-- Resetting data for testing
--
-- @version $Id: msg1.sql, Exp $
--
update fnbl_device set charset='UTF-8' where id='syncml-phone';
```

This script updates the charset of the device in case it was changed by a previous script.

16.4.12. Conversion tool

In addition to tests based upon *xml* messages which have been described in the previous sections, the Funambol Device Simulator is also able to run *wbxml* tests.

In order to obtain *wbxml* messages, a conversion tool is provided to transform messages from the *xml* format. The conversion commands can be found in the *bin* directory of the Funambol Device Simulator home.

Depending on the conversion you wish to perform, run one of the following scripts:

- *xml-to-wbxml dir* (all *xml* messages will be converted to *wbxml* messages)
- *wbxml-to-xml dir* (all *wbxml* messages will be converted to *xml* messages)

The parameter *dir* is the directory containing the messages you wish to convert; all messages contained in the specified directory will be converted and placed in the same folder.

Note: the conversion tool will preserve source messages.

16.4.13. Configuring the Funambol Device Simulator

Before launching the Funambol Device Simulator tool, you will need to configure your environment following these steps:

1. include the *ant* executable in the system path

   Note: remember to use *ant* version 1.6.5

2. copy your JDBC driver in the directory *SDK_HOME/tools/sdk/lib*
3. set the *FUNAMBOL_HOME* variable if you unzip this tool in a different directory than the one where you installed Funambol, since in that case that variable is already set by default.
4. set the *JAVA_HOME* variable
5. add/remove tests you wish to launch by editing the *phone-testsuite* property in the file *SDK_HOME/tools/sdk/config/phone-testsuite/build.properties* (and remember to do the same for the other test suites). By default, all tests provided with the SDK are launched.

16.4.14. Running the Funambol Device Simulator

In order to verify that the automated test is working properly, you will need to run it against the server to be tested and verify that the execution is not blocked by any errors.

Before running the automated test, compile the *SDK_HOME/config/phone-testsuite/build.properties* file adding the test case to the *phone-testsuite* variable, as shown in the following example:

```
phone-testsuite=NOKIA/N70_CARD
```

Note: multiple tests can be run at the same time by separating them with a comma.

Remember to grant executable permission to script files, by running the *chmod* command.

To run an *xml* test on a Linux system, go to the *SDK_HOME* directory and type:

```
bin/test phone
```

On Windows, go to the *SDK_HOME* directory and type:

```
bin\test.cmd phone
```

To run a *wbxml* test on a Linux system, go to the *SDK_HOME* directory and type:

```
bin/testwbxml phone
```

On Windows, go to the *SDK_HOME* directory and type:

```
bin\testwbxml.cmd phone
```

Note: on Linux/Unix systems you need to launch test scripts with superuser privileges using *sudo*. In this case, you also need to specify the *-E* parameter in order to maintain the user environment during script execution.

Expected results

If the test was successful, at the end of the test execution the expected result is:

```
[echo] Test results
[echo] ================
[echo] NOKIA/N70_CARD: passed
```

The full report on executed tests is stored in a file. According to the type of test that was launched (*xml* or *wbxml*), you will find a file named *results.testxml* or *results.testwbxml* in the directory *report*.

16.5. Test case documentation

In this section you will learn how the documentation of an automated test case should be written.

16.5.1. Compiling the *ReadMe.txt* file

The *ReadMe.txt* file specifies which tests are performed by the Funambol Device Simulator. It must be compiled with all relevant informations on the tested synclets.

The file contains the initial state of the tested scenario and the steps performed by the test, listing in which messages the actions happen.

Below is an example of the *ReadMe.txt* taken from the Nokia N70 automated test for contacts:

```
Initial state:
Server with no contacts


Step performed:


1. (1-4) Slow sync. The mobile (a Nokia N70) sends 5 contacts to the
         server. Device capabilities are sent as well in this first step.
         Input synclets are tested in this step:
         . NokiaXin.bsh
               a. Processes the incoming vcard items and adds the missing tokens.
               b. Handles large objects.
               c. Replace the property X-EPOCSECONDNAME with NICKNAME


