Important Information

© Copyright Funambol, Inc. 2006. All rights reserved.

The information contained in this publication is subject to US and international copyright laws and treaties. Except as permitted by law, no part of this document may be reproduced or transmitted by any process or means without the prior written consent of Funambol, Inc.

Funambol, Inc. has taken care in preparation of this publication, but makes no expressed or implied warranty of any kind. Funambol, Inc. does not guarantee that any information contained herein is and will remain accurate or that use of the information will ensure correct and faultless operation of the relevant software, service or equipment.

Funambol, Inc., its agents and employees shall not be held liable for any loss or damage whatsoever resulting from reliance on the information contained herein.

Funambol and Sync4j are trademarks and registered trademarks of Funambol, Inc.

All other products mentioned herein may be trademarks of their respective companies.

Published by Funambol, Inc., 643 Bair Island Road, Suite 305, Redwood City, CA  94063
## Contents

### Chapter 1  SyncML Device Management

- Introduction .......................................................... 1
- Purpose ................................................................. 2
- SyncML DM Protocol Message Sequence Overview ................. 2
- SyncML Device Management Tree Overview ......................... 3
- The ./DevInfo Node .................................................. 6
- Management Object Manipulation .................................. 8
- Management Object Security ....................................... 8
- SyncML Security and Initial Provisioning (Bootstrap) ............. 9
- Security ................................................................. 9
- Bootstrap Provisioning .............................................. 9
- Bootstrap Methods ..................................................... 10

### Chapter 2  Developer Overview

- System Architecture ................................................ 11
- Funambol DM Server Architecture ................................ 12
- Execution Flow ....................................................... 14
- Database Schema .................................................... 16
- Device Management State .......................................... 18

### Chapter 3  Server Configuration

- Overview ............................................................... 21
- Funambol.properties ................................................. 22
- Server JavaBeans ..................................................... 23
- Lazy Initialization ................................................... 25
- Configuring a Standard Component ................................ 26
- Configuring a Custom Component ................................ 27
- Getting a Configured Instance ..................................... 29
- Tips and Tricks ....................................................... 29
- Logging ................................................................. 30
- Adding Logging for Custom Components ......................... 31

### Chapter 4  Customizing Message Processing

- Overview ............................................................... 33
- Preprocessing an Incoming Message ............................... 34
- Creating an Input Synclet ......................................... 37
DM SERVER DEVELOPER'S GUIDE

Chapter 5 Implementing Management Operations ................................................. 41
Overview .................................................................................................................. 42
Creating a Processor Selector .............................................................................. 43
DeviceIdProcessorSelector .................................................................................. 43
Configuring the Management Engine ................................................................. 48
Creating a Management Processor ..................................................................... 49
ManagementOperation ...................................................................................... 50
ManagementOperationResult ............................................................................ 51
Using Scripting Management Processors ........................................................... 53
Scripting Variables ............................................................................................... 53

Chapter 6 External Applications ........................................................................... 59
External Application Interfaces ............................................................................ 60
The EJB Layer ........................................................................................................ 60
Implementing the Sender Component ................................................................. 62
Sender Interface ...................................................................................................... 62
Sender Component Configuration ........................................................................ 64

Chapter 7 Bootstrapping Devices ...................................................................... 65
Bootstrap Overview ............................................................................................... 66
WAP Provisioning Profile .................................................................................... 69
Plain Profile ............................................................................................................ 70

Appendix A Appendix ......................................................................................... 71
Resources ................................................................................................................ 72
Related Documentation ....................................................................................... 72
Other Resources .................................................................................................... 73
Bootstrap XML Message Examples .................................................................... 74
WAP Profile ........................................................................................................... 74
Plain Profile ............................................................................................................ 75
WAP Headers for Bootstrap Message ................................................................. 80
PLAIN Profile ........................................................................................................ 80
WAP Profile ............................................................................................................ 80
Notification Message Using WAP Push .............................................................. 82
Notification Message Created by the DM Server ............................................... 82
Complete SMS (WDP + WSP + Notification Message) ....................................... 82
Chapter 1    SyncML Device Management

Topics

- Introduction, page 2
- SyncML DM Protocol Message Sequence Overview, page 3
- SyncML Device Management Tree Overview, page 6
- SyncML Security and Initial Provisioning (Bootstrap), page 9
Introduction

The Open Mobile Alliance Device Management (OMA DM) protocol, formerly known as the SyncML DM protocol, specifies the message sequence and behaviors that will allow device management commands to be executed against management objects on a SyncML DM compliant device. Management objects might include configuration parameters that enable Internet connectivity, e-mail connectivity, WAP connectivity, MMS settings, and basic network configuration options allowing voice access to an operator network. Other management objects may include the Java Runtime Environment on J2ME enabled devices or any other applicable software environment where extensions of features and functionality can be added via an Over The Air (OTA) upgrade to those environments. SyncML DM protocol is not limited to any particular set of management objects that can be modified via OTA, although the protocol does define a specific methodology and object management tree structure that serves as a profile on how a DM server accesses specific management objects on a particular device.

The Funambol DM Server is a server side implementation of the OMA DM protocol and an extensible framework for the development of device management based applications. The Funambol DM Server architecture and implementation derives from the Funambol OMA DS platform.

Purpose

This document provides developers with basic concepts and guidance for extending the functionality of the Funambol DM Server. Using this document, a developer will acquire the following skills:

- An understanding of the OMA DM protocol
- An understanding of the Funambol DM Server architecture
- The ability to integrate the Funambol DM Server with external applications
- The ability to pre- and post-process incoming and outgoing OMA DM messages
- The ability to implement new management operations
SyncML DM Protocol Message Sequence Overview

The SyncML DM protocol is relatively simple from a messaging sequence standpoint. The message sequence consists of three parts:

- Alert phase – used only for unsolicited management initiation from the server to the client.
- Set up phase (authentication and device information exchange)
- Management phase

Transaction 1: Alert Phase – Server to Client Only

Not required if the client is contacting server. SyncML DM supports the concept of unsolicited alerts via a “notification initiation alert” mechanism. This mechanism allows a management server to initiate a management session with a device, rather than solely relying on a client device to initiate a session. Some devices may be capable of listening on a particular port for alert messages; other devices may not be capable of this paradigm and need an alternate method to trigger a management session. SyncML will rely on two primary methods for delivery of unsolicited alerts:

- WAP Push – This method will deliver the alert via a Push Initiator through a Push Proxy Gateway as defined by the WAP protocol. The SyncML server will act as a Push Initiator in this example, and will deliver the message via an SMS message. The message will have a unique application ID and the message will be routed to the device management user agent per the WAP Specification.
- OBEX – The OBEX protocol can be used to deliver unsolicited alerts to a device via the PUT command as defined by the protocol.

Transaction 2: Set Up Phase – Client to Server

Always required. The set up phase consists of a request from the client and the response from the server. The initial client request of the set up phase will contain three primary pieces of information.

- The information contained in the DevInfo (Device Information) object, as follows:
  - **Ext** - An optional, internal object, marking up the single branch of the DevInfo sub tree into which extensions can be added, permanently or dynamically.
  - **Bearer** - An optional, internal object, marking up a branch of the DevInfo sub tree into which items related to the bearer (CDMA, etc.) are stored. Use of this sub tree can be mandated by other standards.
  - **DevId** - A unique identifier for the device. SHOULD be globally unique.
  - **Man** - The manufacturer identifier.
  - **Mod** - A model identifier (manufacturer specified string).
DmV - A SyncML device management client version identifier (manufacturer specified string).
Lang - The current language setting of the device.

- The client credentials information used for authenticating the client.
- A token that informs the server if this is a client-initiated session or server-initiated session. This is required so the server can synchronize a server-initiated session with an initial incoming request from the client. From the server perspective, server-initiated sessions will look the same as client-initiated sessions, and a token must be present so the server can distinguish both types of transactions.

Transaction 3: Set-up Phase – Server to Client

Always required. The server will respond to the initial client request with server credentials, so as to identify the server to the client for authentication and identification purposes. The server may also send user interaction commands with the response, as well as initial management data.

Transaction 4: Management Phase – Client to Server

Only required if management data or user interaction commands were sent in the previous message. The client will respond to the server with the results of the management message sent in the previous transaction, as well as any user interaction command results.

Transaction 5: Management Phase – Server to Client

Always required if Transaction 4 was initiated. This transaction will occur to either close the management session or to begin a new iteration if more management operations are needed. If additional management operations are needed the response to this message will be the same transaction type as transaction 4. This iteration will continue until the management server sends a message in this transaction to close the session with the client.
The diagram below is a representation of the preceding transactions.
SyncML Device Management Tree Overview

The SyncML DM protocol identifies various messages and message content, the sequence of messages, security framework, and so on. The device itself must adhere to a specific methodology of managing various functions. Because features and functionality are device specific and are often proprietary, a framework defining how a device utilizes device management messages must be specified and in place to operate properly with a device management server. This framework allows a device manufacturer to add new devices or functionality to the market, then modify or add a new device description to the device management server’s library of device profiles. This framework is defined as the Device Management Tree. The tree data structure allows URI addressing of SyncML DM messages, as well as provides a common framework for device management object addressing.

The device management tree is a data structure of manageable device objects. Device objects can be anything from a single parameter, to a splash screen GIF file, to an entire application. The device management tree is essentially mapped to permanent or dynamic objects as an addressing schema to manipulate these objects. Permanent objects can be thought of as objects that are built into the device at the time of manufacture and typically cannot be deleted e.g., the Device Info object that defines the basic information about a device such as manufacturer or model number. Dynamic objects are objects that can be added or deleted, e.g., ringtones or wallpaper.

The ./DevInfo Node

The initial request from the client will always contain information retrieved from the ./DevInfo (or Device Info) sub tree. The ./DevInfo Node is only part of the overall device management tree structure, and it maps to basic device parameters that allow initial operations and inspection of the device by the CRM specialist.
The diagram below is an example that illustrates how the device management tree maps to certain objects.

