This deliverable lists the prerequisite software tools and describes the set-up of the development environment for Third Party SOCIETIES Service developers. It also provides a set of technical tutorials and explanations of the main Third Party APIs so that a developer will be able to have a quick understanding of their semantics and the suggested usage patterns to exploit the SOCIETIES Framework functionalities in their Third Party Service implementations.
This document contains material, which is the copyright of certain SOCIETIES consortium parties, and may not be reproduced or copied without permission.

All SOCIETIES consortium parties have agreed to full publication of this document.

The commercial use of any information contained in this document may require a license from the proprietor of that information.

Neither the SOCIETIES consortium as a whole, nor a certain party of the SOCIETIES consortium warrant that the information contained in this document is capable of use, or that use of the information is free from risk, and accept no liability for loss or damage suffered by any person using this information.

Impressum

[Full project title] Self Orchestrating Community Ambient Intelligence Spaces
[Short project title] SOCIETIES
[Number and title of work-package] WP6: Third Party Services
[Document title] D6.1 Service Creation Environment, Documentation and User Guide (initial prototype)
[Editor: Name, company] Guido Spadotto, Soluta.Net
[Work-package leader: Name, company] Stavros Xynogalas, Amitec

Copyright notice

© 2012 Participants in project SOCIETIES
Executive summary

In this deliverable, the steps required to get a development environment specifically tailored to implement SOCIETIES Third Party Services is described, together with some examples and tutorials regarding how to use the functionalities exposed by SOCIETIES API.

On the first part of D6.1, we start by the description of how to get and install the Virgo Application Server, required to run the SOCIETIES Framework Components within a node, and how to debug the components running in a SOCIETIES node by exploiting the Remote Debugging features provided by the IDE of choice (Eclipse). We then detail the installation procedure of an XMPP Server (OpenFire) which is required for communication purposes across SOCIETIES Nodes. We finally explain how to quickly obtain a typical project layout for SOCIETIES in the Eclipse integrated development environment, how to use Maven to define SOCIETIES dependencies, and how to configure the declarative settings of Spring Contexts in order to express which services are required (“osgi:reference” elements) and which services are provided (“osgi:service” elements) by the project.

On the second part of the deliverable we focus on code examples and tutorials related to the usage of the main External APIs so that third party developers are helped in their learning curve.

We describe the Messaging and Publish/Subscribe features provided by the Communication Framework Components, which allow for the integration of external components in an implementation language agnostic way, by means of XML Messages. The Activity Management component, that supports the sending and notification of events from feeds defined at the CIS and at the CSS level, and the CIS Management APIs, allowing a Third Party Service to execute maintenance operations on CIS entities (create, update, subscribe, etc.) are described next. We then follow on describing the Communication / XMPP component APIs, regarding the definition of the wire protocol for XML Messages being exchanged across SOCIETIES Nodes, and the Context Broker APIs, dealing with the persistency and notification of changes of Context Entities and their related Context Attributes. Our description of the exposed APIs comprehends also the Device Management component (for receiving and handling event from physical devices), the Identity component (which maps the multiple identities a given user might have, depending on his/her role, to the relevant Context Entities and Attributes) and the Personalisation component (that implements the logic to achieve proactive personalisation and advanced User Intent and Preference Management features).

The objective of this deliverable is to provide the necessary open source IDE (Integrated Development Environment) embedded with documentation / user guide on how to create a 3rd party service for both the OSGI and the Android platform, although at the time of writing the developer documentation has not been integrated fully within the proposed IDE of Eclipse. This will be reported on in the final version of this deliverable, D6.2.
# List of authors

<table>
<thead>
<tr>
<th>Company</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>Soluta.Net</td>
<td>Guido Spadotto</td>
</tr>
<tr>
<td>Amitec</td>
<td>Stavros Xynogalas, Yiorgos Bouloudis</td>
</tr>
<tr>
<td>Sintef</td>
<td>Thomas Vilarinho</td>
</tr>
<tr>
<td>Intel</td>
<td>Alec Leckey</td>
</tr>
<tr>
<td>Heriot-Watt University</td>
<td>Eliza Papadopoulou</td>
</tr>
<tr>
<td>Trialog</td>
<td>Rafik Said Mansour</td>
</tr>
<tr>
<td>ICCS</td>
<td>Nikos Kalatzis</td>
</tr>
<tr>
<td>PTIN</td>
<td>João Miguel Gonçalves</td>
</tr>
</tbody>
</table>
# Table of Contents

Executive summary .................................................................................................................. 3  
List of authors ............................................................................................................................ 4  
1 Introduction ............................................................................................................................. 7  
2 Development Environment and Toolkit Setup ....................................................................... 8  
  2.1 Prerequisite downloads and versions ............................................................................... 8  
  2.2 Installing Virgo Tomcat Server ....................................................................................... 8  
    2.2.1 Prerequisites ............................................................................................................ 9  
    2.2.2 Installing ............................................................................................................... 9  
    2.2.3 Environment Variables Set Up ............................................................................... 9  
    2.2.4 Starting, stopping and debugging Virgo .................................................................. 9  
    2.2.5 Running Virgo inside the Eclipse IDE .................................................................... 13  
  2.2.6 SOCIETIES Platform Properties Configuration ...................................................... 14  
  2.3 Installing OpenFire ......................................................................................................... 16  
    2.3.1 Prerequisites ......................................................................................................... 16  
    2.3.2 OpenFire Configuration ......................................................................................... 16  
    2.3.3 PSI Configuration ................................................................................................. 18  
  2.4 OSGi/SpringDM Bundle development with Maven ........................................................... 18  
    2.4.1 Creating new Maven-Spring-OSGI project within Eclipse using maven plug-in .... 18  
    2.4.2 Spring Configuration for Registering a Bean as an OSGI Service ...................... 20  
    2.4.3 Creating a “Service Consuming” Project .............................................................. 21  
  2.5 SOCIETIES Platform Distribution ............................................................................... 23  
  3 3rd Party Service APIs ......................................................................................................... 24  
    3.1 Description ................................................................................................................... 24  
    3.2 Activity ....................................................................................................................... 24  
    3.3 CIS Management ........................................................................................................... 26  
      3.3.1 API Definition ....................................................................................................... 26  
      3.3.2 Technical Configuration ...................................................................................... 27  
    3.4 Communication / XMPP ............................................................................................... 28  
      3.4.1 Technical Architecture ......................................................................................... 28  
      3.4.2 Schema Definition for Message Beans ................................................................. 29  
      3.4.3 Communication Group Manager ........................................................................ 31  
      3.4.4 From The Client Side - Calling Methods and Passing Parameters ..................... 35  
      3.4.5 Eventing ................................................................................................................ 36  
      3.4.6 Event Bean Persistency Format ............................................................................ 40  
    3.5 Context .......................................................................................................................... 41  
      3.5.1 Model .................................................................................................................... 41  
      3.5.2 Broker .................................................................................................................. 42  
    3.6 Device Management ...................................................................................................... 45  
      3.6.1 API Definition ....................................................................................................... 45  
      3.6.2 Examples ............................................................................................................... 47  
      3.6.3 Technical Configuration ...................................................................................... 48  
    3.7 Identity .......................................................................................................................... 49  
      3.7.1 Interface Identity .................................................................................................... 50  
      3.7.2 Interface IdentityContextMapper ......................................................................... 50  
      3.7.3 Interface IdentityManager .................................................................................... 50  
      3.7.4 Interface INetworkNode ...................................................................................... 51  
      3.7.5 Class Requestor ..................................................................................................... 51  
      3.7.6 Class RequestorCis ............................................................................................... 52  
      3.7.7 Class RequestorService ....................................................................................... 52  
      3.7.8 Enum IdentityType ............................................................................................... 53  
    3.8 Personalisation ................................................................................................................. 53  
      3.8.1 Explicit Personalisation ......................................................................................... 53  
      3.8.2 Proactive Personalisation ....................................................................................... 54  

© SOCIETIES consortium 2012
3.9 Registering a Third Party Service on a Cloud/Rich Node .......................................................... 55
4 Conclusions and Future Work........................................................................................................... 57
1 Introduction

As the development of the SOCIETIES core components goes on, we need to be able to validate that providing new SOCIETIES Third Party Services is indeed as much as possible a feasible and stream-lined process.

In fact, the availability of the SOCIETIES framework alone, without a varied set of Third Party services exploiting its functionalities, would not be a satisfying demonstration of the innovations achieved within the project.

A prerequisite of this validation is the availability of a comprehensive technical documentation regarding the best development practices to apply and the concepts to be aware of when implementing a Third Party service: this deliverable aims specifically at providing this information.

Keeping the perspective of the developer in mind, we will describe the procedures to set up the development environment and the SOCIETIES external APIs whose semantics each Third Party developer will have to understand, in order for him to successfully integrate his Service with the SOCIETIES Framework.

Even though the APIs documented in these deliverable will evolve, we feel that it is important to have the feedback of developers that are not part of the consortium as soon as possible, as they will provide their specific points of view and requirements. Throughout all of this document we will provide tutorials and code examples whenever possible to help Third Party developers in their development process and – hopefully – get their feedback about how easy or hard the development of a third party service is, what framework concepts are not clear, which functionalities are missing or would be nice to have, and so on.

The SOCIETIES platform allows for numerous deployment models for 3rd party services (see Figure 1) for providing flexible points of interoperability with the platform APIs and technical platforms. The effective result of this architecture is to not impose any technical constraints in terms of deployment and platform dependencies, thus freeing up 3rd party services to develop their implementations as they wish or to easily integrate existing services.

![Figure 1 SOCIETIES Deployment models](image)
2 Development Environment and Toolkit Setup

In this section of this deliverable we are going to explain how to set up the development environment that Third Party developers will have to use to implement and test their services. Although not all of the components will be strictly required to develop a Third Party service, we feel that it will be beneficial for Third Party developers to have all the tools available in order for them to navigate the SOCIETIES platform code. The following set up instructions will allow to have a configured development environment that supports this option.

2.1 Prerequisite downloads and versions

Rich/Cloud Node:
- Java Sun/Oracle Java - JDK 6 Update 29 (not Java 7)
- Eclipse 3.7 Indigo Enterprise Edition which provides by default EGit (1.1.0+) to interact with GIT code repositories, m2e(1.0+) that integrates with the Maven build system, and Mylyn (teamwork tool). Use the Eclipse Marketplace to install the Spring IDE plug-in (for spring configuration support).
- Maven, Version 3.0.3+
- Virgo Tomcat Server 3.0.2 as the Application Server (includes Spring v3.0.5.RELEASE)
- XMPP
  - XMPP server - Openfire 3.7.0
  - Java Client - Smack v3.2.1
  - JS Client - Strophe.js (Bosh impl) v1.0.2
- HTML5:
  - jQuery - v1.6.4

Light Node:
- Android SDK r12 - v2.2
- XMPP
  - Android Client - aSmack (v2010-12-11 (Smack v3.1))
- HTML5:
  - jQueryMobile - 1.0beta3

2.2 Installing Virgo Tomcat Server

The Virgo Web Server from EclipseRT is a completely module-based Java application server that is designed to run enterprise Java applications and Spring-powered applications with a high degree of flexibility and reliability. It offers a simple yet comprehensive platform to develop, deploy, and service enterprise Java applications. The project web site is www.eclipse.org/virgo/. Its installation and use is described in detail because it represents one of the most important software components that Third Party developers will interact with developing their services.
2.2.1 Prerequisites

The Virgo Tomcat Server, or VTS for short, requires Java SE 6 or later to be installed, and is available as a zip file from http://www.eclipse.org/virgo/download/

2.2.2 Installing

To install Virgo Tomcat Server on Linux, unzip the distribution package to the desired installation directory. For example, to install into /opt:

prompt$ unzip virgo-web-server-3.0.1.RELEASE.zip -d /opt

This creates a directory called virgo-web-server-3.0.1.RELEASE under /opt.