2. (5-8) Fast sync: contacts status and last_timestamp are updated.
         The server sends all the contacts back to the device.
         Output synclets are tested in this step:
         . NokiaS60out.bsh
               a. Moves some TEL properties to the end of the VCARD in a special order
                  because the devices don't behave properly at update time, otherwise.
               b. Removes TYPE information from PHOTO token because the phones of the
                  series 60 are not able to understand this information properly.
               c. Replaces the TEL;CAR;VOICE token into TEL;VOICE;CAR.
               d. Replace the property NICKNAME with X-EPOCSECONDNAME.
```

16.6. Funambol Device Simulator test case directory structure

The test case standard uses a specific directory structure and naming convention:

```
[manufacturer]/
```



```
        [model_SyncSource]/
                error/
                reference/
                response/
```

where:

- *Manufacturer* is the manufacturer's name (e.g. *NOKIA*)
- *Model_SyncSource* indicates the device's model and the tested SyncSource (e.g. *N70_CARD*)
- the *error* directory contains the errors reported during the automated test execution
- the *reference* directory contains the server's response messages, which will be used as reference during the automated test execution to verify the correct behavior of the server that is being tested. The messages contained in this directory correspond to the expected server's response.
- the *response* directory contains the actual server's response to the messages sent by the automated client. They represent how the tested server replies to the SyncML messages of the simulated device.

Below is an example of how a test case directory should look:

```
NOKIA/
        N70_CARD/

                build.xml
                header.properties
                msg1.sql
                msg1.xml
                msg2.xml
                msg3.xml
                msg4.xml
                msg5.sql
                msg5.xml
                msg6.xml
                msg7.xml
                msg8.xml
                msg-end.sql
                ReadMe.txt
                error/
                reference/
                        msg1.xml
                        msg2.xml
                        msg3.xml
                        msg4.xml
                        msg5.xml
                        msg6.xml
                        msg7.xml
                        msg8.xml
                response/
                        msg1.xml
```



```
msg2.xml
msg3.xml
msg4.xml
msg5.xml
msg6.xml
msg7.xml
msg8.xml
```

Note: the *manufacturer* and *model_SyncSource* names are in upper case format.

The directory structure exemplified above must be located under the FUNAMBOL_HOME/ds-server/test/phone-testsuite/testcases/ directory.

16.7. Funambol Custom Ant Task

In order to make the development of an automatic test tool based on ant tasks easier, we introduced some custom tasks that help developers to quickly setup automated tests, simulating real devices interacting with the server.

Those tasks are used in the device simulator and we're going to describe them in more detail in the next sub sections.

In order to define the following tasks, we included the following global definition in the test.xml of the device simulator:

```
<taskdef resource  = "com/funambol/ant/funambol-tasks.properties">
```

we're assuming that all the needed jar files are contained in the classpath. With that definition, you include the task described in the table below:

Task Name	*Task class*
iterate	com.funambol.ant.IterateTask
httppost	com.funambol.ant.http.PostMethodTask
filesize	com.funambol.ant.FileSizeTask

If you want to use Funambol ant tasks separately in your own projects, you can separately define, only the tasks you need. Take into consideration that the following dependencies are required:

- commons-lang version 2.5
- commons-httpclient version 3.0

16.7.1. Iterate Task

It's a custom task that repeatedly calls a target passing different values from a delimited string.

The table below shows which are the task attributes:

Attribute	*Description*	*Required*
target	This is the name of the target to call.	Yes
property	The name of the property in which each value from the item list will be stored for each iteration. This allows the called target to access each item from the item list.	Yes
items	A delimited list of string items.	Yes

Attribute	*Description*	*Required*
inheritAll	Boolean to enable/disable the called target from inheriting all the properties from the environment.	No
delimiter	The delimiter that is used to separate the strings in the item list.	No

Let's look at some examples for the iterate task.