Properties of Management Tree Objects

Each management tree object has a set of properties that defines metadata information about the object, such as access control, etc. The properties are as follows:

- **ACL** (required) – Access Control List, defines who can manipulate the underlying object.
- **Format** (required) – determines how an object is interpreted, i.e., if the underlying object is a URL for a particular management server, the Format may be defined as chr (character).
- **Name** (required) – the object in the tree.
- **Size** (required for leaf objects, not applicable for interior nodes) – the size of the object in bytes.
- **Title** (optional) – user-friendly name.
- **Tstamp** (optional) – the time stamp of the last modification.
- **Type** (required for leaf objects, optional for interior nodes) – the MIME type of the object.
- **VerNo** (optional) – the version number of the object.
**Management Object Manipulation**

Management objects can be manipulated via SyncML messages with the following commands through a valid SyncML DM message.

- **Add** – Adds an object (Node) to a tree.
- **Get** – Returns a Node name based on the URI passed with the GET request
- **Replace** – Replaces an object on the Tree.
- **Delete** – Deletes an object on the tree.
- **Copy** – Copies an object (Node) on the tree.

**Management Object Security**

The ACL property defines the security framework for objects within a tree. This framework allows only certain server’s access to objects for manipulation. This will allow tight control on how objects are added, changed, deleted or replaced, as well as how object properties are manipulated, and more importantly *who* is allowed to manipulate objects.
SyncML Security and Initial Provisioning (Bootstrap)

Security

Security is a primary concern when modifying any attributes on a device. SyncML DM protocol specifies that authentication take place in either the transport level or the SyncML DM protocol level. If the transport level authentication is considered too weak, then authentication must occur at the protocol level.

Example 1 – Transport Level Authentication

A device may authenticate itself to a WAP server using basic HTTP authentication. Authentication credentials accompany each request after the initial transaction is sent to the WAP gateway. The WAP gateway in this case would be considered “trusted” since it serves as a Proxy to the SyncML DM server, and additional authentication may not be required at the SyncML protocol level if the requests come via the trusted proxy.

Example 2 – Session Level Authentication

Similar to example 1, this example assumes that a GPRS device authenticates to the operator’s portal via TLS or HTTPS. The underlying session is established and considered authenticated therefore any messages that are a part of this secured session can be considered authenticated.

Example 3 – SyncML Protocol Level Authentication

If session level or transport level authentication is not available or considered weak, then the SyncML protocol level authentication must occur. SyncML requires that regardless what the underlying security mechanism that is in place, if the server or client requests credentials one or both must comply. The four basic credentials are Server ID, Username, Password, and Nonce.

SyncML DM requires that Basic, MD-5 (server side) and HMAC authentication must be supported.

Bootstrap Provisioning

SyncML DM defines two different use cases of bootstrapping a device and two methods for initial bootstrapping.

Bootstrap Use Cases

- Factory Bootstrap: Devices are loaded with SyncML DM bootstrap information at the time of manufacture or initial distribution.
- Server Initiated Bootstrap: Server initiated bootstrap is intended for devices that do not have the necessary configuration parameters set to establish a SyncML DM session.
Bootstrap Methods

WAP Profile Provisioning

Other aspects of server initiated bootstrap are very similar if not identical to WAP bootstrap provisioning. If the device supports WAP provisioning, extensions to the WAP profile that define how SyncML parameters are mapped into the SyncML DM management object are defined in the SyncML specifications, and can be used to configure the device for SyncML DM via WAP bootstrap provisioning.

Plain Profile Bootstrap Provisioning

Plain profile is currently defined for devices that do not support WAP bootstrap provisioning. This method utilizes the SyncML DM format for the bootstrap message, and uses the same bootstrap method for security as WAP bootstrap provisioning.

WAP defines several methods for authenticating a bootstrap session and these methods are utilized by the SyncML DM protocol:

- NETWPIN – a shared secret is known by the device and server i.e. an IMSI or ESN. No user intervention is required, and is the simplest yet least secure method of authenticating a bootstrap message.

- USERPIN – where the user enters a PIN code delivered out of band i.e. through customer care who will initiates the bootstrap after confirming the identity of the user. A Plain Profile can use any method capable of sending unprompted requests to a device, i.e. OBEX, SMS, and WAP Push.

- USERNETWPIN – a combination of the NETWPIN and USERPIN methods, requiring the use of a shared secret and a user PIN.

- USERPINMAC – the PIN is delivered out of band to the user. This method calculates the PIN based on the actual bootstrap method using a hashing function. When the bootstrap message arrives, the user is prompted to enter the PIN. If the PIN matches the re-hash of the bootstrap message on the device then the message is accepted.
Chapter 2  

**Developer Overview**

This section provides an overview of the Funambol DM Server architecture appropriate for developers who wish to extend the server or integrate it with other applications (i.e., a customer care front end).

**Topics**

- *System Architecture, page 12*
- *Execution Flow, page 14*
- *Database Schema, page 16*
- *Device Management State, page 18*
System Architecture

The system architecture of the Funambol DM Server is shown below. The transport and the business logic (protocol handling) are separated in two distinct blocks and handled respectively by a web application running in a J2EE web container and by an Enterprise Java Bean running in a J2EE EJB container.

Funambol DM Server Architecture

The Funambol DM Server architecture is layered and modular, as shown below:
These layers represent groups of functionality with well-defined boundaries and communication interfaces.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transport Layer</td>
<td>The medium through which client messages reach the system. The current implementation of the Funambol DM Server implements the HTTP transport protocol and binding as defined by the HTTP binding OMA DM specification. The system is designed so that other transport protocols can be added in the future.</td>
</tr>
<tr>
<td>Protocol Layer</td>
<td>Responsible for the interpretation and handling of the SyncML protocol. It works at both representation and protocol levels. This layer is designed so that other device management protocols can be added in the future.</td>
</tr>
<tr>
<td>Server Layer</td>
<td>The Funambol DM Server implementation. It is a J2EE-based application that can be deployed on any J2EE compliant application server.</td>
</tr>
<tr>
<td>Application Layer</td>
<td>Implements the way the Funambol DM Server interacts with end user DM applications, such as the CRM applications used by customer support personnel. It is not a fully implemented layer, but more a framework used to extend the server in order to meet any application-specific needs.</td>
</tr>
<tr>
<td>Framework</td>
<td>Implements and provides services and abstractions used by the different layers to implement the component they are built of. The most important services provided by the framework are the following:</td>
</tr>
<tr>
<td></td>
<td>• Core SyncML representation and protocol</td>
</tr>
<tr>
<td></td>
<td>• Configuration framework</td>
</tr>
<tr>
<td></td>
<td>• Logging framework</td>
</tr>
<tr>
<td></td>
<td>• SyncML DM engine framework</td>
</tr>
<tr>
<td></td>
<td>• Security framework</td>
</tr>
<tr>
<td></td>
<td>• Commonly used utilities</td>
</tr>
</tbody>
</table>
Execution Flow

The execution flow of an OMA DM request is shown below:

The green blocks are external systems or applications. They interact with the Funambol DM Server directly through its EJB interface or indirectly through a web services interface (not available yet). The Notification Sender is used by the Funambol DM Server to send Pkg 0 notifications to the devices (for server initiated management sessions).

The light blue and violet boxes represent the main Funambol DM Server building blocks and are part of the core implementation. The orange blocks are components added and customized by developers to meet the end-user management application needs.

An OMA DM session can be started from the device or solicited by the server; when started from the device, the execution flow of the message is slightly shorter than when started by the server. When the server initiates the DM session, the execution flow is the following.

1. The management application (e.g., a customer support management console) starts a new “management operation” for a specific device. This results in an interaction between the external application and the Funambol DM Server, e.g., via a call to a web service deployed into the server.

2. The web service invokes the corresponding service of the DM Server EJB interface.

3. The EJB wrapper forwards the call to the management engine, which is the core of the Funambol DM Server.
4. The management engine builds the notification message (OMA DM PKG 0) and tells the Notification Sender to push it to the phone; note in the picture that the management engine is a logical component, which in reality is built of many other blocks.

5. The device receives the notification message; it starts a new OMA DM session sending the OMA DM PKG 1 to the server; the DM message is first received by the HTTP listener and processed by the Funambol DM Server’s HTTP handler.

6. The HTTP handler is now ready to open the DM session on the Funambol DM Server and start the real message processing; as show in the figure, the incoming message passes through the input pipeline before getting to the management engine.

7. The management engine processes the request, including authentication and session management.

8. To build the management commands to send to the client, the management engine selects and calls the appropriate management processor for the management operation requested in Step 1.

9. The management actions to be performed are ready to be sent to the client; the outgoing message passes through the output pipeline for post-processing.

10. The return message is translated into an OMA DM message.

11. The return message is returned to the device.

The processing now starts again at Step 5 with a new message from the client; in this case, all commands and results exchanged between client and server belong to the same session until the server stops sending commands. Note that in the case of an unsolicited new DM session, the client starts at Step 5 without receiving a notification message.
Database Schema

The internal database schema used by the Funambol DM Server is shown below.

<table>
<thead>
<tr>
<th>Table Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fnbl_user</td>
<td>Stores basic user information, such as username, first and last name, and email. The internal_user field represents an applicative user, not a real person, but an application. This field is currently not used, but it was added for future use.</td>
</tr>
<tr>
<td>fnbl_role</td>
<td>Stores the list of the available roles (for future use)</td>
</tr>
<tr>
<td>fnbl_user_role</td>
<td>Stores the associations between the users and the roles (for future use).</td>
</tr>
</tbody>
</table>
fnbl_device
Stores basic information about a device:

- device id
- description
- device type (e.g., Nokia 7650)
- digest (the MD5(user:password)) of the user this device will be associated with
- client_nonce (the nonce that the client will use to calculate the next session's digest)
- server_nonce (the nonce that the server will use to calculate the next session's digest)
- server_password (the server password)

fnbl_principal
A principal is an association between a user and a device. This allows more generic scenarios where a user can use many devices and/or a device can be used by many users. The current DM implementation allows an 1:1 association between a user and a device.

fnbl_dm_state
Stores data on pending operations to be performed:

- id: record id
- device: device id
- mssid: session id
- state: operation status. One of the following: 'N' = notified; 'P' = management session in progress; 'E' = error.
- start_ts: timestamp of the beginning of the session
- end_ts: timestamp of the end of the session
- operation: operation to be performed in this session
- info: application-specific details

fnbl_id
Used to create unique ids in a database-independent manner. There can be many counters, each identified by its own namespace.
Device Management State

The management process is usually performed in multiple phases which are not necessarily close in time or even bound to the same OMA DM session. The state of the overall management process is partially the responsibility of the management server and partially dependent on the specific management process or application. The DM Server defines and handles the following state information for a management process:

- sid - the session id notified by the management server to the client.
- deviceId - the unique identifier of the device under management
- operation - the requested operation (application-specific)
- state - the operation state (management server/application-specific)
- start_ts - timestamp of the beginning of the session
- end_ts - timestamp of the end of the session
- info - additional application-specific details

The state diagram of a management session is illustrated below. The NOT EXISTING state is not a real session state, it simply represents the absence of a session. If a device connects to the server and there is no session associated with it, no management operations are executed.