Virgo Tomcat Server requires write access to the installation directory, in this case /opt/virgo-web-server-3.0.1.RELEASE. Typically this means it must be run as the user that installed it, or the installation directory’s ownership must be changed.

To install the Virgo Tomcat Server on Windows, unzip the distribution package to the desired installation directory. Note that both Windows and Java have some issues with long file names and file paths, so we recommend installing to the root directory of your chosen drive.

2.2.3 Environment Variables Set Up

Two environment variables have to be set up for VTS to run reliably, namely JAVA_HOME and VIRGO_HOME.

On Linux, edit the .profile file in your home directory to add the VIRGO_HOME and JAVA_HOME environment variables. For example, if you installed into /opt:

    export VIRGO_HOME=/opt/virgo-web-server-3.0.1.RELEASE/
    export JAVA_HOME=/user/java/jdk1.6.0_27
    export PATH=$JAVA_HOME/bin:$PATH

To verify the setting of JAVA_HOME, issue the command $JAVA_HOME/bin/java -version from a new terminal window and ensure that the command completes successfully and reports a Java version 1.6.x (denoting Java 6) or greater.

On Windows, from the Start menu, open the Control Panel and double-click on “System”, select the “Advanced” tab, select “Environment Variables”. Next, click the “New” button in the “System Variables” section (see Figure 2).

This will display the “New System Variable” window. Enter VIRGO_HOME as the “Variable name” and the installation directory as the “Variable value”. Click OK (see Figure 3).

Repeat the procedure for setting the JAVA_HOME environment variable. To verify the setting of JAVA_HOME, issue the command “%JAVA_HOME%\bin\java -version from a new command prompt and ensure that the command completes successfully and reports a Java version 1.6.x (denoting Java 6) or greater.

2.2.4 Starting, stopping and debugging Virgo

To start Virgo Tomcat Server run the startup.sh (Linux) or startup.bat (Windows) script. For both platforms, the script is located in the VIRGO_HOME/bin directory.

Once Virgo Tomcat Server has started, the console will display a log message similar to the one shown below, along with other status messages:

The preceding message indicates that you can start using VTS.
To start Virgo in “clean mode”, add the –clean option as a suffix of the script invocation. When you start Virgo Tomcat Server in clean mode, the startup script removes the VIRGO_HOME/work directory (and hence all running applications) as well as all trace, log and dump files. It leaves the VIRGO_HOME/repository and VIRGO_HOME/pickup directories untouched, which means that any applications previously hot deployed will be automatically reinstalled.

Finally, to start Virgo in debug mode, launch the startup script with the following parameters:

```
startup.[bat|sh] -debug 8001 -suspend
```

this will start the Virgo debug agent, listening on port 8001, and suspend the Virgo boot procedure until a debugger attaches to the agent.

This feature will be invaluable when debugging a Third Party service that has been deployed into or that interacts with a SOCIETIES node running on Virgo.

Once the server is started, you might access its web-based console using the following url: 
[http://localhost:8080/admin](http://localhost:8080/admin) (input “admin” as ID, and “springsource” as password).

With the default configuration, server logs are available at VIRGO_HOME\serviceability\logs

---

**Figure 2 Environment Variables Settings for Windows**

![Environment Variables Settings for Windows](image)

Finally, to start Virgo in debug mode, launch the startup script with the following parameters:

```
startup.[bat|sh] -debug 8001 -suspend
```

this will start the Virgo debug agent, listening on port 8001, and suspend the Virgo boot procedure until a debugger attaches to the agent.

This feature will be invaluable when debugging a Third Party service that has been deployed into or that interacts with a SOCIETIES node running on Virgo.

Once the server is started, you might access its web-based console using the following url: 
[http://localhost:8080/admin](http://localhost:8080/admin) (input “admin” as ID, and “springsource” as password).

With the default configuration, server logs are available at VIRGO_HOME\serviceability\logs
2.2.4.1 Debugging Bundles running on a remote Virgo App Server from within Eclipse

Build the project using maven and deploy the jar to the virgo pickup directory

Start virgo server in debug mode as follows.

VIRGO_HOME\bin\startup.bat –debug 8000

Or

VIRGO_HOME\bin\startup.bat –debug -suspend

Note: The –suspend option will suspend Virgo’s Virtual Machine until a debugger is attached to the specified port (i.e. it will pause the boot of the Virgo server until a debug session is initiated from Eclipse)

From Eclipse navigate to run → debug configuration (Figure 4):

This will open the Debug Configuration window (Figure 5). Select "Remote Java Application" icon on left side, Right click and say "New" (give your preferred name of your config – in this case “Virgo_External_Debug”). After clicking on New, Eclipse will create a new Remote Java Application configuration for your selected project.

Provide the host name and port on which your process is listening for debugger connections. Check the "Allow termination of remote VM" check box if you would like to close the remote Java application from within Eclipse.
The debug configuration is ready for use, as long as you remember to start the target application (in our case, Virgo) before connecting the Eclipse remote debugger to it. You can now debug by going to "Debug Configurations" selecting your project in "Remote Java Application" and clicking on "DEBUG".

![Debug Configurations](image1)

**Figure 5 Remote Java Application Debug Configuration Settings**

If you want to perform line-by-line executions of the debugged code, you will have to tell the debugger which locations to search for the matching code, Figure 6 shows how to define the code lookup folders:

![Code Lookup Path](image2)

**Figure 6 Source Lookup Path**

Make sure you have built the code recently and have set breakpoints at the desired locations in the code before starting the debug process, and enjoy your remote Virgo debugging sessions.
2.2.5 Running Virgo inside the Eclipse IDE

Running and debugging Virgo instances from within the development environment is a handy feature for developers that will save them from switching back and forth across two different contexts (the IDE and the VIRGO server). To set up this feature, first install the "Eclipse Virgo Tools" package from the repository: http://download.eclipse.org/virgo/milestone/IDE. In order to do so, go to Help->Install New Software. There you can add the repository with the ADD button and you will find the packet once the repository is added.

Figure 7 Virgo Server Runtime Definition

To configure Virgo server inside eclipse goto Windows->Preference->Server option from the left panel and then choose runtime environment (see Figure 7). On the right side you may see an option to add servers, click “add” and type the server name “Virgo”, then click “next” to type the server name, server installation directory and JRE (see Figure 8). Click “Ok” to finish.
To add the server and the view console go to Windows->show view->servers. This will add the “server perspective” view, right click press new->server then filter for virgo and select the server already configured. You should see something similar to Figure 9.

Finally, use the start and debug buttons (white “play” icon over green circle and bug icon respectively) to run the server in required mode and view the server console window for various logs. Note: You can drag a project and drop it directly on the virgo server entry in the “Servers” tab for direct deployment from within Eclipse (instead of dropping a jar inside the VIRGO_HOME/pickup directory which has a directory watcher thread scanning it for changes in its content).

Double click Virgo Server to view various attributes and properties as shown in Figure 10.

2.2.6 SOCIETIES Platform Properties Configuration

For any custom configuration, the folder VIRGO_HOME/config has to be used. To allow certain bundles that utilise a database (currently Derby) a properties file named org.societies.platform.properties is required in the SVIRGO_HOME/config directory (N.B. The VIRGO_HOME environment variable must be created to allow this properties to work).

Its contents for Windows are:

```
jdbc.driverClassName=org.apache.derby.jdbc.EmbeddedDriver
jdbc.url=jdbc\:derby\:\$\{VIRGO_HOME\}\Databases\serviceregdb;create\=true
jdbc.password=app
jdbc.username=app
hibernate.dialect=org.hibernate.dialect.DerbyDialect
hibernate.connection.url=jdbc\:derby\:\target\database\message;create\=true
hibernate.dialect=org.hibernate.dialect.DerbyDialect
```
Whilst for Linux there are some minor changes. These are the entries that have to be altered:

```sql
jdbc.url=jdbc:derby:${VIRGO_HOME}/Databases/serviceregdb;create=true
hibernate.connection.url=jdbc:derby:target/database/message;create=true
```

all other entries remain untouched.
2.3 Installing OpenFire

OpenFire is the XMPP server of choice for the SOCIETIES Communication Layer. If a Third Party service developer wants to communicate with a SOCIETIES node, it will have to exchange XMPP messages with a given payload.

2.3.1 Prerequisites

Download OpenFire from [http://www.igniterealtime.org/projects/openfire/](http://www.igniterealtime.org/projects/openfire/) and accept the default values during the installation phase (further details are available in the [official installation guide](#)). Download and install an XMPP client for testing purposes. We suggest using PSI.

2.3.2 OpenFire Configuration

Start the OpenFire Server and select "Launch Admin" to launch the admin website. The first time you do this you will need to make some config choices. You can select defaults for most screens, but please make the following edits:

- "Server Setup": Domain name defaults to your machine name. Change it to "societies.local" (see Figure 11)
Figure 11 OpenFire Server Settings

- "Database Config": Choose "Embedded Database" (see Figure 12)

Figure 12 OpenFire Database Settings

When complete, log in to the admin website and perform the following changes:

- Under "Users/Groups", select "Create New User" and create a new user account, e.g. alec
- Under "Server" -> "Server Settings" -> "External Components". In the "Service Enabled" box, click the "Enabled" checkbox and enter a "Default shared secret", e.g. "password" and click "Save Settings"
- In the "Allowed to Connect" box, create a new subdomain (which represents a new External Component) and provide the values as in Figure 13

Figure 13 OpenFire Subdomain Settings for XMPP External Components

Click "Add Component" to finish adding the external component. You do not have to provide any implementation file: this setting simply represents the connection informations of an external component that will connect later on (ie the XCManager component).
Finally, edit your local “hosts” file to ensure that “societies.local” maps to 127.0.0.1. Confirm this by pinging societies.local from a command prompt.

### 2.3.3 PSI Configuration

PSI is a simple IM client. We want to point it at our Openfire installation for testing purposes.

- Under General -> Account Setup add our login details
- Account: under "Account” tab, add the user account we created earlier. The full Jabber ID will be e.g. alec@societies.local
- Connection: under "Connection” tab, add the server running Openfire, Host: localhost

With Openfire started, select your account in PSI and set your status to "Online". This ensures your server is configured correctly.

### 2.4 OSGi/SpringDM Bundle development with Maven

The SOCIETIES framework relies on SpringDM bundles being deployed into the VIRGO App Server running in instances of Rich and Cloud nodes. This section will provide a primer for Third Party Service developers that want to provide the bundles that expose their services using the Maven build system together with the Spring Tooling plugin.