```
<project name="Test" default="main" basedir=".">
   <taskdef name="iterate" classname="com.funambol.test.tools.ant.IterateTask"/>


   <target name="main">
     <iterate target="hello" itemList="1,2,3" property="i" >
     </iterate>
   </target>


   <target name="hello">
     <echo message="${i}" />
   </target>
 </project>
```

The above example will call the “hello” *<target>* three times, each time passing a value from the item list. In this case the “hello” *<target>* will echo 1, then 2 and then 3.

Another interesting example is the following:

```
<pre>
 <target name="build" depends="init">


   <!-- iterate through the ${build.modules} variable, compiling each module specified
-->
               <iterate     target="javac"     itemList="myModule,myModule/mySubModule"
property="iterate.module"/>
 </target>


     <target     name="javac"     depends="checkSource,     cleanBuild,     prepareBuild"
if="compile.source.exist">


   <javac  srcdir="${iterate.module}/src" destdir="${iterate.module}/build">
     <include name="**\/*.java"/>
   </javac>


   <!-- create a jar file for each module-->
   <mkdir dir="${iterate.module}/lib"/>
   <jar jarfile="${iterate.module}/lib/classes.jar">
     <fileset dir="${iterate.module}/build"/>
   </jar>
 </target>
 </pre>
```


This extracted code produces the following result, multiple source directories are compiled and packaged into multiple jar files, in particular:

- compiles the myModule/src directory into myModule/lib/classes.jar
- compiles the myModule/mySubModule/src directory into myModule/mySubModule/lib/classes.jar

We used this task in the test.xml to build the report about the running of each test suite:

```
        <iterate target    = "result_test"
                 items     = "${test-list}"
                 property  = "test"
                 delimiter = ","/>
```

16.7.2. PostMethodTask

This task can be used to perform http post request against a particular server/url in order to post file data or parameters but since multipart encoding isn't supported yet, if both are present, the parameters are put in the url as a query string while the data file are sent into the http post request.

Both input file names and parameters are not required since it's possible to perform a http request without data in the body.

If you want to define this task into your build file without using the global definition, you can insert the following xml extract

```
<taskdef name="httppost" classname="com.funambol.ant.http.PostMethodTask">
    <classpath>
     <fileset dir="lib">
      <include name="**/*.jar"/>
    </fileset>
   </classpath>
  </taskdef>
```

The task is based on the commons-httpclient api version 3.0 (http://hc.apache.org/) and can be configured providing the attributes described in the table below. The ${testdir} property represents the directory of the test that is run.

Attribute	*Value*	*Default*	*Required*
url	Is the url of the server to which the http post request must be sent to. It may contain parameters but if nested parameter elements are defined, the ones contained in the url will be overwritten.		Yes
contentType	Is the content type value set into the header (Content-Type) of the http request.	application/octet-stream	No
inputfilename	Is the name of the file of which the content must be sent in the request body. The inputfile can be left out since we can send an http request without data in the body.		No
responsefilename	Is the name of the file where the http response may be written. The response is stored on that file only if the http response code matches the expected response code on the contrary the errorfilename is used. The file is opened in append mode only if the append attribute is set to true, otherwise the existing content is lost.	${testdir}/response/http-response.txt	No
errorfilename	Is the name of the file where the http	${testdir}/error/http-error.txt	No

Attribute	Value	Default	Required
	response may be written. The response is stored on that file only if the http response code doesn't match the expected response code, on the contrary the responsefilename is used. The file is opened in append mode only if the append attribute is set to true, otherwise the existing content is lost.		
login	Is the login to be used in the credentials for http authentication.		Only if any authenticati on schema is enabled.
password	Is the password to be used in the credentials for the http authentication.		Only if any authenticati on schema is enabled.
expectedResponseCode	Is the expected value of the response code.	200	No
append	Is a flag that establishes whether the response/error file are written in append mode or not.	true	No
debug	Is a flag that establishes whether the debug calls are shown or not.	false	No
failOnError	Is a flag that establishes whether the task must fail in case of error or not.	true	No
useHttpBasicAuthentication	Is a flag that establishes whether the http basic authentication must be used or not.	true	No
usePreemptiveAuthentication	Is a flag that establishes whether the authentication schema enabled (if any) must be used in a preemptive mood, i.e. in the first interaction with the server.	true	No
useChunkedEncoding	Is a flag that establishes whether the chunked encoding must be used or not, sending data to the server	false	No

Besides, it is possible to define these task headers and parameters as nested elements and they only require name and value attributes. See the example below:

```
<httppost url="http://localhost:8080/services/services"
          contentType="image/jpg"
          inputfilename="${basedir}/images/bricolage.jpg"
          responsefilename="${basedir}/response.log"
          errorfilename="${basedir}/error.log"
          useHttpBasicAuthentication="false"
          usePreemptiveAuthentication="false"
          failOnError="false"
          append ="false"
          expectedResponseCode="401"
          useChunkedEncoding="true">
    <header name="x-funambol-luid" value="1234567890"/>
    <header name="x-funambol-syncdeviceid" value="fwm-50F0063006B000000"/>
    <header name="x-funambol-file-size" value="${filesize}"/>
    <parameter name="action" value="content-upload"/>
    <parameter name="login" value="utente"/>
    <parameter name="password" value="password"/>
</httppost>
```


According to the xml extract in the example, the task will perform a post http request to the specified url, loading the input file with the name ${basedir}/images/bricolage.jpg in the post body. The body data will be uploaded using the chunked encoding since the corresponding attribute is set to true.

The authentication is disabled so the login and password are not needed. The response file is specified.

Since there are parameters nested elements, they will be added to the url as a query string.

There are headers that will also be added to the http request. The Content-Type header is handled with special attributes and will be set to the “image/jpg value”.

The expected response code is 401, so the purpose of this usage is to test that the server will reply with an error code and the corresponding http response will be stored in the file ${basedir}/response.log. If any other code is returned upon the http request, than the task considers this event as an error (since the returned http code differs from the expected one) and the response is stored in the ${basedir}/error.log file.

The output file will be written erasing the previous content of the file as the append attribute is set to false.

In the device simulator, we use this task providing the minimum set of attributes, as you can see below:

```
<target name="upload_file">
      <httppost url="${media-picture-url}"
                inputfilename="${basedir}/images/logo.png"
                login="${media-picture-login}"
                password="${media-picture-password}">
                                                    <header     name="x-funambol-luid"
value="file:///store/home/user/pictures/IMG00002.jpg"/>
          <header name="x-funambol-syncdeviceid" value="syncml-phone"/>
          <header name="x-funambol-file-size" value="${picturesize}"/>
      </httppost>
    </target>
```

Where there are some properties defined in the test.xml and their values are:

```
media-picture-url=http://localhost:8080/sapi/media/picture?action=content-
upload&login=guest&password=guest
media-picture-login=guest
media-picture-password=guest
```

As you can see, it's possible to specify request parameters directly in the url attributes. In this case, no parameter elements are listed as nested element of the http post task.

16.7.3. FileSizeTask

Is a simple task that can be used to calculate the size of a file in bytes and store this information in a desired property.

The task class is com.funambol.ant.FileSizeTask and assumes to be defined as the filesize task, you can use it as follows:

```
<filesize propertyName="imgsize" fileName="${basedir}/images/image.jpg"/>
```

In this way, if the file *${basedir}/images/image.jpg* is found, its size is calculated and stored into the property *imgsize*.

17. Appendix A - Sync4j Interchange Formats

Note: starting from Funambol version 7.1, both the server and the client will use vCard as preferred format; the server will still accept the old format for backward compatibility. Since vCard and vCalendar/iCalendar are widely used standards and they must be supported anyway, starting from Funambol version 7.1 the SIF representation will be smoothly phased out.

The Sync4j Interchange Format (SIF) is a way to represent PIM data coming from different clients in a common structure to make it easier information exchange.

SIF format is based on a XML representation of PIM data.

There is a SIF representation for each type of PIM data:

Previous versions sections **17.1 SIF-C, 17.2 SIF-E, 17.3 SIF-T** have been deprecated, these are still supported for a short period of time for backward compatibility, but development has to be done using vCard and vCal. Please refer to previous versions for more details.

SIF-N for notes will still be maintained and is documented below for reference and easily understanding existing code, even if they are not suggested for new developments.

17.1. SIF-N

SIF notes are also represented in a SIF format:

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
  <Body>New Note the first note</Body>
  <Categories/>
  <Subject>New Note</Subject>
  <Color>3</Color>
  <Height>166</Height>
  <Width>200</Width>
  <Left>80</Left>
  <Top>80</Top>
</note>
```