A new session is automatically created by the engine when a device is notified to start a server-initiated management session. In this case, the session is created with the state NOTIFIED ('N'). When the device starts the management session, the state moves to MANAGEMENT IN PROGRESS ('P'). During the management session, in which many device-server interactions may take place, the session stays in state P. When the session successfully completes, the state moves to COMPLETED ('C'). If an unrecoverable error occurs, the state moves to ERROR ('E').
Note that the scenario above requires that the device is notified to start the management session. When a management application needs to support a device-initiated session, the session must be created in state P. This is accomplished by creating a row in the `FNBL_DM_STATE` table, as follows:

<table>
<thead>
<tr>
<th>id</th>
<th>device</th>
<th>mssid</th>
<th>state</th>
<th>start_ts</th>
<th>end_ts</th>
<th>operation</th>
<th>info</th>
</tr>
</thead>
<tbody>
<tr>
<td>row_id</td>
<td>device_id</td>
<td>NULL</td>
<td>‘P’</td>
<td>NULL</td>
<td>NULL</td>
<td>operation to perform</td>
<td>application specific value</td>
</tr>
</tbody>
</table>

For details, see “Database Schema” on page 16.
Chapter 3  Server Configuration

Topics

- Overview, page 22
- Configuring a Standard Component, page 26
- Configuring a Custom Component, page 27
- Getting a Configured Instance, page 29
- Logging, page 30
Overview

The Funambol DM Server provides a framework for implementing many kinds of device management services. You can extend existing modules or add new modules. You configure the server using the following:

- Funambol.properties
- Server JavaBeans

Configuration files are stored in a configuration path (or configpath) containing a tree of subdirectories that is handled in the same way as the JVM classpath. The configuration path is specified by the `funambol.dm.home` system property, to which `config` is appended. For example, if the system property is set to `/opt/funambol`, the base directory of configuration files is `/opt/funambol/config/`.

Funambol.properties

This is the primary configuration file. It is located directly under the configpath and defines the following properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>server.uri</td>
<td>The server URI that identifies the server.</td>
<td><a href="http://localhost:8080/">http://localhost:8080/</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>funambol/dm</td>
</tr>
<tr>
<td>server.id</td>
<td>Server identifier.</td>
<td>funambol</td>
</tr>
<tr>
<td>syncml.dtdversion</td>
<td>The supported SyncML DTD version</td>
<td>1.1</td>
</tr>
<tr>
<td>engine.manufacturer</td>
<td>The manufacturer used in server capabilities.</td>
<td>funambol</td>
</tr>
<tr>
<td>engine.modelname</td>
<td>The model name used in server capabilities.</td>
<td>DM Server</td>
</tr>
<tr>
<td>engine.oem</td>
<td>The oem used in server capabilities.</td>
<td>-</td>
</tr>
<tr>
<td>engine.firmware version</td>
<td>The firmware version used in server capabilities.</td>
<td>-</td>
</tr>
<tr>
<td>engine.software version</td>
<td>The software version used in server capabilities.</td>
<td>3.0.x</td>
</tr>
<tr>
<td>engine.hardware version</td>
<td>The hardware version used in server capabilities.</td>
<td></td>
</tr>
<tr>
<td>engine.deviceid</td>
<td>The device identifier used in server capabilities.</td>
<td>funambol</td>
</tr>
<tr>
<td>engine.devicetype</td>
<td>The device type used in server capabilities.</td>
<td></td>
</tr>
</tbody>
</table>
Server JavaBeans

With the exception of Funambol.properties, all other Funambol DM Server components are configured as server JavaBeans. Server JavaBeans are JavaBeans used server-side. A bean configuration is stored as the serialized form of a bean instance; a bean can be instantiated, configured and serialized to persist its configuration. Later, the bean can be deserialized in a properly configured instance.

It would inconvenient, however, if a bean had to be instantiated, configured and serialized every time its configuration changes. To avoid this, the Funambol DM Server uses the standard Java facility to serialize objects into XML (and to deserialize them from XML) in the form of the java.beans.XMLEncoder and java.beans.XMLDecoder classes. Configuration files created with these classes are easy to work without a dedicated GUI – they can be created, read, and modified with a text editor. An additional advantage is that server JavaBeans do not implement java.io.Serializable because XMLEncoder does not require it.
The following is an example of a server JavaBean:

Example:

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

For server JavaBean handling, Funambol provides the factory class `com.funambol.framework.tools.beans.BeanFactory`, which in turn uses a customized class loader that handles configuration files in the configpath (class loaders normally handle classes in the class path).

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.

Lazy Initialization

When a bean is deserialized from its XML form, the classloader that loads the serialization file first calls the bean class’s empty constructor, then it sets the values of the bean properties using the setXXX() methods. However, some classes need additional work to properly initialize; that work has to be done with meaningful properties values (in other words, after the setXXX() methods are called). To support this lazy initialization approach, those classes can implement `com.funambol.framework.tools.beans.LayInitBean`, which defines a separate init() method. When the DM Server loads a LazyInitBean, after the bean instantiation (or deserialization), it calls its init() method, giving the bean the opportunity to complete its initialization.
Configuring a Standard Component

Changing a configuration bean is as easy as editing a text file. Let’s take as an example the configuration file for the NotificationSender component. The full path of the configuration bean is `com/funambol/server/engine/dm/NotificationSender.xml` (this path is relative to the `configpath`) and its contents are shown below:

Example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.2_01" class="java.beans.XMLDecoder">
<object class="com.funambol.dm.engine.MySender">
  <void property="sendingUrl">
    <string>http://theserver.com/sms/send</string>
  </void>
</object>
</java>
```

The object element specifies the Java class to be instantiated, and the property element sets the corresponding instance property. To change the sending URL (suppose the sender has a HTTP based interface) of the service used to send notifications, you simply edit and save the file. The next time this bean is used, the new configuration value will be picked up.
Configuring a Custom Component

Any Java object can be configured with this technique, from a simple Java class to a very complex Java object tree. For example, the following configures a String object:

Example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.2" class="java.beans.XMLDecoder">
  <string>Hello world</string>
</java>
```

For a more interesting example, suppose we have a “device inventory” component that can store the properties of a device, and when queried, retrieve the device’s capabilities. The class could be defined as follows:

Example:

```java
public class DeviceInventory {
    private String s1 = "s1";
    public String s2 = "s2";
    private HashMap capabilities = new HashMap();
    public DeviceInventory() {}
    public void setCapabilities(String model, Capabilities caps) {
        capabilities.put(model, caps);
    }
    public int getMaxMsgSize(String model) {
        return ((Capabilities)capabilities.get(model)).getMaxMsgSize();
    }
    public int getMaxObjSize(String model) {
        return ((Capabilities)capabilities.get(model)).getMaxObjSize();
    }
    public boolean supportNumberOfChanges(String model) {
        return ((Capabilities)capabilities.get(model)).getSupportNumberOfChanges();
    }
    public boolean supportLargeObjects(String model) {
        return ((Capabilities)capabilities.get(model)).getSupportLargeObjects();
    }
    public void setCapabilities(HashMap capabilities) {
        this.capabilities = capabilities;
    }
    public HashMap getCapabilities() {
        return capabilities;
    }
}
```
The configuration file for such a class could be as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.2_04" class="java.beans.XMLDecoder">
<object class="com.funambol.dm.examples.DeviceInventory">
  <void property="capabilities">
    <void method="put">
      <string>siemens-s55</string>
      <object class="com.funambol.dm.examples.Capabilities">
        <void property="maxMsgSize">
          <int>2700</int>
        </void>
        <void property="supportLargeObjects">
          <boolean>true</boolean>
        </void>
        <void property="supportNumberOfChanges">
          <boolean>true</boolean>
        </void>
      </object>
    </void>
    <void method="put">
      <string>nokia-7650</string>
      <object class="com.funambol.dm.examples.Capabilities">
        <void property="maxMsgSize">
          <int>5000</int>
        </void>
        <void property="maxObjSize">
          <int>10000</int>
        </void>
        <void property="supportLargeObjects">
          <boolean>true</boolean>
        </void>
        <void property="supportNumberOfChanges">
          <boolean>true</boolean>
        </void>
      </object>
    </void>
  </void>
</object>
</java>
```
Getting a Configured Instance

Configuration beans are accessed through the singleton object com.funambol.framework.config.Configuration. For example, to instantiate a configured DeviceInventory instance, use the following code:

Example:

```java
Configuration c = Configuration.getConfiguration();
DeviceInventory inventory = c.getBeanInstanceByName("com/funambol/server/dm/DeviceInventory.xml");
```

Tips and Tricks

You do not need to manually write a configuration file from scratch. To write a bean instance for the first time, use the saveBeanInstance() method from com.funambol.framework.tools.beans.BeanFactory to save a configured instance into a file, as shown below:

Example:

```java
import com.funambol.dm.examples.DeviceInventory;
import com.funambol.dm.examples.Capabilities;
import com.funambol.framework.tools.beans.BeanFactory;
DeviceInventory inventory = new DeviceInventory();
Capabilities nokia = new Capabilities();
Capabilities siemens = new Capabilities();
nokia.setMaxMsgSize(5000);
nokia.setMaxObjSize(10000);
nokia.setSupportNumberOfChanges(true);
nokia.setSupportLargeObjects(true);
siemens.setMaxMsgSize(2700);
siemens.setMaxObjSize(0);
siemens.setSupportNumberOfChanges(true);
siemens.setSupportLargeObjects(false);
inventory.setCapabilities("nokia-7650", nokia);
inventory.setCapabilities("siemens-s55", siemens);
BeanFactory.saveBeanInstance(inventory, new java.io.File("inventory.xml"));
```
Logging

The Funambol DM Server uses the standard Java logging APIs introduced with the JDK 1.4.x. The output produced by the logging system can be configured in terms of content and writing media (the system output console, the file system, a database, etc.). To configure the JDK logging system, edit the file `{funambol.dm.home}/lib/logging/common/logging.properties`.

The Funambol DM Server uses many logging namespaces allowing you to selectively enable/disable the logging of a specific module. The namespaces are as follows:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>funambol</td>
<td>The default logging namespace, used when no other namespace is specified.</td>
</tr>
<tr>
<td>funambol.dm</td>
<td>DM logging information.</td>
</tr>
<tr>
<td>funambol.dm.bootstrap</td>
<td>DM Bootstrap logging information.</td>
</tr>
<tr>
<td>funambol.dm.notification</td>
<td>DM Notification logging information.</td>
</tr>
<tr>
<td>funambol.engine</td>
<td>Synchronization engine logging information.</td>
</tr>
<tr>
<td>funambol.engine.pipeline</td>
<td>Pipeline logging information.</td>
</tr>
<tr>
<td>funambol.handler</td>
<td>Session handling logging information.</td>
</tr>
<tr>
<td>funambol.framework.engine</td>
<td>Framework engine logging information.</td>
</tr>
<tr>
<td>funambol.transport.http</td>
<td>Transport logging information.</td>
</tr>
</tbody>
</table>

To enable the maximum of verbosity for a given module, the configuration file `logging.properties` should have the following line (other loggers settings under `funambol` should be commented out):

funambol.level=ALL
Adding Logging for Custom Components

The Funambol DM Server logging feature can be used by any Funambol DM Server class or extension. It is even possible to create your own logging namespace, so that you can isolate the logging information produced by your components from all other logging.

The `java.util.logging.Logger` used for logging is acquired with the following sample code:

```java
Logger log = FunambolLogger.getLogger(name);
```

where `name` is one of the standard names defined in `com.funambol.framework.logging.FunambolLoggerName` or your own logger name. Note that “funambol” will be prepended to the given name, so that all Funambol DM Server loggers will be hierarchically grouped under the funambol name space. This allows all Funambol DM Server logging activity to be enabled or disabled by changing a single line in the `logging.properties` file.
Chapter 4  

Customizing Message Processing

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

Topics

- Overview, page 34
- Preprocessing an Incoming Message, page 37
- Postprocessing an Outgoing Message, page 39
Overview

The OMA DM protocol is an XML-based protocol. This means that each OMA DM message is an XML document.

When a OMA DM message reach the Funambol DM Server, it passes through both XML transformations and message transformations. The former works on the message in its XML representation, the latter on a Java representation of the message.

In order to save bandwidth and processing power, OMA DM messages can be also WBXML encoded. No matter how the message is coded, its content is first delivered to a SyncAdapter component by the transport layer. The SyncAdapter first translates the message in 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 XML level transformation.

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 translated into an object tree that represents exactly the message.

For example, the following DM PKG #1 message:

Example:

```xml
<SyncML xmlns='SYNCML:SYNCML1.1'>
  <SyncHdr>
    <VerDTD>1.1</VerDTD>
    <VerProto>DM/1.1</VerProto>
    <SessionID>5b</SessionID>
    <MsgID>1</MsgID>
    <Target>
      <LocURI>http://localhost:8080/funambol/dm</LocURI>
    </Target>
    <Source>
      <LocURI>FunambolTest</LocURI>
    </Source>
    <Cred>
      <Meta>
        <Format xmlns='syncml:metinf'>b64</Format>
        <Type xmlns='syncml:metinf'>syncml:auth-basic</Type>
      </Meta>
    </Cred>
  </SyncHdr>
</SyncML>
```
This message will be translated into the object hierarchy shown below.
After being translated into an object tree, an incoming message passes through the input message processing pipeline, before getting to the Management Engine. This gives the opportunity to further processing the message when it is in a manageable representation. In a similar way, a response message going out from the Management Engine, passes through the output message processing pipeline before getting translated to its XML (and then WBXML) representation.

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

Input and output message processing components are also referred to as “synclets.”
Preprocessing an Incoming Message

To preprocess an incoming message, you create and input processor component and configure the Pipeline Manager accordingly.

Creating an Input Synclet

An input synclet is a class that implements the `com.funambol.framework.engine.pipeline.InputMessageProcessor` interface. This interface defines a single method,

```java
void preProcessMessage(MessageProcessingContext context, SyncML msg)
```

where `context` is a request-scoped parameter that is shared by all synclets (both input and output) involved in the message processing, and `msg` is the object tree representing the DM message.

The following is an example of an input synclet.

Example:

```java
public class MotorolaV500
    implements InputMessageProcessor {
    // ------------------------------------------------ InputMessageProcessor
    public void preProcessMessage(MessageProcessingContext processingContext,
        SyncML message)
        throws Sync4jException {
        List items, validItems;
        List results = message.getSyncBody().getCommands();
        Item item;
        AbstractCommand c;
        Results r;
        Iterator i = results.iterator();
        while (i.hasNext()) {
            c = (AbstractCommand)i.next();
            if (c instanceof Results) {
                r = (Results)c;
                validItems = new ArrayList();
                items = r.getItems();
                Iterator j = items.iterator();
                while (j.hasNext()) {
                    item = (Item)j.next();
                    if (item.getSource() != null) {
                        validItems.add(item);
                    }
                }
                List oldItems = r.getItems();
                oldItems.clear();
                oldItems.addAll(validItems);
            }
        }
    }
}
The scope of the synclet is to remove all items from the incoming message that do not have a Source element. For example, the Motorola V500 phone sometimes sends erroneous <Item></Item> elements that are not allowed in SyncML. With the above code, those Items will be removed before the message is processed by the management engine.

### Configuring an Input Synclet

You configure the input synclet by telling the Pipeline Manager to insert the new synclet in the input pipeline, as shown in the following server-side JavaBean:

Example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.0" class="java.beans.XMLDecoder">
<object class="com.funambol.framework.engine.pipeline.PipelineManager">
   <void property="inputProcessors">
      <array class="com.funambol.framework.engine.pipeline.InputMessageProcessor" length="1">
         <void index="0">
            <object class="com.funambol.dm.synclet.MotorolaV500"/>
         </void>
      </array>
   </void>
   <void property="outputProcessors">
      <array class="com.funambol.framework.engine.pipeline.OutputMessageProcessor" length="0"/>
   </void>
</object>
</java>
```
**Postprocessing an Outgoing Message**

To postprocess an outgoing message, you create an output processor component and configure the Pipeline Manager accordingly.

**Creating an Output Synclet**

An output synclet is a class that implements the com.funambol.framework.engine.pipeline.OutputMessageProcessor interface. This interface defines a single method, 

```java
void postProcessMessage(MessageProcessingContext context, SyncML msg)
```

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

An example of an output synclet is the com.funambol.server.engine.RespURISynclet class. The scope of this synclet is to insert the RespURI element into the outgoing message; this element tells the client the URL to which the next message is sent. The code is as follows:

```java
public class RespURISynclet
    implements OutputMessageProcessor {
    // ---------------------------------------------------------- Constants
    public static final String PARAM_SESSION_ID = "sid";
    // --------------------------------------------- OutputMessageProcessor
    public void postProcessMessage(MessageProcessingContext processingContext,
                                    SyncML message)
        throws Sync4jException {
        Configuration config = Configuration.getConfiguration();
        String sessionId = (String)processingContext.getProperty(
            processingContext.PROPERTY_SESSIONID);
        if (sessionId == null) {
            if (log.isLoggable(Level.INFO)) {
                log.info(processingContext.PROPERTY_SESSIONID + " is null!
Synclet ignored");
            }
            return;
        }
        String serverUri =
            config.getStringValue(ConfigurationConstants.CFG_SERVER_URI);
        message.getSyncHdr().setRespURI(
            serverUri +
            '?' +
            PARAM_SESSION_ID +
            '=' +
            sessionId
        );
    }
}
```
Configuring an Output Synclet

You configure the output synclet by telling the Pipeline Manager to insert the new synclet in the output pipeline, as shown in the following server-side JavaBean (using the same configuration for the input pipeline as in the preceding section on the input synclet).

Example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.0" class="java.beans.XMLDecoder">
<object class="com.funambol.framework.engine.pipeline.PipelineManager">
  <void property="inputProcessors">
    <array class="com.funambol.framework.engine.pipeline.InputMessageProcessor"
      length="1">
      <void index="0">
        <object class="com.funambol.dm.synclet.MotorolaV500"/>
      </void>
    </array>
  </void>
  <void property="outputProcessors">
    <array class="com.funambol.framework.engine.pipeline.OutputMessageProcessor"
      length="1">
      <void index="0">
        <object class="com.funambol.server.engine.RespURISynclet"/>
      </void>
    </array>
  </void>
</object>
</java>
```
Chapter 5  Implementing Management Operations

This chapter describes extending the Funambol DM Server by developing custom management operations.

Topics

- Overview, page 42
- Creating a Processor Selector, page 43
- Creating a Management Processor, page 49
- Using Scripting Management Processors, page 53
Overview

A management operation is a sequence of management commands that the server sends to the device in order to perform a higher level task. For example, in the case of the client settings provisioning, a “setBrowserSettings” operation is translated into a sequence of Get/Replace commands that will result in setting the phone browser configuration.

The Management Engine is the core component that handles device management sessions and operations. It implements the protocol requirements but delegates to external management processors the accomplishment of the management actions to perform during a management session.

When a client starts a new management session, the Management Engine selects the Management Processor to use by the means of the Manager Selector. The selector will make its choice based on the content of the first device information sent by the client in the SyncML DM PKG 1.

The architecture of the management engine is shown below.

The orange colored components are those that can be customized. In this section we will focus on management selector and processor developments.
Creating a Processor Selector

The Processor Selector component is represented by an implementation of the interface `com.funambol.framework.server.dm.ProcessorSelector`, which is defined by the following method:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ManagementProcessor.getProcessor(DeviceDMSSession dms, DevInfo devInfo)</td>
<td>Called by the Management Engine at the beginning of a management session to determine the manager that must handle the session.</td>
</tr>
</tbody>
</table>

Two simple Processor Selector implementations are provided out of the box: `DeviceIdProcessorSelector`, which associates management processors to sets of device identifiers; and `OperationProcessorSelector`, which associates a management processor to the operation stored in the device management state.