The source code supporting this tutorial is available at [https://github.com/societies/Societies-Seed/tree/master/OSGI_Spring_Tutorial](https://github.com/societies/Societies-Seed/tree/master/OSGI_Spring_Tutorial)

There is a Maven archetype that eases the creation of SpringDM bundle project:

```
mvn archetype:create -DarchetypeGroupId=org.springframework.osgi
-DarchetypeArtifactId=spring-osgi-bundle-archetype
-DarchetypeVersion=<spring-dm version>
-DgroupId=<your-project-groupId>
-DartifactId=<your-project-artifactId>
-Dversion=<your-project-version>
```

To create a new project open a console and navigate to the directory where you want to create project and simply type the following:

```
mvn archetype:create
-DarchetypeGroupId=org.springframework.osgi
-DarchetypeArtifactId=spring-osgi-bundle-archetype
-DarchetypeVersion=1.1.0
-DgroupId=org.foo
-DartifactId=org.foo.my-springdm-bundle
-Dversion=0.1
```

### 2.4.1 Creating new Maven-Spring-OSGI project within Eclipse using maven plug-in

In Eclipse: new ➔ maven Project
In the new Maven project dialog, select “All Catalogs” and filter for OSGI. Eclipse will search and display the list of archetypes for OSGI project. Select the one highlighted in the screen capture.

Click next, follow the screen, supply your project-specific group-id, artefact-id and package names, then finish. You will get a new Maven project whose structure is depicted in.
Delete the content of the generated pom.xml and replace it with the content of the pom.xml file available at the GIT Code Repository\(^1\), just remember to update the artifactId and groupId entries with the correct values for your project.

![Image of pom.xml file structure]

**Figure 17 bundle-context.xml location**

Edit `bundle-context.xml` file (located as in Figure 17) to define a new Spring bean (see Figure 18).

```xml
<beans version="1.0" encoding="UTF-8">
    <bean name="calSer" class="org.societies.tutorial.cal.service.impl.CalculatorImpl"/>
</beans>
```

**Figure 18 bean declarative definition**

### 2.4.2 Spring Configuration for Registering a Bean as an OSGI Service

Edit `bundle-context-osgi.xml` as per Figure 19.

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
    <osgi:service ref="calSer" interface="org.societies.tutorial.cal.service.api.ICalculator"/>
</beans>
```

**Figure 19 OSGi-related bean properties**

Note: to add any new namespace into this spring config file - right click on this page as shown in Figure 20, then select “Configure Namespaces…”.

\(^1\) [https://github.com/societies/Societies-Seed/tree/master/OSGI_Spring_Tutorial/templatePom.xml](https://github.com/societies/Societies-Seed/tree/master/OSGI_Spring_Tutorial/templatePom.xml)
Figure 20 Configuration of Spring Namespaces

The window in Figure 21 will appear, from which you can select the Spring namespaces you are interested in including in your Context Configuration:

Figure 21 Selection of Spring namespaces for XML context definitions

Once configured, build the project using `mvn install` and, if you already have the VIRGO_HOME environment variable set, your jar will be automatically copied to the VIRGO_HOME/pickup/ directory, ready for execution.

2.4.3 Creating a “Service Consuming” Project

Create the project using same approach described in Section 2.4.1 and modify your pom to import the required packages and dependencies. Create your “service consuming” class by importing the service exposed API (see Figure 22 for an example).
Figure 22 Declarative Service Dependency Injection

Importing Spring annotations simply means add the following imports:

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
```

Then add the `@Component` annotation to the class and the `@Autowired` annotation either to the property that will reference the required service or to the constructor taking the parameter of the same type of the service interface (as seen in Figure 22).

Edit the `module-context.xml` file to scan the packages which needs autowiring (Figure 23):

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  <context:component-scan base-package="org.societies.tutorial.cal.user.impl"/>
</beans>
```

Figure 23 Configuration of Packages that need to be Scanned for Component definitions

Edit the `bundle-context-osgi.xml` file to fetch the service that implements the required interface (Figure 24):

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:osgi="http://www.springframework.org/schema/osgi"
       osgi:service="calService"
       osgi:servicePID="calService">
  <osgi:reference id="calService" interface="org.societies.tutorial.cal.service.api.ICalculator"/>
</beans>
```

Figure 24 OSGi Reference definition

Build your project and start Virgo: check the log file in `VIRGO_HOME\serviceability\logs` for the log output of the example service – if you followed the tutorial code – or from your specific service.
2.5 SOCIETIES Platform Distribution

At the time of writing this deliverable, the SOCIETIES platform is distributed in the form of source code, publicly available at https://github.com/societies/SOCIETIES-Platform

In the “builder” sub-folder within the structure of the project, two Maven configuration files are made available to build the sets of components that are relevant for the light nodes and the rich/cloud nodes: android-projects.xml and spring-osgi-projects.xml, respectively.

Building the SOCIETIES platform of choice is thus as simple as invoking:

```
mvn –f [android|spring-osgi]-projects.xml clean install
```

from a command-line shell within the builder folder.

In the same folder, two Virgo plan files are available to list the required components on a Rich/Client node:

1. **societies-platform.plan** lists all components required by the platform to provide its full set of functionalities

2. **societies-3p-services.plan** lists the set of bundles that 3rd Party services might require

A more user-friendly packaging of the SOCIETIES platform distributions is being considered and will be described in D6.2, the final version of this deliverable.

Technical support for 3rd party service developers will be provided through online collaboration tools (guidelines & tutorials, bug reporting, forums, etc) available at https://redmine.ict-societies.eu/projects/sp (user registration required). See Figure 33 and Figure 34 for snapshots of the Collaboration Tool User Interface.
3 3rd Party Service APIs

3.1 Description

The SOCIETIES platform exposes its functionality to 3rd party services via a set of external APIs. This API set is composed by Java Interfaces, XML Schemas and Android Interfaces. The Java Interfaces are designed to be invoked locally from 3rd party services which are deployed within the same runtime as the SOCIETIES cloud/rich nodes. The XML schema APIs allow for remote, standards-based communication via XMPP, regardless of the implementation technology of the communicating 3rd party services, thus providing a platform agnostic API. Finally, the Android Interfaces provide a local invocation point for Android applications to utilise the features of the SOCIETIES APIs, thus reducing the impact of unnecessary remote messages from Android devices (i.e. light client node) to Cloud nodes.

3.2 Activity

The Activity component provides Third Party developers with the possibility of sending a stream of custom events at the CSS or CIS level. The APIs that have been defined so far are a subset of the Open Social Core API Server specification².

The interface org.societies.api.activity.IActivity is a simple provided interface that Service Activities instances have to implement in order for the SOCIETIES framework to be able to deal with them appropriately:

```java
public interface IActivity {
    public Long getId();
    public void setId(Long id);
    public String getVerb();
    public void setVerb(String verb);
    public String getActor();
    public void setActor(String actor);
    public String getObject();
    public void setObject(String object);
    public String getTarget();
    public void setTarget(String target);
    public String getPublished();
    public void setPublished(String published);
}
```

On the other hand, the org.societies.api.activity.IActivityFeed interface formalizes the Activity Feed management service that is provided by SOCIETIES Nodes:

```java
public interface IActivityFeed {
    /**
     * This method will parse a timeperiod and return a subset of the activities
     * in this activityfeed that is within the given timeperiod
     * @param timeperiod can be: "millisecondsinceepoch millisecondsinceepoch+n"
     * @return a List of IActivity or a empty list if the parameters are wrong or the
     * timeperiod did not match any activities
     */
    public List<IActivity> getActivities(String timePeriod);
    /**
     * This method will parse a query and a timeperiod and return a subset of the activities
     * in this activityfeed that matches the query constraints and is within the given
     * timeperiod
     * @param activity feed can be: "millisecondsinceepoch millisecondsinceepoch+n"
     * @return a List of IActivity or a empty list if the parameters are wrong or the query
     * and/or timeperiod did not match any activities
     */
    public List<IActivity> getActivities(String query, String timePeriod);
    /**
     * This method will add a activity and post it on the associated pubsub service.
     * @param (Ilink IActivity) activity, the activity that will be added.
     */
}
```

² [http://opensocial-resources.googlecode.com/svn/spec/2.0.1/Core-API-Server.xml](http://opensocial-resources.googlecode.com/svn/spec/2.0.1/Core-API-Server.xml)
public void addCisActivity(IActivity activity); /*
* This method will parse a criteria and delete the activities that match the criteria
* @param criteria which has the same definition as the query of "getActivities(String query.."
* @return (short) number of deleted activities.
*/

public int cleanupFeed(String criteria); /*
* @param IActivity activity the activity that should be deleted.
* @return boolean true if the the activity was found and deleted, false if not.
*/

public boolean deleteActivity(IActivity activity); /*
* @param List of ActivityEntry as used by the social data connector, and implemented by
* shindig.
* @return long number of entries successfully imported
*/

public long importActivityEntries(List<?> activityEntries); }

The “query” input parameter syntax (for the getActivities and cleanupFeed methods) is defined as per
OpenSocial Query Specs3 but with one addition: the Query String should be a JSON-compliant String that
is structured as follows:

```json
{
...
"filterBy" : "name",
"filterOp" : "startsWith",
"filterValue" : "John"
...
}
```

Thus it needs to contain the keys (and their corresponding values) "filterBy", "filterOp" and "filterValue"
(the last one can have a empty value given certain filterOps, see below)

Filter operators are those specified as values for the “filterOp” key. The operation to use when filtering a
collection by a field specified in 'filterBy', defaults to "contains".

Valid operation values are:

1. **contains** Returns elements where filterValue appears somewhere in the element's filterBy field value.
2. **equals** Returns elements where filterValue exactly matches the element's filterBy field value.
3. **startsWith** Returns elements where filterValue exactly matches the first N characters of the
element's filterBy field value, where N is the length of the filterValue string.
4. **present** Returns elements where the element's filterBy field value is not empty or null.
5. **isNull** is the exact opposite of "present", NOTE: this is in addition to the OpenSocial specification.

The last two filter operators do not need to have a value set for the corresponding "filterValue" key.

---

3 [http://opensocial-resources.googlecode.com/svn/spec/2.0.1/Core-API-Server.xml#Request-Parameter-FilterBy-Value](http://opensocial-resources.googlecode.com/svn/spec/2.0.1/Core-API-Server.xml#Request-Parameter-FilterBy-Value)
3.3 CIS Management

This section provides information regarding the use of the SOCIETIES CIS Manager facilities by Third Party context-aware services. The CIS Manager Cloud is the component used by the user to manage his own CISs and the CISs in which he is part of.

3.3.1 API Definition

The CIS Manager Component is delivered as a single bundle, exposing both server and client functionalities. As depicted in Figure 25, there are four main interfaces that a Third Party service developer must be aware of:

1. ICisManager: exposes the local methods (createCIS, deleteCIS, listSubscribedCIS, listOwnedCIs, ) and the remote methods (joinCIS and leaveCIS). This is the main interface that clients will use.

Figure 25 CIS Manager Main Classes

The CIS Manager Component is delivered as a single bundle, exposing both server and client functionalities. As depicted in Figure 25, there are four main interfaces that a Third Party service developer must be aware of:

1. ICisManager: exposes the local methods (createCIS, deleteCIS, listSubscribedCIS, listOwnedCIs, ) and the remote methods (joinCIS and leaveCIS). This is the main interface that clients will use.
2. ICis: exposes the methods on remote or local CISs (listMembers, getCisinfo), the implementation is different depending on the fact that the represented CIS is local or remote, but the method signatures are the same.

3. ICisOwned: exposes methods on the owned (i.e. “local”) CIS. It extends the ICis interface and adds methods which can be applied just to an owned CISs (addMember, removeMember).

4. ICisManagerCallback: this is the callback interface for remote methods. At the moment, the data is encapsulated in an object of type org.societies.api.schema.cis.community.Community

Instances of the ICis interface are created when the user joins a CIS, whilst instances of ICisOwned are created when the user creates his own CISs.