The fields defined and used by Sync4j are listed in the following table.

Property	*Description*
Body	Returns or sets the clear-text body of the note item.
Categories	Returns or sets the categories assigned to the note item.
Color	Color of note
Date	Date of received note
Height	Height of the box note
Left	Left position of the box of the note
Subject	Returns or sets the subject for the note item. This property corresponds to the MAPI property PR_SUBJECT. The Subject property is the default property for Outlook items. IT IS READ ONLY FOR NOTES. It value is retrieved by the first line of the body.
Top	Top position of the box of the note
Width	Width of the box note

The following table lists all fields and their use on clients:

Property	*Outlook*	*Pocket PC*	*Exchange*
Body	Y	Y	Y
Categories	Y	N	N
Color	Y	N	N
Date	N	N	Y
Height	Y	N	N
Left	Y	N	N
Subject	Y	Y	Y
Top	Y	N	N
Width	Y	N	N

17.1.1. Constants

The following constants are defined:

OlNoteColor

```
olBlue                  = 0;
olGreen                 = 1;
olPink                  = 2;
olYellow                = 3;
olWhite                 = 4;
```


18. Appendix B – List of acronyms

API	Application Programming Interface
CTP	Client TCP Push
CVS	Concurrent Versions System
DB	DataBase
DS	Data Synchronization
GUI	Graphical User Interface
GUID	Globally Unique IDentifier
HTTP	HyperText Transfer Protocol
IDE	Integrated Development Environment
IMAP	Internet Message Access Protocol
IP	Internet Protocol
JDK	Java Development Kit
JVM	Java Virtual Machine
LUID	Local Unique IDentifier
OMA	Open Mobile Alliance
OTA	Over The Air
PDA	Personal Digital Assistant
PIM	Personal Information Management
POM	Project Object Model
POP	Post Office Protocol
REST	REpresentational State Transfer
SDK	Software Development Kit
SIF	Sync4j Interchange Format
SMS	Short Message Service
SMTP	Simple Mail Transfer Protocol
SVN	Subversion
TCP	Transmission Control Protocol
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WS	Web Service
XML	eXtensible Markup Language

19. Resources

This section lists resources you may find useful.

[1] *SyncML Representation Protocol, version 1.1*,
http://www.syncml.org/docs/syncml_represent_v11_20020215.pdf

[2] *SyncML Sync Protocol, version 1.1*,
http://www.syncml.org/docs/syncml_sync_protocol_v11_20020215.pdf

[3] Funambol Installation and Administration Guide

[4] Apache maven, http://maven.apache.org/index.html

[5] Community projects, http://www.funambol.com/opensource/projects.php

[6] Funambol public source code repository, https://core.forge.funambol.org/source/browse/core/

[7] Funambol SDK, http://m2.funambol.com/repositories/artifacts/funambol/sdk/8.0.0/sdk-8.0.0.tar.gz

[8] OpenXchange connector, https://funamboloxconnector.forge.funambol.org

[9] Exchange connector, https://exchange-connector.forge.funambol.org

[10] AGPL, http://www.fsf.org/licensing/licenses/agpl-3.0.html

[11] Liferay customization guide:
http://content.liferay.com/4.2/doc/installation/liferay_4_customization_guide.pdf

[12] JavaMail API: http://java.sun.com/products/javamail/