**DeviceIdProcessorSelector**

This is represented by the class `com.funambol.server.dm.DeviceIdProcessorSelector`. It is configured with an array of associations `<regexp>-<management_processor>`, where `<regexp>` is a regular expression interpreted by the JDK package `java.util.regex` and used to match the device id; `<management_processor>` is a server side bean configuration path. If no device id matches any of the given regexp, a default processor will be returned; otherwise the first match is returned.

An example of a `DeviceIdProcessorSelector` configuration file is as follows:

Example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.2" class="java.beans.XMLDecoder">
<object class="com.funambol.server.dm.DeviceIdProcessorSelector">
<void property="defaultProcessor">
<string>com/funambol/server/dm/manager/DefaultProcessor.xml</string>
</void>
<void property="patterns">
<array class="com.funambol.framework.tools.PatternPair" length="2">
<void index="0">
<object class="com.funambol.framework.tools.PatternPair">
<void property="manager">
<string>com/funambol/server/dm/manager/DeviceDetailProcessor.xml</string>
</void>
<void property="pattern">
<string>IMEI:333*</string>
</void>
</object>
</void>
<void index="1">
<object class="com.funambol.framework.tools.PatternPair">
<void property="manager">
<string>com/funambol/server/dm/manager/OperationProcessor.xml</string>
</void>
<void property="pattern">
<string>IMEI:333*</string>
</void>
</object>
</void>
</array>
</void>
</object>
</java>
```
The `DeviceIdProcessorSelector` class is a good example of how to develop a selector. A simplified version of the source code is shown below.

Example:

```java
public class DeviceIdProcessorSelector
    implements ProcessorSelector, LazyInitBean {
    // --------------------------------------------------------- Private data
    private Pattern[] regexps;
    // --------------------------------------------------------- Properties
    /**
    * The pattern-pairs used to match device ids
    */
    private PatternPair[] patterns;
    /**
    * Sets patterns
    *
    * @param patterns the new patterns
    */
    public void setPatterns(PatternPair[] patterns) {
        this.patterns = patterns;
    }
    /**
    * Gets patterns
    *
    * @return the patterns property
    */
    public PatternPair[] getPatterns() {
        return patterns;
    }
    /**
    * The default processor server bean name
    */
    private String defaultProcessor;
    /**
    * Sets defaultProcessor
    *
    * @param defaultProcessor the new default processor name
    */
    public void setDefaultProcessor(String defaultProcessor) {
```
this.defaultProcessor = defaultProcessor;
}
/**
 * Returns defaultProcessor
 * @return defaultProcessor property value
 */
public String getDefaultProcessor() {
    return this.defaultProcessor;
}

// ---------------------------------------------------- ProcessorSelector
/**
 * @param sessionId the management session id: ignored
 * @param devInfo the device info
 *
 * @see ProcessorSelector
 */
public ManagementProcessor getProcessor(DeviceDMState dms, DevInfo devInfo) {
    String beanName = defaultProcessor;
    String device = devInfo.getDevId();
    Matcher m;
    for (int i=0; i<regexps.length; ++i) {
        m = regexps[i].matcher(device);
        if (m.matches()) {
            beanName = patterns[i].processor;
            break;
        }
    }
    ManagementProcessor processor = null;
    try {
        processor = (ManagementProcessor)
            Configuration.getConfiguration().getBeanInstanceByName(beanName);
    } catch (Exception e) {
        // error handling
    }
    return processor;
}

// --------------------------------------------------------- LazyInitBean
/**
 * During bean initialization all the given regular expressions are compiled.
 * If there are errors, a BeanInitializationException is thrown.
 *
 * @throws BeanInitializationException if one of the patterns cannot be compiled
 */
public void init() throws BeanInitializationException {
    if ((patterns == null) || (patterns.length == 0)) {
        regexps = new Pattern[0];
        return;
    }
}
regexps = new Pattern[patterns.length];
for (int i=0; i<patterns.length; ++i) {
    try {
        regexps[i] = Pattern.compile(patterns[i].pattern);
    } catch (Exception e) {
        if (log.isLoggable(Level.SEVERE)) {
            log.severe("Error compiling pattern "+ patterns[i].pattern
                      + ": "
                      + e.getMessage());
        }
        throw new BeanInitializationException("Error compiling pattern "+ patterns[i].pattern + "!", e);
    }
}
}

Most of the methods are getters and setters for the properties that we want to be able to
configure through the XML document seen above. The core of the class is the getSelector() method, which tries a match between the device id (extracted from the DevInfo object) and the
given regular expressions. If a match is found, the corresponding processor name is
considered a server side JavaBean and then is instantiated by the means of the Configuration
object.

Note also the use of lazy initialization: init() is called after the instance is created and all
properties have been set. It gives DeviceIdProcessorSelector the opportunity to compile
the regular expressions specified in the configuration file as strings.

OperationProcessorSelector

This is implemented by the com.funambol.server.engine.dm.OperationProcessorSelector class. This selector is used as a dispatcher – it reads the operation to perform on a
given device and uses that operation to build the name of the processor that should process
the request. This processor is configured with an error processor name (to be used if the device
state is 'E' - error) and a default processor name (to be used when no other processors could be selected).

The management processor name is constructed as follows:

prefix + operation + suffix
Where prefix and suffix are configurable values and operation is read from the device management state.

The algorithm used to select the correct management state for the device currently under management is represented in the above figure. If the device management session is in an error state, the error processor is selected. If instead, the device management session is in any other state, the operation field, if specified, is used to select the right processor. Otherwise, the default selector will be used.

An example of an OperationProcessorSelector configuration file is as follows:

Example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.2" class="java.beans.XMLDecoder">
<object class="com.funambol.server.dm.OperationProcessorSelector">
  <void property="defaultProcessor">
    <string>com/funambol/server/dm/processor/DefaultProcessor.xml</string>
  </void>
  <void property="errorProcessor">
    <string>com/funambol/server/dm/processor/ErrorProcessor.xml</string>
  </void>
  <void property="namePrefix">
    <string>com/funambol/server/dm/processor/</string>
  </void>
  <void property="namePostfix">
    <string>.xml</string>
  </void>
</object>
</java>
```
**Configuring the Management Engine**

To configure the management engine in order to use a specific processor selector, set the `server.dm.selector` property in `Funambol.properties`.

Example:

```
server.dm.selector=com/funambol/server/dm/OperationProcessorSelector.xml
```
## Creating a Management Processor

A Management Processor is represented by the interface `com.funambol.framework.engine.dm.ManagementProcessor`, which defines the following methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void beginSession(String sessionId, Principal principal, int type, DevInfo info, DeviceDMState dmstate)</code></td>
<td>Called when a management session is started for the given principal. <code>sessionId</code> is the content of the <code>SessionID</code> element of the OTA DM message; <code>type</code> is the management session type (as specified in the message <code>Alert</code>); <code>info</code> is the device info of the device under management; <code>dmstate</code> is the device management state, which represents a row of the <code>fnbl_dm_state</code> table.</td>
</tr>
</tbody>
</table>
| `void endSession(int completionCode)` | Called when the management session is closed. CompletionCode can be one of the following:  
  - `STATE_COMPLETED`  
  - `STATE_ABORTED`  
  - `STATE_ERROR`  
    These constants are defined in `com.funambol.framework.engine.dm.DeviceDMState`. |
| `ManagementOperation[] getNextOperations()` | Called to retrieve the next management operations to be performed. |
| `void setOperationResults(ManagementOperationResult[] results)` | Called to set the results of a set of management operations. |
| `String getName()` | Name to uniquely identify the management processor instance (each installed management processor should have a different name in its configuration file). |
**ManagementOperation**

This class represents an action that can be performed on a client management tree, such as a Get, Replace, Exec and so on. It belongs to the `com.funambol.framework.engine.dm` package.

ManagementOperation can represent one of the following actions (for additional information, see the SyncML Device Management Representation Protocol, version 1.1.2):

- Add
- Atomic
- Copy
- Delete
- Exec
- Get
- Replace
- Sequence

To represent all possible operations, the hierarchy shown above is implemented; gray boxes are abstract classes, white boxes are concrete implementations.

`com.funambol.framework.engine.dm.ManagementOperation` is an abstract class used for abstraction purposes only.
AggregatedManagementOperation adds the following methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ManagementCommand[] getCommands()</td>
<td>Returns the contained commands.</td>
</tr>
<tr>
<td>void setCommands(ManagementCommand[] commands)</td>
<td>Sets the commands aggregation.</td>
</tr>
</tbody>
</table>

TreeManagementOperation adds the following methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map getNodes()</td>
<td>Returns the management nodes affected by the operation.</td>
</tr>
<tr>
<td>void setNodes(Map)</td>
<td>Sets the management nodes affected by the operation.</td>
</tr>
</tbody>
</table>

The Map can contain the following:

- A pair of name/value. The value can be null performing a Get or a Delete.
- A pair of name/TreeNode. A TreeNode has the following properties: name; format; type; value.

ManagementOperationResult

When a management action is performed on the client, result status and possibly data are returned. This information is wrapped into a com.funambol.framework.engine.dm.ManagementOperationResult object. It represents a combination of the following SyncML DM commands:

- Results
- Status

For details, refer to the SyncML Device Management Representation Protocol (see “Open Mobile Alliance Documentation, Version 1.1.2” on page 72).

com.funambol.framework.engine.dm.ManagementOperation has the following methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int getStatusCode()</td>
<td>Returns the corresponding status for the operation.</td>
</tr>
<tr>
<td>void setStatusCode(int statusCode)</td>
<td>Sets the operation status code.</td>
</tr>
<tr>
<td>Map getNodes()</td>
<td>Returns the nodes property.</td>
</tr>
<tr>
<td>void setNodes(Map nodes)</td>
<td>Sets the nodes property.</td>
</tr>
</tbody>
</table>
Note that the nodes properties may contain results if the ManagementOperationStatus regards a Get, or a set of nodes if it relates to a status of any command with items in it. For example, if the following status is returned for a Delete command:

Example:

```
<Status>
  <CmdID>2</CmdID>
  <MsgRef>1</MsgRef>
  <CmdRef>1</CmdRef>
  <Cmd>Delete</Cmd>
  <TargetRef>./DevInfo/Lang</TargetRef>
  <Data>405</Data>
</Status>
```

The corresponding ManagementOperationResults would have:

Example:

```
statusCode: 405
command: Delete
nodes: {./DevInfo/Lang}
```
Using Scripting Management Processors

The ability to implement your own management processing is a powerful tool for the development of device management applications. However, the fact that you have to develop a java class, compile it and embed it into the server may be somehow complicated. For this reason, the Funambol DM Server provides out-of-the-box a concrete implementation of a management processor that makes things much simpler.