The CIS and CisManager Classes implement the IFeatureServer interface (see Section 3.4.3.1) which is required to asynchronously handle requests coming from both other peers, android clients or any other software component able to send and receive XMPP messages.

The CisManager uses the org.societies.cis.manager.CisManager.ICommMgr service provided by the CSS through dependency injection, whilst the ICisOwned instances use their own ICommMgr implementation, which is created on the fly when they are instantiated.

3.3.2 Technical Configuration

If you want to get the Cis Manager service injected in your service, you will have to add the following dependency on your project’s Maven configuration file (i.e. pom.xml):

```xml
<dependency>
  <groupId>org.societies.api.external</groupId>
  <artifactId>societies-api</artifactId>
  <version>${release.version}</version>
  <scope>provided</scope>
</dependency>
```

Once the dependency is in place, you will be able to insert a reference in Spring’s bundle-context-osgi.xml file (which, by SOCIETIES convention, holds all the OSGi-related bean settings):

```xml
<osgi:reference id="CisManagementBundle" interface="org.societies.api.cis.management.ICisManager"/>
```

Finally, the following bundles will have to be available in the Virgo instance that you will be running. Check that they are included in the Virgo plan:

```xml
<artifact type="bundle" name="org.societies.api.external.societies-api"/>
<artifact type="bundle" name="org.societies.api.internal.societies-api-internal"/>
<artifact type="bundle" name="org.societies.comm.xmpp.XCCommunicationMgr"/>
<artifact type="bundle" name="org.societies.data-source"/>
<artifact type="bundle" name="org.societies.comm.xmpp.pubsub.PubsubServiceBundle"/>
<artifact type="bundle" name="org.societies.comm.xmpp.pubsub.PubsubClientBundle"/>
<artifact type="bundle" name="org.societies.platform.servicelifecycle.serviceRegistry"/>
<artifact type="bundle" name="org.societies.platform.servicelifecycle.serviceManagement"/>
<artifact type="bundle" name="org.societies.slm.SLMCommsManager"/>
<artifact type="bundle" name="org.societies.cis.CisDirectoryClient"/>
<artifact type="bundle" name="org.societies.cis.CisManager"/>
</plan>
```

Example projects are available to understand how to use the Cis Manager functionalities both from a Web App\(^4\), and as an integration test project\(^5\).

\(^4\) [https://github.com/societies/SOCIETIES-Platform/tree/development/platform-infrastructure/service-lifecycle/platform-webapp](https://github.com/societies/SOCIETIES-Platform/tree/development/platform-infrastructure/service-lifecycle/platform-webapp)

\(^5\) © SOCIETIES consortium 2012
3.4 Communication / XMPP

The goal of the Communication Framework is to create a Wired Protocol, i.e. a documented set of XML communication stanzas. This means that our solution does not pass Java classes between Client/Server and therefore locks our clients to Java only. With this architecture, other development languages can invoke SOCIETIES node services and be part of our solution, eg. a .NET or a Python client.

Understanding how the communication framework works is a fundamental prerequisite for Third Party service developers that want to integrate their services with SOCIETIES Nodes, be it to provide the node with additional informations or to exploit the functionalities that the platform exposes externally.

3.4.1 Technical Architecture

![Figure 26 Communication Framework Architecture](https://github.com/societies/SOCIETIES-Platform/tree/development/cis-modules/cis-management/cisMgmtTester)

The "Platform Comms Gateway" in Figure 26 represents the External Component that Openfire will forward all incoming messages to for processing. The implementation of this feature is deployed as the XC Manager bundle. This bundle is dependent on a configuration file called "xc.properties" that needs to be in the $VIRGO_HOME/pickup directory.

```
#Configuration for the the XMPP Server. XCManager reads following config
host=societies.local
# Make sure xcmanager (or XCManager) is spelled the same way as in your
# Openfire configuration. It is case sensitive!
subDomain=xcmanager.societies.local
secretKey=password.societies.local
```

Having every single Service Bundle registering with the XC Manager does not scale. It is preferable that a Group Manager registers and manages access to each of the end-point services. In the example depicted in Figure 26, the "Examples Comms Manager" component will implement the remote interfaces that allow services to communicate remotely. Also, it will receive requests from remote services which it then will route to the relevant local services.
3.4.2 Schema Definition for Message Beans

As depicted in Figure 26, the communication across Communication Gateways happens by the exchange of XML documents, transported by XMPP. These XML documents must be compliant to a Schema that is shared across the communicating nodes, and that represents the service interfaces that are exposed for remote invocation. In the following sections we will describe how a developer is supposed to create such a schema so that XML message beans might be reliably created on a variety of programming languages.

We have created 3 example bundles, a Calculator service, a Fortune Cookie service and a Complex Service which are running as bundles in the container (all are available in the GIT Code Repository). The creation of the raw XML that is passed between client/server/client is abstracted within the SOCIETIES framework. You cannot call the interface of your Service directly. Instead, each developer must create an XML schema to which message beans being passed between Client and Server comply to. Each message bean stores as properties:

- the method to be called
- relevant parameters for that method

You will create one XSD schema file per service interface. The XSD file contains your XMPP namespace, methods, parameters and parameter types. We have tried to simplify the process by breaking it into 3 steps. If you are already comfortable writing the XML for your Message Bean schema directly, then go straight to step 3 (Section 3.4.2.3).

3.4.2.1 Step 1: Create a temporary bean for your interface

We are going to create a set of beans (i.e. Java Classes that comply with the javabeans naming conventions) that will be automatically introspected to generate the corresponding XML Schema. Below is the interface of the minimalistic example Calculator service that we want to expose to remote clients.

```java
public interface ICalc {
    /**Adds 2 numbers together and returns total*/
    public int Add(int a, int b);

    /**Subtracts 2 numbers and returns total */
    public int Subtract(int a, int b);
}
```

Our temporary bean needs to hold only the name of the method the user wants to call and each of the parameters. Do not worry about the return type here, the return type is held in a separate bean described later.

```java
public class CalcBean {
    public enum methodType {Add, Subtract};
    private methodType method;
    private int a;
    private int b;

    public int getA() {     return a; }
    public void setA(int a) { this.a = a; }

    public int getB() {     return b; }
    public void setB(int b) { this.b = b; }

    public methodType getMethod() {
        return method;
    }
    public void setMethod(methodType method) {
        this.method = method;
    }
}
```

Now, you must create a temporary java bean for your result. The calculator returns a simple integer but other, more complex examples are available in the GIT Code Repository.

```java
public class CalcBeanResult {
    public int result = 0;
}
```

6 See [https://github.com/societies/SOCIETIES-Platform/tree/development/platform-infrastructure/comms-frwk/ExampleServices](https://github.com/societies/SOCIETIES-Platform/tree/development/platform-infrastructure/comms-frwk/ExampleServices)
Lastly, put your datatype Classes in this project. They must be in bean format, that is Get/Set methods for any property and no parameterised constructors. This will allow them to be passed as parameters in your methods.

### 3.4.2.2 Step 2: Generate and adapt the schema for your message bean

If you are using Maven to build your services, check that your project pom.xml file contains the following plugin configuration:

```xml
<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>jaxb2-maven-plugin</artifactId>
      <version>1.3.1</version>
      <configuration>
        <includes>
          <include>**/*.java</include>
        </includes>
        <packageName>org.your.xmpp.package.exampleservice</packageName>
      </configuration>
    </plugin>
    ...
  </plugins>
</build>
```

Once you have that configuration in place, at the command line, execute:

```bash
> mvn jaxb2:schemagen
```

This will create a file called `schema1.xsd` in the `/target/generated-resources/schemagen` directory of your Maven project.

Unfortunately, the auto generated schema requires some minor editing. Ensure that the first `<xs:schema>` tag does actually contain your namespace in the “targetNamespace” and the “tns” attributes.

```xml
<xs:schema  elementFormDefault="qualified" version="1.0"
    xmlns="http://societies.org/api/schema/example/calculator"
    targetNamespace="http://societies.org/api/schema/example/calculator"
    xmlns:tns="http://societies.org/api/schema/example/calculator"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

Find your 2 beans, e.g. CalcBean and CalcBeanResult.

- Add an `<element>` node before `<complexType>`
- Move the NAME to `<element>`

**BEFORE**

```xml
<xs:complexType name="calcBean">
  <xs:sequence>
    <xs:element name="a" type="xs:int"/>
    <xs:element name="b" type="xs:int"/>
    <xs:element name="method" type="tns:methodType" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

**AFTER**

```xml
<xs:element name="calcBean">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="a" type="xs:int"/>
    </xs:sequence>
  </xs:complexType>
```

```xml
<xs:element name="calcBean">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="a" type="xs:int"/>
    </xs:sequence>
```

```xml
<xs:element name="calcBean">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="a" type="xs:int"/>
    </xs:sequence>
```
You do not need to add <element> nodes to the schema of any of your datatypes you will be passing, just the message bean and the result bean only. However, if you are passing/receiving a complex datatypes (i.e. other than strings, integers, etc) then you will need to prefix the type with the tns namespace. e.g. The FortuneCookieResult returns an object of type “cookie”, then its Type definition needs to be prefixed: type="tns:cookie".

This manual editing of the generated dschema files is an annoying shortfall that the consortium is working to resolve. You should rename the XSD Schema file to your XMPP package name of choice, eg. org.your.xmpp.package.exampleservice.xsd and copy to the /schema/resources folder in the SOCIETIES API project.

### 3.4.2.3  Step 3: Build the schema

The schema is now part of the main build for the system as it sits in the API project folder so it will include:

1. **Schema Message Beans**
2. **Schema-generated datatypes**

Your original interface class should now be refactored to accept the schema-generated datatypes as parameter types and not your original java datatypes. This is because the Group Manager component will extract the schema-generated datatypes from the message bean and will pass these as the parameters to your service interface. You will get a type mismatch if you will have not modified your interface, e.g. Original interface to FortuneCookie service is

```java
public interface IWisdown {
  public org.societies.examples.fortunecookie.datatypes.Cookie getCookie();
}
```

However, what is retrieved from the MessageBeanResult class is is

```java
public org.societies.api.schema.fortunecookie.Cookie getCookie();
```

### 3.4.3  Communication Group Manager

The Platform Comms Gateway manages access from the XMPP server to the service bundles by passing/receiving messages (see Figure 27). As already stated, we don’t want every single Service Bundle to register with the XC Manager Component. Rather, it is preferable that a Comms Group Manager registers and manages access to each of its end point services. This removes the need for developers writing custom XMPP code in their services. It is this Comms Group Manager that handles the XMPP communication and accesses the bundles. This component can operate in

- **Server mode**: it receives messages and forwards them to the end point services through their internal API
- **Client mode**: it implements the remote API of each service to send messages
3.4.3.1 Server mode

First, you need to get a reference to the CommunicationManager service and a reference to each of the services you manage by registering in the Bundle-context-osgi.xml:

```xml
<osgi:reference id="CommManagerBundle" interface="org.societies.comm.xmpp.interfaces.CommManager"/>
<osgi:reference id="CalculatorBundle" interface="org.societies.examples.calculator.ICalc"/>
<osgi:reference id="FortuneCookieBundle" interface="org.societies.examples.fortunecookie.api.IWisdom"/>
<osgi:reference id="ComplexServiceBundle" interface="org.societies.examples.complexservice.IComplexService"/>
```

Then you need to register your Comms Group Manager with the Platform Comms Gateway in your InitService() method. So ensure you specify an init-method in the Bundle-context.xml file.

```xml
<bean id="CommsServerManager"
class="org.societies.comm.examples.commsmanager.impl.CommsServer"
depends-on="CommManagerBundle, FortuneCookieBundle, CalculatorBundle, ComplexBundle"
init-method="InitService">
  <property name="commManager" ref="CommManagerBundle"/>
  <property name="fcGenerator" ref="FortuneCookieBundle"/>
  <property name="calcService" ref="CalculatorBundle"/>
  <property name="complexSvc" ref="ComplexBundle"/>
</bean>
```

Finally, the following code snippet shows how you actually register your Module Group Manager with the Platform Comms Gateway in your InitService() method

```java
public void InitService() {
    getCommManager().register(this);
}
```

To receive messages from the Platform Comms Gateway you must implement the org.societies.comm.xmpp.interfaces.IFeatureServer interface. The methods that have to be implemented are:

```java
/**Returns the list of package names of the message beans you'll be passing*/
public List<String> getJavaPackages();

/**Returns the list of namespaces for the message beans you'll be passing*/
public List<String> getXMLNamespaces();

/** Put your functionality here if there is NO return object, i.e. VOID */
public void receiveMessage(Stanza stanza, Object messageBean);

/** Put your functionality here if there IS a return object */
```
3.4.3.2 Processing incoming requests

The Comms Group Manager will sit in front of several Service Bundles so you need to check the incoming message bean to see which end point to actually call. The Comms Group Manager receives a generic object type, so you must check which bean type it is. The different schema java beans created earlier will also be dependencies for the project implementing the FeatureServer interface:

```java
// --------- CALCULATOR BUNDLE ---------
if (messageBean.getClass().equals(CalcBean.class)) {
    // call method on the Calculator service
}

// --------- FORTUNE COOKIE BUNDLE ---------
else if (messageBean.getClass().equals(FortuneCookieBean.class)) {
    // call method on the FortuneCookie Service
}
```

In our example, we are calling the Add method so our code under the IF statement for the CalcBean.class would look like:

```java
CalcBean calc = (CalcBean) messageBean;
switch (calc.getMethod()) {
    case Add:
        int a = calc.getA();
        int b = calc.getB();
        int result = calcService.Add(a, b); // CALL THE Calculator SERVICE BUNDLE
        break;
    case ...
}
```

calcService is actually the injected instance of the Calc Service Bundle that performs all the work and returns the result. Finally, the return Bean that contains the return value – what is received by the client -- needs to be instantiated. This is how to do that:

```java
CalcBeanResult calcRes = new CalcBeanResult();
calcRes.setResult(result);
return calcRes;
```

3.4.3.3 Client Mode

In client mode, the Comms Group Manager component plays the role of a proxy for several service bundles, exposing interfaces whose implementation is accessed remotely.

Your Comms Group Manager component will implement the Remote API for each of the remote services that it proxies, so it should register these Remote APIs as provided services in the Virgo container. To send and receive messages, you will also need a reference to the Platform Comms Gateway service in your Bundle-context-osgi.xml:

```xml
<osgi:service ref="CalcServiceRemote" interface="org.societies.examples.commsmanager.ICalcRemote"/>
<osgi:service ref="CalcServiceRemote" interface="org.societies.examples.commsmanager.IComplexServiceRemote"/>
<osgi:reference id="CommManagerBundle" interface="org.societies.comm.xmpp.interfaces.ICommManager"/>
```

The required Platform Comms Gateway service is injected in your component by specifying the property in the Bundle-context.xml file and you need to register your Comms Group Manager with the Platform Comms Gateway in your InitService() method. So ensure you define an init-method value in the Bundle-context.xml file:

```xml
<bean id="CalcServiceRemote"
    class="org.societies.comm.examples.commsmanager.impl.CommsClient"
    depends-on="CommManagerBundle"
    init-method="InitService">
    <property name="commManager" ref="CommManagerBundle"/>
</bean>
```
... and register your Comms Group Manager with the Platform Comms Gateway in your InitService() method implementation:

```java
public class CommsClient implements ICommCallback{
    public void InitService() {
        getCommManager().register(this);
    }
}
```

To receive the reply messages back from the Platform Comms Gateway you must implement the org.societies.comm.xmpp.interfaces.ICommCallback interface, which is similar to the IFeatureServer interface:

```java
/**
 * Returns the list of package names of the message beans you'll be passing*
 */
public List<String> getJavaPackages();

/**
 * Returns the list of namespaces for the message beans you'll be passing*
 */
public List<String> getXMLNamespaces();
```

The interface to the Comm Manager to send a message is:

```java
/**For calling methods that have return types */
public void sendIQGet(Stanza stanza, Object messageBean, ICommCallback callback)
    throws CommunicationException;

/**For calling methods that have void types */
public void sendMessage(Stanza stanza, String type, Object messageBean)
    throws CommunicationException;
```

3.4.3.4 Receiving a Callback from the Platform Gateway

You call the sendMessage() method when your end point service returns no object. The sendIQget() method is used to get a result back, but runs asynchronously and you may end up sending many queries before getting the corresponding results back. In other terms, you are not guaranteed the order of requests is the same order of responses. Therefore, you are required to map a response back to the correct request. To aid with this, your message bean is encapsulated in a Stanza object (the <IQ> xml node) that contains To and From attributes but also a unique identifier that you can use to map requests with asynchronous responses:

```xml
<iq type="get" id="7d4d2d90-2d1b-44b9-b746-7f495949850f"
to="xcmanager.societies.local"
from="xcmanger.societies.local">
    <calcBean xmlns="http://societies.org/example/calculatorservice/schema">
        <a>2</a>
        <b>3</b>
        <method>AddAsync</method>
    </calcBean>
</iq>
```

The Stanza object provides an interface to set and get these attributes:

```java
public String getId() { return id; }
public Identity getFrom() { return from; }
public Identity getTo() { return to; }
public void setId(String id) { this.id = id; }
public void setFrom(Identity from) { this.from = from; }
public void setTo(Identity to) { this.to = to; }
```

As usual with network invocations, you need to implement the ICommCallback interface – which is a Callback interface as the name implies -- to handle the reply in the receiveResult() method.

The suggested strategy is to use an object of type Map<String, ICalcRemoteCallback> for storing the unique request ID together with the callback that will handle the matching response.

The ExampleCommsManager project uses the CommsClientCallback calls to implement this suggested strategy:

```java
public class CommsClientCallback implements ICommCallback {
    //MAP TO STORE THE ALL THE CLIENT CONNECTIONS
    private final Map<String, ICalcRemoteCallback> calcClients =
        new HashMap<String, CalcRemoteCallback> ();

    //CONSTRUCTOR - ADD THE CLIENT AND UNIQUE ID TO THE HASHMAP
    public CommsClientCallback(String clientID, ICalcRemoteCallback calcClient) {
        calcClients.put(clientID, calcClient);
    }
```
The ICommCallback interface requires you to process the result in the receiveResult() method. As stated before, you receive a generic message bean as the input parameter, so you need to check its runtime type to perform a correct cast.

```java
class CalcBean {
    private int a;
    private int b;
    private MethodType method;
    // getters and setters...
}
```

```java
public class CalculatorBundle {
    private List<CalcBean> data;
    // getters and setters...
}
```

```java
public class FortuneCookieBean {
    private String fortune;
    // getters and setters...
}
```

```java
public class FortuneCookieBundle {
    private List<FortuneCookieBean> data;
    // getters and setters...
}
```

```java
public void receiveResult(Stanza returnStanza, Object msgBeanResult) {
    // CHECK WHICH END SERVICE IS SENDING US A MESSAGE
    // --------- CALCULATOR BUNDLE ---------
    if (msgBeanResult.getClass().equals(CalculatorBean.class)) {
        CalculatorBean calcResult = (CalculatorBean) msgBeanResult;
        return calcResult.getResult();
    }
    // --------- FORTUNE COOKIE BUNDLE ---------
    else if (msgBeanResult.getClass().equals(FortuneCookieBean.class)) {
        FortuneCookieBeanResult fcBeanResult = (FortuneCookieBeanResult) msgBeanResult;
        //return the fcBeanResult to the calling client
    }
    // now that we have the result, we need to get the originating client
    // (i.e. the IcommCallback instance associated with the original request)
    // and hand it the return object back. The ID of this request can be found
    // in the returnStanza.getID()
    int returnedValue = calcResult.getResult();
    ICalcRemoteCallback requestingClient = (ICalcRemoteCallback) calcClients.get(returnStanza.getId());
    requestingClient.receiveCalcResult(returnedValue);
}
```

Now that we have the result, you can implement your remote method, e.g., AddAsync(). Create your message bean, populate the required properties, put in the correct method, and call sendIQ() for methods with return types or sendMessage() for void methods on the gateway:

```java
public void AddAsync(int valA, int valB, ICalcRemoteCallback calcCallback) {...
    CalcBean calc = new CalcBean();
    calc.setA(valA);
    calc.setB(valB);
    calc.setMethod(MethodType.AddAsync);
    try {
        commManager.sendIQGet(stanza, calc, callback);
    } catch (CommunicationException e) {
        LOG.warn(e.getMessage());
    }
}
```

3.4.4 From The Client Side - Calling Methods and Passing Parameters

To call a service from a remote client, make a reference to the Comms Group Manager service instance which is implementing the Remote API of the required service. Your client class must implement the ICalcRemoteCallback interface to be able to handle the asynchronous result delivery. Then, simply call the method you wish to consume with relevant parameters. For example:

```java
public class ClientTester implements ICalcRemoteCallback {
    public void StartTest() {
        getRemoteCalculator().AddAsync(2, 3, this);
    }
}
```

Implement the receiveCalcResult() method to receive your response:

```java
public void receiveCalcResult(Object calcResult) {
    int result = (Integer)calcResult;
}
```

If the complete idea of how the communication framework works is not yet clear, we do once again suggest to download and study the examples that are available in the GIT Code Repository.

---

7 [https://github.com/societies/SOCIETIES-Platform/tree/development/platform-infrastructure/comms-frwk/ExampleServices](https://github.com/societies/SOCIETIES-Platform/tree/development/platform-infrastructure/comms-frwk/ExampleServices)
3.4.5 Eventing

The eventing and pubsub feature provided by the SOCIETIES Platform allows Third Party service developers to exploit those design patterns within their services. There are currently 2 eventing models available.

1. Internal eventing between bundles within the container using the OSGI EventAdmin. Events only available to bundles running within the same container. Supports native java objects as payloads.

2. External eventing between connecting clients (including other containers). Allows a CSS Node to create an event stream for other clients (CSS nodes) to subscribe to. Supports schema generated objects only as payload.

The Pubsub service is dependent on the communication framework. For further details about it see section 3.4. The examples in this section refer to the code available on GitHub (https://github.com/societies/SOCIETIES-Platform/tree/development/platform-infrastructure/comms-frwk/ExampleServices).

3.4.5.1 Internal Eventing (OSGi)

The internal eventing interface extends the OSGI eventing system. It is targeted at eventing between bundles running within the same Virgo container (see Figure 28). These events are not visible outside of that node, in other terms, they are not available between CSS nodes of the same CSS. As we are contained within the Virgo container, developers are free to add their own Java object types as the payload or event info.

See TestInternalEventing class for full code listing from the Git code repository.

As we are extending the OSGI EventListener, you need to add the following reference in your POM file:

```
<dependency>
  <groupId>org.osgi</groupId>
  <artifactId>org.osgi.compendium</artifactId>
  <version>4.1.0</version>
  <scope>provided</scope>
</dependency>
```

You do need a dependency injection so you should get a reference to the IEventMgr service by registering in the Bundle-context-osgi.xml and Bundle-context.xml.