The class `com.funambol.server.engine.dm.ScriptManagementProcessor` is a concrete implementation of the `ManagementProcessor` interface that uses a scripting language to carry on the required management logic. The scripting language supported by the current Funambol DM Server implementation is BeanShell (see “Other Resources” on page 73).

The interpreter is created once in the ManagementProcessor's `beginSession()` method and is initialized setting the scripting variable listed below and running the script specified in the `scriptFile` property. Scripts are located under the config path `com/funambol/server/dm/processor/bsh`.

The script specified in `scriptFile` must have five entry points: `init()`, `getNextOperations()`, `setOperationResults()`, `setGenericAlerts()`, and `endSession()`. In order to keep the interaction between `ScriptManagementProcessor` and the underlying scripting engine, input and output values are passed by variables and not as input parameters and return values.

**Scripting Variables**

The following scripting variables are set in the interpreter environment:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>processor</td>
<td>The ManagementProcessor instance reference.</td>
</tr>
<tr>
<td>principal</td>
<td>User principal to be managed.</td>
</tr>
<tr>
<td>devInfo</td>
<td>Device info of the device to be managed.</td>
</tr>
<tr>
<td>managementType</td>
<td>Value given by the device when starting the management session (such as server or client initiated management session).</td>
</tr>
<tr>
<td>config</td>
<td>The configuration object used to get server side JavaBeans objects and other configuration parameters.</td>
</tr>
<tr>
<td>sessionId</td>
<td>The current session identifier.</td>
</tr>
<tr>
<td>log</td>
<td>The FunambolLogger to use for logging.</td>
</tr>
<tr>
<td>dmstate</td>
<td>The DeviceDMState object associated with the session.</td>
</tr>
</tbody>
</table>
The following scripting variables are input/output variables that the management script and the management processor share:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>operations</td>
<td>OUT</td>
<td>ManagementOperation[] to be returned to the device management engine.</td>
</tr>
<tr>
<td>results</td>
<td>IN</td>
<td>ManagementOperationResult[] returned by the device management engine.</td>
</tr>
</tbody>
</table>

**Scripting Processor Example**

A good example of how to develop a management processor script is represented by the GetDeviceDetails script used to retrieve from a device some of its ./DevDetail parameters. This script implements the GetDeviceDetails management operation. The code is the following.

**Example:**

```java
import java.util.*;
import java.util.logging.*;
import com.funambol.framework.core.*;
import com.funambol.framework.engine.dm.*;

// -------------------------------------------------------------------------
final String DEVDETAIL_FWV = "./DevDetail/FwV";
final String DEVDETAIL_SWV = "./DevDetail/SwV";
final String DEVDETAIL_HWV = "./DevDetail/HwV";

// -------------------------------------------------------------------------
String buildDetailString(HashMap nodes) {
    StringBuffer xml = new StringBuffer();
    xml.append("<DevDetail>\n        <DevId>").append(devInfo.devId).append("</DevId>\n        <Man>").append(devInfo.man).append("</Man>\n        <Mod>").append(devInfo.mod).append("</Mod>\n        <Lang>").append(devInfo.lang).append("</Lang>\n        <FwV>").append(valueOf(nodes{DEVDETAIL_FWV})).append("</FwV>\n        <SwV>").append(valueOf(nodes{DEVDETAIL_SWV})).append("</SwV>\n        <HwV>").append(valueOf(nodes{DEVDETAIL_HWV})).append("</HwV>\n    </DevDetail>\n    return xml.toString();
}

// -------------------------------------------------------------------------
void init() {
    log.info("Management script initialization");
    cont = true;
}
```
void getNextOperations() {
    log.info("getNextOperations!");
    nodes = new HashMap();
    nodes.put(DEVDETAIL_FWV, "");
    nodes.put(DEVDETAIL_HWV, "");
    nodes.put(DEVDETAIL_SWV, "");
    o = new GetManagementOperation();
    o.nodes = nodes;
    if (cont) {
        operations = new ManagementOperation[] { o };  
        cont = false;
    } else {  
        operations = new ManagementOperation[0];
    }
}

void setOperationResults() {
    log.info("setOperationResults!");
    String fwv = null;
    String swv = null;
    String hwv = null;
    details = "";
    for (result: results) {
        if (log.isLoggable(Level.FINE)) {
            log.fine("status code: " + result.statusCode);
            log.fine("for: " + result.nodes);
            log.fine("command: " + result.command);
        }
        if (Get.COMMAND_NAME.equals(result.command)) {
            if (result.statusCode != 200) {
                if (log.isLoggable(Level.INFO)) {
                    log.info("Received error code "+ result.statusCode "+ " for nodes " + result.nodes);
                    log.info("Device: " + devInfo.devId + "; operation: GetDeviceDetail; sessionId: " + sessionId);
                }
            } else {
                details = buildDetailString(result.nodes);
            }
        }
    }
    if (status.length() == 0) {
        status = "0:";  // it means ok!
    }
}
```java
if (log.isLoggable(Level.FINE)) {
    log.fine("Device detail: " + details);
}

// Reset the operation so that GetDeviceDetails won't be erroneously called again
//
dmstate.operation = null;
dmstate.state = DeviceDMState.STATE_COMPLETED;
}

void endSession(int code) {
    log.info("endSession with code: " + (char)code);
}

// -------------------------------------------------------------------------
log.info("Global script!");
importCommands("com/funambol/server/dm/processor/bsh/command");
cont = true;
status = new StringBuffer();
```

The script looks very similar to a Java class without `main()`. As said before, when the interpreter is first created, this script is evaluated; this makes the global part of the script (the statements in the outermost scope) to be interpreted and executed. In the case above, the utility commands are imported and some variables are initialized. Plus, remember that ScriptManagementProcessor will have set the scripting variables in the table above.

Before calling any other method of the script, ScripManagementProcessor calls `init()`; this is a good point where to put initialization code. Note that in our example there is only the initialization of the `cont` variable. It is done again for two reasons:

- The global `cont=true` is done so that the variable `cont` will be created in the interpreter global scope (it is like a declaration)
- The `init()` method could be called more then once (but always once per management session) – for example when initialization is retried in the case of a failed authentication.

The management processor asks the script processors which commands to send to the client calling `getNextOperation()`. In our case, we have to send a Get command for the three parameters `/DevDetail/FwV`, `/DevDetail/SwV`, `/DevDetail/HwV`; therefore, the needed parameters are set in the nodes map and a new GetManagementOperation is created. Note that a simplified syntax is used to set the operation's nodes.

The management operation so created is returned to the management processor as an array of ManagementOperation objects setting the output variable `operations`. `cont` is then set to false to remember that the Get command has already been returned to the processor.

The processor will then processes all the returned commands and will collect the results from the device. Those results are translated to ManagementOperationsResult objects and `setOperationResults()` is called. Again, note that the ManagementOperationResult[] array is passed to the script in the input scripting variable `results`. 
setOperationResults() processes the status and returned data and builds the device detail string calling buildDetailString().

Since a DM session is intended to be an iterative process, the processor will ask again for the next operations to send calling again getNextOperation(). This time cont is false and an empty array is returned. This tells the management processor that no more management commands are required.

At the end of the process the management processor will call endSession().
Chapter 6  External Applications

This chapter describes how the Funambol DM Server interacts with external applications.

Topics

- External Application Interfaces, page 60
- Implementing the Sender Component, page 62
External Application Interfaces

The Funambol DM Server interacts with the external world in many ways. Any extensible module (such as synclets, message processors, and so on) can interface with external software as needed. A good example is the Notification Sender as described in “Sender Interface” on page 62. This type of integration is mainly used when going from the Funambol DM Server to an external system.

To make Funambol DM Server accessible to other networked applications, an EJB layer is provided. This can be accessed by any EJB client, on the same host or remotely. This EJB layer can be easily wrapped by a web services layer using a WS toolkit such as Apache Axis (for details, see “Other Resources” on page 73).

The EJB Layer

The management server can be accessed through the ManagementBean stateless session EJB. This is implemented by the following classes:

- com.funambol.server.engine.dm.ejb.ManagementBean
- com.funambol.server.engine.dm.ejb.ManagementLocal
- com.funambol.server.engine.dm.ejb.ManagementRemote
- com.funambol.server.engine.dm.ejb.ManagementHomeLocal
- com.funambol.server.engine.dm.ejb.ManagementHomeRemote
The following interface is exposed by the ManagementBean:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
</table>
| bootstrap(int messageType, int transportType, String deviceUri, String phoneNumber, String username, String password, String info) | This method is used to create a new device into the Funambol DM Server system. It may result in sending a bootstrap message to the physical device or not, depending by the configured sender.  
  • messageType - the type of the bootstrap message as define in NotificationConstants  
  • transportType - the type of the transport as define in NotificationConstants  
  • deviceUri - the device id  
  • phoneNumber - the phone number of the device  
  • username - the user name with which the device will do the next device management  
  • password - the password with which the device will do the next device management  
  • info - application specific info |
| sendNotification(int messageType, int transportType, String phoneNumber, String operation, String info) | Sends a notification message to the device with the given phoneNumber  
  • messageType – the type of the notification message as define in NotificationConstants  
  • transportType – the type of the transport as define in NotificationConstants  
  • phoneNumber – the phone number  
  • operation – the management operation to be performed  
  • info – application specific detail information |
| executeManagementOperation(String phoneNumber, String operation, String info) | Executes the management sequence identified by the given operation name.  
  • phoneNumber – the phone number  
  • operation – the management operation name  
  • info – application specific detail information |
Implementing the Sender Component

This section describes how to implement the Sender component.