In the Bundle-context-osgi.xml file, get a reference to the IEventMgr service:

```
<osgi:reference id="eventMgmtRef" interface="org.societies.api.osgi.event.IEventMgr" />
```

In the Bundle-context.xml file, ensure you set a property to receive a reference to the IEventMgr:

```
<bean id="ClientTester"
    class="org.societies.examples.testclient.ClientTester"
    depends-on="eventMgrRef">
    <property name="eventMgr" ref="eventMgrRef"/>
</bean>
```

![Figure 28 Internal Eventing](image)
In your Virgo Plan ensure that the Event Manager Bundle (`org.societies.comm.event.mgmt`) is deployed:

```xml
<artifact type="bundle" name="org.societies.comm.event.mgmt" />
```

If needed, ensure that Communications Bundles are also deployed on your Virgo plan.

```xml
<artifact type="bundle" name="org.societies.comm.CommonLibraries"/>
<artifact type="bundle" name="org.societies.comm.xmpp.XCCommunicationMgr"/>
```

### 3.4.5.1.1 Creating an Event Type

To define your custom event types, add them to the `EventTypes.java` class in the External API project, package `org.societies.api.osgi.event`:

```java
/** Used by Service Management (Service Discovery) to notify a local peer of a failed service */
public static final String ADD_SERVICE_EVENT = "org/societies/servicemgmt/service/addservice";
/** Used by Service Management (Service Discovery) to notify peers of a removed service */
public static final String REMOVED_SERVICE_EVENT = "org/societies/servicemgmt/service/removeservice";
```

### 3.4.5.1.2 Subscribing to an Event

The interface for subscribing to an internal event is:

```java
/** To subscribe a listener for events
 * @param listener instance of the EventListener
 * @param eventType String array of event types [EventTypes]
 * @param filterOption a String containing an LDAP-style filter specification [EventConstants]
 */
public void subscribeInternalEvent(EventListener listener, String[] eventTypes, String filterOption);
```

As the `EventTypes` parameter is an array, you can subscribe to multiple events at the same time. You can also set a filter option using the LDAP style filter spec, based on any of the CSSEventConstants such that the event filter string looks like:

```java
String eventFilter = "(&" +
    "(" + CSSEventConstants.EVENT_NAME + "=test_event_name)" +
    "(" + CSSEventConstants.EVENT_SOURCE + "=test_event_source)" + ")";
```

And the filtering subscription is set up as:

```java
getEventMgr().subscribeInternalEvent(this, new String[] {EventTypes.CONTEXT_EVENT}, eventFilter);
```

### 3.4.5.1.3 Publishing an Event

`InternalEvent` is the object type that is published.

```java
/** Constructor
 * @param eventType String for the type of event [EventTypes]
 * @param eventName String for the name of event
 * @param eventSource String for the source component or peer id
 * @param eventInfo object for the event info. Must implement [Serializable]
 */
public InternalEvent(String eventType, String eventName, String eventSource, Serializable eventInfo);
```

The `EventInfo` property is where developers will add their own object types. In the following example, an instance of the `TestObject` class represents the payload:

```java
//GENERATE TEST PAYLOAD
TestObject myPayload = new TestObject();

//GENERATE EVENT
InternalEvent event = new InternalEvent(EventTypes.CONTEXT_EVENT, "test_event_name", "test_event_source", myPayload);

goSubscribing().subscribeInternalEvent(this, new String[] {EventTypes.CONTEXT_EVENT}, eventFilter);
```

### 3.4.5.1.4 Receiving an Event

Developers must extend the `org.societies.api.osgi.event.EventListener` abstract class to react to the receiving of a given event type. As there may be multiple events occurring within the same class, you should check which `eventType` has raised this event by calling the `getEventType()` method of the `InternalEvent`. You can then extract your payload using the `getEventInfo()` method.

```java
public class SpringEventTest extends EventListener {
    @Override
```
public void handleInternalEvent(InternalEvent event) {
    LOG.info("***** internal event received *****");
    LOG.info("***** event name : " + event.getEventName());
    LOG.info("***** event source : " + event.getEventSource());
    LOG.info("***** event type : " + event.getEventType());
    TestObject payload = (TestObject)event.getInfo();
    LOG.info("***** event name : " + payload.getName());
}

3.4.5.2 External Eventing (XMPP PubSub)

The publish-subscribe eventing functionality added to the SOCIETIES platform provides a framework for a variety of use cases e.g. news feeds, content syndication, extended presence, geolocation, profile management, and any other service that requires event notifications (see Figure 29). This technology uses the classic "publish-subscribe" or "observer" design pattern: a person or application publishes information to a node ("topic"), and an event notification (with or without the actual content) is broadcast to all authorized subscribers. If the content is light, it should be included in the event. However, if it is large the event should simply notify that the content is now available and the service should get it when it is ready (using the communication framework). This lowers the load on the PubSubService.

![Figure 29 External Eventing flows](image_url)

Eventing Nodes are owned by an identity and may take a a hierarchical form, i.e. root and sub-nodes:

- root
  - "Comm Framework"
    - "Sending messages"
    - "Eventing and Pubsub"
  - "Android tutorials"
    - "Client UI"
    - "IPC Services"
    - ...
It is possible to create Pubsub event Nodes ("topics") from clients (CSS Nodes) that will allow other CSS Nodes to subscribe to them. Developers can add their own object types as the payload, but only if you have converted them into XSD Schema objects. This is to support multiple development environments/platforms.

Refer to the TestExternalEventing class in the GIT Code Repository for full technical details.

You do need to configure a dependency injection so you should get a reference to the PubsubClientBundle service by registering in the Bundle-context-osgi.xml and Bundle-context.xml files.

Get a reference to the PubsubClientBundle service by registering in the Bundle-context-osgi.xml file:

```xml
<osgi:reference id="PubsubClient" interface="org.societies.comm.xmpp.pubsub.PubsubClient"/>
```

Ensure you set a property to receive a reference to the PubsubClient in the Bundle-context.xml file:

```xml
<bean id="ClientTestPubSub" class="org.societies.examples.testclient.ClientTester"
     depends-on="PubsubClient">
    <property name="pubsubClient" ref="PubsubClient"/>
</bean>
```

The client methods available are divided into Owner, Publisher and Subscriber related functionality (see Sections 3.4.5.3, 3.4.5.4 and 3.4.5.5 respectively). To discover the list of available Event Nodes, you can call the discoItems() method. Leaving the node parameter null will return the listing at the root level. On the other hand, setting the node parameter to a specific topic will return the list of sub topics at that level. The Identity is the owner of this CSS Node obtained by invoking the getThisNetworkNode() method:

```java
PubsubClient pubsubClient;//INJECTED DEPENDENCY THROUGH CONSTRUCTOR OR PROPERTY
Identity pubsubID = idManager.getThisNetworkNode();
//GET A LIST OF PUBSUB TOPICS (at root level)
List<String> listTopics = pubsubClient.discoItems(pubsubID, null);
for (String s: listTopics)
    System.out.println(s);
```

To add your own object types as payload of the events your component will fire/subscribe to, you need to make them compliant to Schema objects first (see Section 3.4). Then simply add the XMPP package names like in the following snippet of code:

```java
List<String> packageList = new ArrayList<String>();
packageList.add("org.societies.api.schema.calculator");
packageList.add("org.societies.api.schema.fortunecookie");
pubsubClient.addJaxbPackages(packageList);
```

### 3.4.5.3 Owner

The Owner list of APIs allows the creation and administration of a Pubsub node.

1. **pubsubService** the owner identity of the pubsub service
2. **Node** the topic name.

```java
/**Creates a pubsub node for this identity using the provided node name ("topic") */
public void ownerCreate(Identity pubsubService, String node) throws XMPPError, CommunicationException;

/**Deletes the pubsub node for this identity and node name */
public void ownerDelete(Identity pubsubService, String node) throws XMPPError, CommunicationException;

/**Deletes all the published events for this identity on this node */
public void ownerPurgeItems(Identity pubsubService, String node) throws XMPPError, CommunicationException;

/**Returns the list of identities of users who are subscribed to this pubsub node */
public Map<Identity, SubscriptionState> ownerGetSubscriptions(Identity pubsubService, String node) throws XMPPError, CommunicationException;

public Map<Identity, Affiliation> ownerGetAffiliations(Identity pubsubService, String node) throws XMPPError, CommunicationException;

public void ownerSetSubscriptions(Identity pubsubService, String node, Map<Identity, SubscriptionState> subscriptions) throws XMPPError, CommunicationException;

public void ownerSetAffiliations(Identity pubsubService, String node, Map<Identity, Affiliation> affiliations) throws XMPPError, CommunicationException;
```

### 3.4.5.4 Publisher

The Publisher list of APIs allows for publishing/removing content on a SOCIETIES node.
1. **pubsubService** the owner/id of the pubsub service Node represents the topic name

2. **ItemID** a unique identifier for that event. If you re-publish an event using the same ID you will overwrite the previously published event details

3. **item** an EventBean containing your schema datatype. NOT the datatype directly, but a MessageBean/ResultBean or EventBean containing the datatype

```java
/**Publishes an event for this identity, on this node of unique itemID and content of item */
public String publisherPublish(Identity pubsubService, String node, String itemId, object item)
throws XMPPError, CommunicationException;

/**Remove an event for this node of provided itemID */
public void publisherDelete(Identity pubsubService, String node, String itemId) throws XMPPError,
CommunicationException;
```

### 3.4.5.5 Subscriber

The Subscriber list of API's allows the user to subscribe and retrieve info from the pubsub node.

1. **Subscriber** is the actual callback object interface you are required to implement to react to notifications. The only method to implement is `pubsubEvent()`

2. **pubsubService** is the owner/id of the pubsub service

3. **node** is the topic name

4. **itemId** is a unique identifier for that event. If you re-publish an event using the same ID you will overwrite the previously published event details

5. **item** represents the event details expressed as an XML element object

```java
/**Subscribe to a pubsub node and provide the subscriber callback object */
public String subscriberSubscribe(Identity pubsubService, String node, Subscriber subscriber)
throws XMPPError, CommunicationException;

public void subscriberUnsubscribe(Identity pubsubService, String node, Identity subscriber, String subId)
throws XMPPError, CommunicationException;

public List<Element> subscriberRetrieveLast(Identity pubsubService, String node, String subId)
throws XMPPError, CommunicationException;

public List<Element> subscriberRetrieveSpecific(Identity pubsubService, String node, String subId,
List<String> itemIdList) throws XMPPError, CommunicationException;
```

Subscribing to an event requires you to implement the Subscriber interface (refer to the GIT repository for all details):

```java
public class PubsubTest implements Subscriber {
    //SUBSCRIBE
    PubSubManager.subscriberSubscribe(pubsubID, "Fortune_Cookies", this);

    @Override
    public void pubsubEvent(Identity pubsubService, String node, String itemId, Object payload) {
        if (item.getClass().equals(FortuneCookieBeanResult.class)) {
            FortuneCookieBeanResult info = (FortuneCookieBeanResult)item;
            Cookie cookie = info.getCookie();
            LOG.info("### Wisdom: " + cookie.getValue());
        }
    }
}
```

### 3.4.6 Event Bean Persistency Format

The format of the event bean is not obvious from the descriptions and needs some additional explanations. Since the event bean has to be described in the relevant XMPP namespace in order to be transported, it must be

- an existing message bean already being used in remote method invocations
or

- must be added to the XMPP namespace Schema file.