Sender Interface

This component dispatches notification and bootstrap messages. The behavior of a Sender is defined by the interface `com.funambol.framework.notification.Sender`. You can create multiple implementations of the interface, each specific to a particular protocol or delivery mechanism.

The following interface is exposed by the ManagementBean:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
</table>
| void sendMessage(int messageType, String phoneNumber, byte[][] message, String info) | This method is used to send a single binary message (notification or bootstrap).  
  - messageType – the type of message. It can be one of the following:  
    - MESSAGE_TYPE_NOTIFICATION_GENERIC  
    - MESSAGE_TYPE_BOOTSTRAP_WAP  
    - MESSAGE_TYPE_BOOTSTRAP_PLAIN  
  - phoneNumber – the recipient of the messages.  
  - message – the message to be sent.  
  - info – application-specific detail information. |
| void sendMessages(int messageType, String[] phoneNumbers, byte[][][] messages, String info) | This method is used to send multi-binary notification messages.  
  - messageType – the type of message. In the current version, it can only be  
    - MESSAGE_TYPE_NOTIFICATION_GENERIC.  
  - phoneNumbers – the recipients of the messages.  
  - messages – the messages to be sent.  
  - info – application-specific detail information. |
public void sendMessages(int messageType,
        String contentType,
        String[] macs,
        int[] authMethods,
        String[] phoneNumbers,
        byte[][][] messages,
        String info)
throws NotificationException;

This method is used to send multi-binary bootstrap messages

- messageType – the type of message. It can be one of the following:
  - MESSAGE_TYPE_BOOTSTRAP_WAP
  - MESSAGE_TYPE_BOOTSTRAP_PLAIN

- contentType – the content type of the message. In the current version, it is the value:
  - application/vnd.syncml.dm+wbxml

- macs[] – the macs values of the messages.

- authMethod[] – the authentication method to use: Valid values:
  - AUTH_METHOD_NETWPIN
  - AUTH_METHOD_USERPIN
  - AUTH_METHOD_USERNETWPIN

- phoneNumbers – the recipients of the messages.

- messages – the messages to be sent.

- info[] – application-specific detail information.

All constants used in the Sender interface are defined in
Sender Component Configuration

You configure the sender components using the following server-side JavaBeans:

- **NotificationSender** – `com/funambol/server/engine/dm/NotificationSender.xml`
- **BootStrapSender** – `com/funambol/server/engine/dm/BootstrapSender.xml`

A sample configuration is shown below:

Example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.2" class="java.beans.XMLDecoder">
  <object class="com.funambol.dm.engine.MySender">
    <!-- any MySender specific property -->
  </object>
</java>
```

**NOTE:** Since Funambol does not implement a service to deliver SMS messages (i.e., shipping notification/bootstrap messages), a custom component must be implemented. You can configure the Funambol DM Server to use such a component by simply changing the appropriate server JavaBean.
Chapter 7

**Bootstrapping Devices**

Topics

- *Bootstrap Overview, page 66*
- *WAP Provisioning Profile, page 69*
- *Plain Profile, page 70*
Bootstrap Overview

The Open Mobile Alliance defines `bootstrap` as "the process of provisioning the DM client to a state where it is able to initiate a management session to a new DM server." The DM client requires the following information to perform a management session:

- Account information
- Connectivity information

The DM Server uses the information contained in the `dm-server/config/com/funambol/server/engine/dm/SyncMLDMbootstrapMessage.xml` file to create the bootstrap message. The file is an XML serialization of the `com.funambol.framework.core.dm.ddf.SyncMLDM` class and contains the information to send to the device. In addition, the following information is created or set by the server at the time of the bootstrap:

<table>
<thead>
<tr>
<th>Information</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name of the account</td>
<td>Funambol</td>
</tr>
<tr>
<td>Address to be used in the management sessions</td>
<td><a href="http://myserver:myport/funambol/dm">http://myserver:myport/funambol/dm</a></td>
</tr>
<tr>
<td>Server identifier</td>
<td>funambol</td>
</tr>
<tr>
<td>Server password</td>
<td>srvpwd</td>
</tr>
<tr>
<td>Server nonce</td>
<td>cy8rVlFOVi5NJmokM0xcYw==</td>
</tr>
<tr>
<td>Client username</td>
<td>funambol</td>
</tr>
<tr>
<td>Client password</td>
<td>funambol</td>
</tr>
<tr>
<td>Client nonce</td>
<td>PyQzSUM4PDwjJjw/WzlKLA==</td>
</tr>
<tr>
<td>Preferred authentication type</td>
<td>syncml:auth-md5</td>
</tr>
</tbody>
</table>
The following is an example of the `SyncMLDMbootstrapMessage.xml` file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.2_01" class="java.beans.XMLDecoder">
<object class="com.funambol.framework.core.dm.ddf.SyncMLDM">
  <void property="dmAcc">
    <object class="com.funambol.framework.core.dm.ddf.DMAcc">
      <void property="DMAccounts">
        <object class="org.apache.commons.collections.map.ListOrderedMap">
          <void method="put">
            <!-- This name will be replaced from the server -->
            <string>AccountName</string>
            <object class="com.funambol.framework.core.dm.ddf.DMAccount">
              <void property="conRef">
                <string>test</string>
              </void>
            </object>
          </void>
        </object>
      </void>
    </object>
  </void>
  <void property="con">
    <object class="com.funambol.framework.core.dm.ddf.ConNode">
      <void method="put">
        <string>test</string>
        <object class="com.funambol.framework.core.dm.ddf.Connection">
          <void class="com.funambol.framework.core.dm.ddf.PX">
            <void property="address">
              <string>address</string>
            </void>
            <void property="addressType">
              <string>5</string>
            </void>
          </void>
        </object>
      </void>
    </object>
  </void>
</void>
```

<void property="nap">
<object class="com.funambol.framework.core.dm.ddf.NAP">
  <void property="address">
    <string>address</string>
  </void>
  <void property="addressType">
    <string>5</string>
  </void>
  <void property="auths">
    <object class="org.apache.commons.collections.map.ListOrderedMap">
      <void method="put">
        <string>PAP</string>
        <object class="com.funambol.framework.core.dm.ddf.Auth">
          <void property="id">
            <string>username</string>
          </void>
          <void property="secret">
            <string>password</string>
          </void>
        </object>
      </void>
    </object>
  </void>
  <void property="bearer">
    <string>11</string>
  </void>
</object>
</void>
</object>
</object>
</void>
</object>
</void>
</void>
</object>
</void>
</void>
</object>
</object>
</object>
</java>
WAP Provisioning Profile

The content of the bootstrap message is based on the OMA Provisioning Content Specification using the registered document ‘w7’ that specifies how the APPLICATION characteristic should be used to bootstrap a DM device. For an example, see “Bootstrap XML Message Examples” on page 74.

An external application can start a bootstrap process with this profile using the bootstrap() method of the external application API, with the following message type:


For details on the external application API, see “External Application Interfaces” on page 60.

The process of generating and sending a bootstrap message by the server consists of the following steps:

1. Update the database with the information of the device to be bootstrapped.
2. Read the `SyncMLbootstrapMessage.xml` file and create a new SyncMLDM object.
3. Update the object created at the previous step with the server information.
4. Convert the SyncMLDM object in a WBXML WAP message and send it using the configured BootStrap sender.
Plain Profile

The content of the bootstrap message is a SyncML DM message containing the nodes to be added to the device. For an example, see “Bootstrap XML Message Examples” on page 74.

An external application can start a bootstrap process with this profile using the `bootstrap()` method of the external application API, with the following message type:

```
```

For details on the external application API, see “External Application Interfaces” on page 60.

The process of generating and sending a bootstrap message by the server consists of the following steps:

1. Update the database with the information of the device to be bootstrapped.
2. Read the `SyncMLbootstrapMessage.xml` file and create a new SyncMLDM object.
3. Update the object created at the previous step with the server information.
4. Create a SyncML DM 1.1.2 message with the bootstrap information.
5. Convert the message in WBXML and send it using the configured BootStrap sender.
Appendix A  Appendix

Topics

- Resources, page 72
- Bootstrap XML Message Examples, page 74
- WAP Headers for Bootstrap Message, page 80
- Notification Message Using WAP Push, page 82
Resources

This section lists resources you may find useful.

Related Documentation

This section lists documentation resources you may find useful.

Funambol DM Server Documentation

The Funambol DM Server documentation set includes the following documents:

- *Funambol DM Server Administration Guide*: Read this guide for server installation instructions.

Open Mobile Alliance Documentation, Version 1.1.2

- SyncML Device Management Protocol
- SyncML Device Management Tree and Description
- SyncML Device Management Bootstrap
- SyncML Notification Initiated Session
- SyncML Device Management Security
- SyncML Device Management Standardized Objects
- SyncML Device Management Representation Protocol
- SyncML Data Sync Protocol
- SyncML Representation Protocol (version 1.1)
Other Resources

For information on BeanShell, visit http://www.beanshell.org.


For information on Apache Axis, visit http://ws.apache.org/axis/.
Bootstrap XML Message Examples

WAP Profile

The following example is for demonstrative purposes only.

Example

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wap-provisioningdoc version="1.0">
  <characteristic type="APPLICATION">
    <parm name="APPID" value="w7"/>
    <parm name="NAME" value="1"/>
    <parm name="PROVIDER-ID" value="funambol"/>
    <parm name="TO-NAPID" value="test"/>
    <characteristic type="APPADDR">
      <parm name="ADDR" value="http://localhost/funambol/dm"/>
      <characteristic type="PORT">
        <parm name="PORTNBR" value="8080"/>
      </characteristic>
    </characteristic>
    <characteristic type="APPAUTH">
      <parm name="AAUTHLEVEL" value="APPSRV"/>
      <parm name="AAUTHNAME" value="funambol"/>
      <parm name="AAUTHSECRET" value="PkFcRTVRLzc1dUd1Qns1JA=="/>
      <parm name="AAUTHDATA" value="Q3lHUkRRKUEhMlQnSH88Tg=="/>
    </characteristic>
    <characteristic type="APPAUTH">
      <parm name="AAUTHLEVEL" value="CLIENT"/>
      <parm name="AAUTHNAME" value="1"/>
      <parm name="AAUTHSECRET" value="1"/>
      <parm name="AAUTHDATA" value="TzpNeD5FWSctMGRPCdOMg=="/>
    </characteristic>
  </characteristic>
  <characteristic type="NAPDEF">
    <parm name="NAPID" value="test"/>
    <parm name="BEARER" value="11"/>
    <parm name="NAP-ADDRESS" value="address"/>
    <parm name="NAP-ADDRTYPE" value="5"/>
    <characteristic type="NAPAUTHINFO">
      <parm name="AUTHTYPE" value="PAP"/>
      <parm name="AUTHNAME" value="username"/>
      <parm name="AUTHSECRET" value="password"/>
    </characteristic>
  </characteristic>
  <characteristic type="PXLOGICAL">
    <characteristic type="PXPHYSICAL">
      <parm name="PXADDR" value="address"/>
      <parm name="PXADDRTYPE" value="5"/>
      <characteristic type="PORT">
        <parm name="PORTNBR" value="80"/>
      </characteristic>
      <characteristic type="PXAUTHINFO">
        <parm name="PXAUTH-TYPE" value="auth-method"/>
        <parm name="PXAUTH-ID" value="username"/>
        <parm name="PXAUTH-PW" value="password"/>
      </characteristic>
    </characteristic>
  </characteristic>
</wap-provisioningdoc>
```
Plain Profile

The following example is for demonstrative purposes only.

Example

```xml
<?xml version="1.0" encoding="UTF-8"?>
<SyncML>
  <SyncHdr>
    <VerDTD>1.1</VerDTD>
    <VerProto>DM/1.1</VerProto>
    <SessionID>0</SessionID>
    <MsgID>0</MsgID>
    <Target>
      <LocURI>1</LocURI>
    </Target>
    <Source>
      <LocURI>http://localhost:8080/funambol/dm</LocURI>
    </Source>
  </SyncHdr>
  <SyncBody>
    <Add>
      <CmdID>1</CmdID>
      <Item>
        <Target>
          <LocURI>./SyncML/DMAcc/funambol</LocURI>
        </Target>
        <Meta>
          <Format xmlns='syncml:metinf'>node</Format>
        </Meta>
      </Item>
      <Item>
        <Target>
          <LocURI>./SyncML/DMAcc/funambol/Addr</LocURI>
        </Target>
        <Data>http://localhost/funambol/dm</Data>
      </Item>
      <Item>
        <Target>
          <LocURI>./SyncML/DMAcc/funambol/AddrType</LocURI>
        </Target>
        <Data>1</Data>
      </Item>
      <Item>
        <Target>
          <LocURI>./SyncML/DMAcc/funambol/PortNbr</LocURI>
        </Target>
        <Data>8080</Data>
      </Item>
    </Add>
  </SyncBody>
</SyncML>
```
<Meta>
  <Format xmlns='syncml:metinf'>node</Format>
</Meta>
<Item>
  <Target>
    <LocURI>./SyncML/Con/test/NAP</LocURI>
  </Target>
  <Meta>
    <Format xmlns='syncml:metinf'>node</Format>
  </Meta>
</Item>
<Item>
  <Target>
    <LocURI>./SyncML/Con/test/NAP/Bearer</LocURI>
  </Target>
  <Data>11</Data>
</Item>
<Item>
  <Target>
    <LocURI>./SyncML/Con/test/NAP/AddrType</LocURI>
  </Target>
  <Data>5</Data>
</Item>
<Item>
  <Target>
    <LocURI>./SyncML/Con/test/NAP/Addr</LocURI>
  </Target>
  <Data>address</Data>
</Item>
<Item>
  <Target>
    <LocURI>./SyncML/Con/test/NAP/Auth</LocURI>
  </Target>
  <Meta>
    <Format xmlns='syncml:metinf'>node</Format>
  </Meta>
</Item>
<Item>
  <Target>
    <LocURI>./SyncML/Con/test/NAP/Auth/PAP</LocURI>
  </Target>
  <Meta>
    <Format xmlns='syncml:metinf'>node</Format>
  </Meta>
</Item>
<Item>
  <Target>
    <LocURI>./SyncML/Con/test/NAP/Auth/PAP/Id</LocURI>
  </Target>
  <Data>username</Data>
</Item>
<Item>
  <Target>
    <LocURI>./SyncML/Con/test/NAP/Auth/PAP/Secret</LocURI>
  </Target>
  <Data>password</Data>
</Item>
<Item>
    <Target>
        <LocURI>./SyncML/Con/test/PX</LocURI>
    </Target>
    <Meta>
        <Format xmlns='syncml:metinf'>node</Format>
    </Meta>
    <Data></Data>
</Item>
<Item>
    <Target>
        <LocURI>./SyncML/Con/test/PX/PortNbr</LocURI>
    </Target>
    <Data>80</Data>
</Item>
<Item>
    <Target>
        <LocURI>./SyncML/Con/test/PX/AddrType</LocURI>
    </Target>
    <Data>5</Data>
</Item>
<Item>
    <Target>
        <LocURI>./SyncML/Con/test/PX/Addr</LocURI>
    </Target>
    <Data>address</Data>
</Item>
<Item>
    <Target>
        <LocURI>./SyncML/Con/test/PX/Auth</LocURI>
    </Target>
    <Meta>
        <Format xmlns='syncml:metinf'>node</Format>
    </Meta>
</Item>
<Item>
    <Target>
        <LocURI>./SyncML/Con/test/PX/Auth/auth-method</LocURI>
    </Target>
    <Meta>
        <Format xmlns='syncml:metinf'>node</Format>
    </Meta>
</Item>
<Item>
    <Target>
        <LocURI>./SyncML/Con/test/PX/Auth/auth-method/Id</LocURI>
    </Target>
    <Data>username</Data>
</Item>
<Item>
    <Target>
        <LocURI>./SyncML/Con/test/PX/Auth/auth-method/Secret</LocURI>
    </Target>
    <Data>password</Data>
</Item>
</Add>
</Final>
</SyncBody>
</SyncML>
WAP Headers for Bootstrap Message

PLAIN Profile

Header
06 05 04 0B 84 C0 02 01 06 2E C2 91 80 92 45 36 34 30 37 42 37 42 30 46 37 42
35 43 30 37 44 42 31 44 32 34 33 39 34 00 AF 87

WDP Header
06: User-Data-Header (UDHL) Length
05: UDH IE Identifier Port Number
04: UDH port number IE length
0B: Destination port (high)
84: Destination port (low)
C0: Origination port (high)
02: Origination port (low)

WSP Header
01: Transaction ID / Push ID
06: PDU Type (push)
2E: Headerslength (content type + headers)
C2: Content-type
91: SEC
80: Auth. method = 0
92: MAC
MAC Value: 45 36 34 30 37 42 37 42 30 46 37 42 46 38 39 37 43 32 45 44 37 43
41 45 46 35 31 35 43 30 37 44 42 31 44 32 34 33 39 34 00
AF: X-WAP-Application-ID
87: x-wap-application:syncml.dm

WAP Profile

Header
06 05 04 0B 84 C0 02 01 06 2E B6 91 80 92 36 34 42 33 45 46 37 35 45 38 39 42
32 35 41 33 44 35 36 45 37 30 30 32 33 30 33
46 41 31 39 41 36 34 38 41 33 34 42 37 00 AF 82
**WDP Header**

06: User-Data-Header (UDHL) Length  
05: UDH IE Identifier Port Number  
04: UDH port number IE length  
0B: Destination port (high)  
84: Destination port (low)  
C0: Origination port (high)  
02: Origination port (low)  

**WSP Header**

01: Transaction ID / Push ID  
06: PDU Type (push)  
2E: Headerslength (content type + headers)  
B6: Content-type  
91: SEC  
80: Auth. method = 0  
92: MAC  

MAC Value: 36 34 42 33 45 46 37 35 45 38 39 42 32 35 41 33 44 35 36 45 37 30 30 32 33 30 33 46 41 31 39 41 36 34 38 41 33 34 42 37 00  
AF: X-WAP-Application-ID  
82: x-wap-application:wml.ua
Notification Message Using WAP Push

Notification Message Created by the DM Server

The following examples are valid notification messages with 'sync4j' as server id.

Example 1
81 6B 41 D3 1C 99 84 52 65 73 70 F8 C1 CC 32 C5 02 C0 00 00 00 00 04 06 73 79 6E 63 34 6A

Example 2
47 E3 FC D5 C5 09 81 36 AF 01 4F E7 9C 1C AD F1 02 C0 00 00 00 00 03 06 73 79 6E 63 34 6A

Complete SMS (WDP + WSP + Notification Message)

Example 1
06 05 04 0B 84 C0 02 01 06 03 C4 AF 87 81 6B 41 D3 1C 99 84 52 65 73 70 F8 C1 CC 32 C5 02 C0 00 00 00 00 04 06 73 79 6E 63 34 6A

Example 2
06 05 04 0B 84 C0 02 01 06 03 C4 AF 87 47 E3 FC D5 C5 09 81 36 AF 01 4F E7 9C 1C AD F1 02 C0 00 00 00 00 03 06 73 79 6E 63 34 6A

Example 3
06 05 04 0B 84 C0 02 01 06 03 C4 AF 87 A0 96 3B 47 B8 D0 B3 D7 4C 45 9F B5 44 35 98 B1 02 C0 00 00 00 00 09 06 73 79 6E 63 34 6A

Explanation of Example 3:

WDP Header:
06: User-Data-Header (UDHL) Length = 6 bytes
05: UDH IE Identifier Port Number
04: UDH port number IE length
0B: Destination port (high)
84: Destination port (low)
C0: Origination port (high)
02: Origination port (low)
WSP Header:
01: Transaction ID / Push ID
06: PDU Type(push)
03: Headers length (content type + headers)
C4: Content type
AF: X-WAP-Application-ID
87: Id for urn: x-wap-application:syncml.dm

Digest:
A0 96 3B 47 B8 D0 B3 D7 4C 45 9F B5 44 35 98 B1

Notification Message:
02 C0 00 00 00 00 09 06 73 79 6E 63 34 6A