The format of the event bean can be of any complexity but must follow the following format:

```xml
<xs:element name="cssEvent">
    <xs:complexType>
        <xs:sequence>
        ...
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

### 3.5 Context

The Context Management (CM) component block acts as an intermediate layer between context-aware services and the sources of context information. An outline of the CM components can be found in D5.2.

#### 3.5.1 Model

The context model will be used in order to represent all the necessary information that describes the situation of a CSS entity or a group of entities that form a CIS. In more detail, the model consists of the following main classes:

- **CtxModelObject**
  
  It is the base abstract class of all context model objects. It is extended by the following classes:

- **CtxEntity**
  
  The CtxEntity class represents the notion of a context entity which corresponds to an object of the physical or conceptual world. The type of a CtxEntity defines the nature of the modelled entity. For example an entity could be a “person”, a “device”, or a “service”.

- **CtxAttribute**
  
  The CtxAttribute class describes a CtxEntity’s static and dynamic properties such as the “name”, “age”, or “location”. Multiple CtxAttributes can be assigned to a CtxEntity, while the value of each CtxAttribute can be set and retrieved using the appropriate setter and getter method. The following CtxAttributeValueTypes are supported:

  - STRING: Text value.
  - INTEGER: Integer value.
  - DOUBLE: Double-precision floating point numeric value.
  - BINARY: Binary value (byte[]).

- **CtxAssociation**
  
  The CtxAssociation is used in order to describe relations among different CtxEntities. We define two types of associations; Undirected associations that are peer relations (e.g. “friends”, “schoolmates”, “fellowPassengers”) and directed associations that are non-peer relations (“locatedIn”, “owns”, “uses”).

- **CommunityMemberContextEntity**
It is a subclass of the `CtxEntity` class and a generalisation of the two classes below.

- **IndividualContextEntity**
  It is used in order to represent individuals (CSS owners). More specifically, an IndividualCtxEntity may belong to zero or more CISs, simultaneously. The individual members of a pervasive community do not need to be human beings (i.e. of type “person”). They can also be organisations, smart space infrastructures, autonomous or semi-autonomous agents, etc. It inherits all attributes and operations of the `CtxEntity` class and extends them by methods related to the management of communities the individual belongs to.

- **CommunityContextEntity**
  This class is used to represent community context entities. A CommunityCtxEntity corresponds to a pervasive community (CIS) and is associated with a set of CommunityMemberCtxEntity objects representing its members, i.e. individuals and/or other communities. The CtxAttribute class is used in order to describe the community context attributes of a CIS. Common context characteristics shared among community members are described by the `CtxBond` class.

### 3.5.2 Broker

The Context Broker component manages access to current, past and future context data. The past context refers to the data stored in the context history database. The future context information is provided on the fly based on context prediction methods. The Context Broker also supports distributed context queries; it is a gateway to context data and decides whether the local DB, a remote DB or the Context Inference Management need to be contacted to retrieve the requested context data. The Context Broker exposes two interfaces, namely `org.societies.api.context.broker.ICtxBroker` and `org.societies.api.internal.context.broker.ICtxBroker`. The latter is used by platform services, therefore it does not perform any access control when handling context data.

Some of the methods that the internal Context Broker interface provides allow the creation, update, retrieval, and removal of context model objects. The following section demonstrates what are the necessary interfaces and classes that need to be imported and how to use them.

#### 3.5.2.1 Creating and updating context model objects

```java
import org.societies.api.internal.context.broker.ICtxBroker;
import org.societies.api.context.model.CtxAttribute;
import org.societies.api.context.model.CtxAttributeIdentifier;
import org.societies.api.context.model.CtxEntity;
import org.societies.api.context.model.CtxEntityIdentifier;
import org.societies.api.context.model.CtxIdentifier;
import org.societies.api.context.model.CtxModelObject;

/** The Internal Context Broker service reference. */
@Autowired
private ICtxBroker internalCtxBroker;

// create an Entity that represents a device
CtxEntity deviceEntity = this.internalCtxBroker.createEntity("device").get();

// get the context identifier of the created entity
CtxEntityIdentifier deviceEntityId = .getId();

// create an attribute to model the name of the device entity
```

```java
// create an attribute to model the name of the device entity
```
Deliverable D6.1

Deliverable D6.1

© SOCIETIES consortium 2012 Page 43 of (58)

```
CtxAttribute deviceNameAttr = 
this.internalCtxBroker.createAttribute(deviceEntityId, "name").get();

// assign a String value to the attribute
deviceNameAttr.setStringValue("device1234");
deviceNameAttr.setValueType(CtxAttributeValueType.STRING);

// update the attribute in the Context DB
deviceNameAttr = (CtxAttribute)
this.internalCtxBroker.update(deviceNameAttr).get();

// create an attribute to model the operating voltage of the device
CtxAttribute deviceVoltAttr = 
this.internalCtxBroker.createAttribute(deviceEntityId, "voltage").get();

// assign an integer value
deviceVoltAttr.setIntegerValue(new Integer(220));
deviceVoltAttr.setValueType(CtxAttributeValueType.INTEGER);
deviceVoltAttr.setValueMetric("volts");

// update the attribute in the Context DB
deviceVoltAttr = (CtxAttribute)
this.internalCtxBroker.update(deviceVoltAttr).get();

// create an attribute to model the temperature of the device
CtxAttribute deviceTempAttr = 
this.internalCtxBroker.createAttribute(deviceEntityId, "temperature").get();

// assign a double value
deviceTempAttr.setDoubleValue(36.6);
deviceTempAttr.setValueType(CtxAttributeValueType.DOUBLE);
deviceVoltAttr.setValueMetric("celsius");

// update the attribute in the Context DB
deviceTempAttr = (CtxAttribute)
this.internalCtxBroker.update(deviceTempAttr).get();

// create an attribute with a Binary value
CtxAttribute deviceBinAttr = 
this.internalCtxBroker.createAttribute(deviceEntityId, "serializableData").get();

// this is a mock Serializable class
MockBlobClass blob = new MockBlobClass();
byte[] blobBytes = SerialisationHelper.serialise(blob);
deviceBinAttr.setBinaryValue(blobBytes);
deviceBinAttr.setValueType(CtxAttributeValueType.BINARY);

// update the attribute in the Context DB
deviceBinAttr = (CtxAttribute)
this.internalCtxBroker.update(deviceBinAttr).get();

// Alternatively, the value of the attribute could be updated as follows:
deviceBinAttr = this.internalCtxBroker.updateAttribute(deviceBinAttr.getId(),
blobBytes).get();

3.5.2.2 Retrieving and Looking-up context information

// Retrieve methods
// if the identifier of a context model object is known
// the context object is retrieved with the following method
```
Retrieve a context entity
CtxEntity retrievedCtxEntity =
(CtxEntity)this.internalCtxBroker.retrievedentityIdentifier).get();

Retrieve a context attribute with a string value
CtxModelObject ctxAttributeRetrievedStringFuture =
this.internalCtxBroker.retrieve(this.ctxAttributeStringIdentifier).get();

Retrieve a ctxAttribute with binary value
CtxAttribute ctxAttributeRetrievedBinary = (CtxAttribute)
this.internalCtxBroker.retrieve(this.ctxAttributeBinaryIdentifier).get();

// deserialise object
MockBlobClass retrievedBlob = (MockBlobClass)
SerialisationHelper.deserialise(ctxAttributeRetrievedBinary.getBinaryValue(),
this.getClass().getClassLoader());

// Lookup methods
// Context Entity and Attribute identifiers is possible to be retrieved based //
on criteria via lookup methods.
// Lookup for CtxEntities of type “FooBar”
List<CtxIdentifier> ids = internalCtxBroker.lookup(CtxModelType.ENTITY,
"FooBar").get();

// Lookup for CtxEntities of type “FooBar”
List<CtxIdentifier> ids = internalCtxBroker.lookup(CtxModelType.ATTRIBUTE,
"attrFooBar").get();

3.5.2.3 Removing context model objects
this.internalCtxBroker.remove(identifier);

3.5.2.4 Subscribing for and reacting to context change events

The ICtxBroker interface provides methods for registering CtxChangeEventListeners in order to
listen for context change events. There are two ways to subscriber for context change event
notification:

// 1a. Register listener by specifying the context attribute identifier
this.internalCtxBroker.registerForChanges(new MyCtxChangeEventEventListener(),
this.ctxAttributeStringIdentifier);

// 1b. Register listener by specifying the context attribute scope and type
this.internalCtxBroker.registerForChanges(new MyCtxChangeEventEventListener(),
this.ctxEntityIdentifier, "DeviceID");

The subscriber's implementation of the CtxChangeEventListener interface allows reacting to the
following event notifications:

private class MyCtxChangeEventEventListener implements CtxChangeEventListener {
public void onCreation(CtxChangeEvent event) {
    LOG.info(event.getId() + "": *** CREATED event ***");
}
public void onModification(CtxChangeEvent event) {
    LOG.info(event.getId() + "": *** MODIFIED event ***");
}
3.5.2.5 Maintaining history of context attributes

Context Attributes can be registered for recording each time an update on the attribute value is performed. Old values are recorded along with timestamps and stored in order to form a Context History log. Context History data can be retrieved or removed by methods provided by the ICtxBroker.

// by setting the history recording flag to true the CtxAttribute values will be stored to Context History Database upon update
ctxAttributeString.setHistoryRecorded(true);

// Retrieval of Context history data for a specified time period
// if null values are used for starting and ending Date the whole set of history data is retrieved
List<CtxHistoryAttribute> ctxHistoryData = internalCtxBroker.retrieveHistory((CtxAttributeIdentifier) ctxAttributeStringIdentifier, startDate, endDate).get();

Additionally a developer can register for recording a group (tuples) of context attributes. One CtxAttribute type is considered as the primary while the rest are considered as escorting. The recording of tuples will be performed when the value of the primary attribute is updated.

internalCtxBroker.setHistoryTuples(primaryAttrIdentifier, listOfEscortingAttributeIds);

// Retrieval of Context history tuples of data for a specified time period
Map<CtxHistoryAttribute, List<CtxHistoryAttribute>> contextTuples = retrieveHistoryTuples(primaryAttrIdentifier, listOfEscortingAttributeIds, Date arg2, Date arg3).get();

3.6 Device Management

This section provides guidelines regarding the use of the SOCIETIES Device Management facilities by Third Party context-aware services. The Device Management (DM) component acts as an intermediate layer between sensors and actuators attached to a CSS node and managed by a Third Party service. A Third Party service example using the DM component, and other DM related projects, can be found in the GitHub Code Repository.

3.6.1 API Definition

In order to use the Device Management feature, a Third Party service should use two interfaces provided by two components of the Societies platform.

- **IDevice** interface (provided by the Device Management bundle): used to be informed about the type of a newly discovered device.
- **IEventMgr** interface (provided by the Event Management bundle): used to be informed of new data from drivers.

The **IDevice** interface shall be tracked directly from the OSGi service registry while the **IEventMgr** interface is bound through the Spring Service reference declared in the bundle-context-osgi.xml file.

Figure 30 shows an example of a system where three devices are connected to the CSS nodes. The drivers and devices are not represented in the picture. In this case, the Third Party Service Consumer has bound two **IDevice** interfaces out of the three available.

![Figure 30 Tracking IDevice Instances](image)

The Device Management model is exposed in the external APIs, in the package org.societies.api.css.devicemgmt.model.

The Device model is used in order to represent all the necessary information that describes the situation of a CSS entity or a group of entities that form a CIS. In more detail, the model consists of the following main classes:

- **DeviceMgmtConstants**: This class contains property key constants used when registering **IDevice** services. All these properties are provided by the DeviceManager when it registers a new **IDevice** service. These property keys can be used by the consumer of the service as an LDAP filter to refine the set of **IDevice** services it is interested in being notified about. The LDAP Syntax of the filter is as per OSGi specifications\(^9\), which in turn are based on the RFC 1960\(^{10}\).

- **DeviceTypeConstants**: This class defines static strings identifying the type of devices that should be integrated in the User Trials. For instance: "public static final String SCREEN = "screen";".

- **DeviceMgmtEventConstants**: This class defines constants used to define a topic in which events of a given type, related to a specific device type, are sent. For instance: "public static final String SCREEN_EVENT = "actuator/screenEvent";".

- **DeviceMgmtDriverServiceNames**: This class defines constants used to get the driver service name object which will be used to invoke actions on the associated devices type. For instance: public static final String SCREEN_DRIVER_SERVICE = "actuator/screenDriverService";.

- **DeviceActionsConstants**: this class does not exist yet. It is used to provide the list of actions implemented by a device type.

---

\(^9\) [http://www.osgi.org/javadoc/r4v43/core/org/osgi/framework/Filter.html](http://www.osgi.org/javadoc/r4v43/core/org/osgi/framework/Filter.html)

All these classes defining constants should be shared among the driver developer(s) and the Third Party Service developer(s) so that both parties rely on a consistent semantic model describing the informations related to the given device.

Future revisions of the Device Management model will address the issues related to the fact that a Semantic metamodel has not been defined yet, thus forcing the developer (both Driver’s and Third Party) to rely on hard-coded constants to distinguish which type of Device, Event, Service or Action a given event deals with.

### 3.6.2 Examples

The tracking mechanism relies on the OSGi ServiceTracker functionality, as explained in the OSGi specification R4.3 compendium specification (chapter 701)\(^\text{11}\). The following code snippet highlights how the tracking functionality might be implemented, to get a reference to a device of type "Screen":

```java
public class DeviceManagerConsumer implements ServiceTrackerCustomizer, BundleContextAware {
  ...

  public void initConsumer () {
    this.serviceTracker = new ServiceTracker(bundleContext, IDevice.class.getName(), this);
    this.serviceTracker.open();
  }

  @Override
  public Object addingService(ServiceReference reference) {
    Object obj = bundleContext.getService(reference);
    IDevice iDevice = (IDevice)obj;
    // The following condition is to test the type of the device before to use it.
    if (iDevice.getDeviceType() == DeviceTypeConstants.SCREEN) {
      //store the iDevice object for future usage
      IDevice screen1 = iDevice;
    }
    }
}

### 3.6.2.1 IDevice Example Usage

This is a provided interface, it is used by the Third Party service to get device metadata (device name, type, description, id, the type of network used to connect the device, the location, the manufacturer, etc). It also provides system informations such as the name of the associated OSGi services and the events managed by the underlying driver.

```java
IDevice ls2 = iDevice;
LOG.info("Device Type: " + ls2.getDeviceName());
LOG.info("Device Type: " + ls2.getDeviceType());
LOG.info("Device ID: " + ls2.getDeviceId());
```

### 3.6.2.2 IDriverService Example Usage

This is a provided interface, it is used by the Third Party service to interact with the device, i.e. controlling an actuator or getting data from a sensor.

```java
IDevice screen1 = iDevice;
IDriverService driverService = screen1.getService(DeviceMgmtDriverServiceNames.SCREEN_DRIVER_SERVICE);
IAction ia = driverService.getAction("displayMessage");
```

Notice that, in the code snippet above, the Action "displayMessage" is not associated to any Model class (as the Action Model class does not yet exist). This example is a typical example of an ad-hoc

\(^\text{11}\) [http://www.osgi.org/download/r4v43/osgi.cmpn-4.3.0.pdf](http://www.osgi.org/download/r4v43/osgi.cmpn-4.3.0.pdf)
semantic information defined by the driver provider without using a Model approach. This is very inconvenient because using “free-text” strings with implied semantics poses a very high risk of inconsistencies in behaviour and reduces the reliability of integrating a wider set of devices.

### 3.6.2.3 IAction Example Usage

This is a provided interface, it is used by the Third Party service to invoke the action to interact with the device, i.e. actually control the actuator, or get the data from a sensor.

For an actuator, the action should have a parameter of type `Dictionary<String, Object>`.

```java
IDevice screen1 = iDevice;
IDriverService driverService = screen1.getService(DeviceMgmtDriverServiceNames.SCREEN_DRIVER_SERVICE);
IAction ia = driverService.getAction("displayMessage");
Dictionary<String, Object> dic = new Hashtable<String, Object>();
dic.put("message", "Display this message for me please ! ");
ia.invokeAction(dic);
```

For a sensor, the action has no parameter (null), but returns the sensor data under the Dictionary format. We assume in this example that the strings "getLightLevel" and "outputLightLevel" has been provided by the driver developer in an ad-hoc documentation.

```java
IDevice ls1= iDevice;
IDriverService driverService = ls1.getService(DeviceMgmtDriverServiceNames.LIGHT_SENSOR_DRIVER_SERVICE);
IAction ia = driverService.getAction("getLightLevel");
Dictionary dic = ia.invokeAction(null);
LOG.info("getLightLevel action Return: " + dic.get("outputLightLevel"));
```

### 3.6.3 Technical Configuration

If not already done for the Third Party service purpose, the following dependencies have to be added in Maven’s pom.xml file:

```xml
<dependency>
   <groupId>org.societies.api.external</groupId>
   <artifactId>societies-api</artifactId>
   <version>${release.version}</version>
   <scope>provided</scope>
</dependency>
<dependency>
   <groupId>org.societies.comm.xmpp</groupId>
   <artifactId>CommunicationFrameworkCommons</artifactId>
   <version>${version.comm}</version>
   <scope>compile</scope>
</dependency>
```

Also, in the build section of the pom.xml file, as part of the maven-bundle-plugin configuration settings, you should also add the following imports:

```xml
<Import-Package>
   ...,
   org.slf4j.*,
   org.societies.api.css.devicemgmt,
   org.osgi.framework,
   org.osgi.util.tracker,
   org.springframework.osgi.context,
   org.societies.api.osgi.event,
   ...  
</Import-Package>
```
The only mandatory service reference that needs to be injected by the runtime environment is an instance of IEventManager, so the bundle-context-osgi.xml file associated with the Third Party must include:

```
<osgi:reference id="EventManager" interface="org.societies.api.osgi.event.IEventMgr"/>
```

Finally, the following bundles should be added in your Virgo plan file (together with other SOCIETIES platform bundles such as the Communication Manager, not listed here as they are mandatory):

```
<!-- Device Management bundles. -->
<artifact type="bundle" name="org.societies.css.devicemgmt.DeviceCommsMgr"/>
<artifact type="bundle" name="org.societies.css.devicemgmt.devicemanager"/>
```

### 3.7 Identity

The Identity component provides the mapping from SOCIETIES Identifiers for Nodes to and from the underlying XMPP endpoint identifiers. It also helps in managing the multiple identities a user might have by mapping sets of Context Entities to specific role represented by an identity alias of a given user (see Figure 31 for an overview).

![Identity External APIs](image)

**Figure 31 Identity External APIs**

The semantics of each interface or class in this package is simple, so – for completeness – we feel that a selection of the Javadoc documents for this package will be enough for Third Party developers to understand.
how to use the provided or required Identity interfaces. All interfaces represent functionalities exposed by the framework to Third Party service developers, and are thus “provided” interfaces.

### 3.7.1 Interface IIdentity

#### Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getBareJid()</code></td>
<td>Returns the bare Jabber ID without resource eg, <a href="mailto:john@societies.local">john@societies.local</a></td>
</tr>
<tr>
<td><code>getDomain()</code></td>
<td>Returns the domain of a Jabber ID eg, societies.local from <a href="mailto:john@societies.local">john@societies.local</a></td>
</tr>
<tr>
<td><code>getIdentifier()</code></td>
<td>Returns the username of a Jabber ID, eg, John from <a href="mailto:john@societies.local">john@societies.local</a></td>
</tr>
<tr>
<td><code>getJid()</code></td>
<td>Returns the full Jabber ID including resource eg, <a href="mailto:john@societies.local">john@societies.local</a>/laptop</td>
</tr>
<tr>
<td><code>getType()</code></td>
<td>Returns the CSS Type</td>
</tr>
</tbody>
</table>

### 3.7.2 Interface IIdentityContextMapper

Utility class that allows to map context parameters to different identities of the user.

#### Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addMappedCtxIdentifier(IIdentity publicId, CtxIdentifier attributeID)</code></td>
<td>Add a mapped context parameter.</td>
</tr>
<tr>
<td><code>getMappedCtxIdentifier(IIdentity publicId, String attributeType)</code></td>
<td>Retrieve a mapped context parameter.</td>
</tr>
<tr>
<td><code>removeMappedCtxIdentifier(IIdentity publicId, CtxIdentifier ctxIdentifier)</code></td>
<td>Remove a mapped context parameter.</td>
</tr>
</tbody>
</table>

### 3.7.3 Interface IIdentityManager

Provides methods for Identity Management

#### Method Summary

<table>
<thead>
<tr>
<th>Type</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INetworkNode</td>
<td><code>fromFullJid(String jid)</code></td>
<td>Parses a full Jabber ID including the resource to an IIdentity object eg,</td>
</tr>
<tr>
<td></td>
<td></td>
<td><a href="mailto:john@societies.local">john@societies.local</a>/laptop</td>
</tr>
<tr>
<td>Identity</td>
<td><code>fromJid(String jid)</code></td>
<td>Parses a Jabber ID to an IIdentity object, eg, <a href="mailto:john@societies.local">john@societies.local</a></td>
</tr>
<tr>
<td>INetworkNode</td>
<td><code>getDomainAuthorityNode()</code></td>
<td>Returns the Domain Authority Node Identity</td>
</tr>
</tbody>
</table>

© SOCIETIES consortium 2012
3.7.4 Interface INetworkNode

Extends the IIdentity interface and simply returns the identity of this node.

Method Summary

<table>
<thead>
<tr>
<th>Type</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>getNodeIdentifier()</td>
<td>Returns the Node Identifier</td>
</tr>
</tbody>
</table>

3.7.5 Class Requestor

Direct Known Subclasses: RequestorCis, RequestorService

This class is used to represent a CSS requesting resources.

Constructor Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requestor(IIdentity requestorId)</td>
<td>Create a CSS requestor from the CSS identity</td>
</tr>
</tbody>
</table>

Method Summary

<table>
<thead>
<tr>
<th>Type</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>equals(Object obj)</td>
<td>Customised equivalence relation</td>
</tr>
<tr>
<td>IIdentity</td>
<td>getRequestorId()</td>
<td>Identity of the resource requestor</td>
</tr>
<tr>
<td>int</td>
<td>hashCode()</td>
<td>Customised hash function</td>
</tr>
</tbody>
</table>
### 3.7.6 Class RequestorCis

public class RequestorCis extends Requestor implements Serializable. This class is used to represent a CIS requesting resources.

<table>
<thead>
<tr>
<th>Constructor Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>RequestorCis(IIdentity requestorId, IIdentity cisRequestorId)</td>
</tr>
<tr>
<td>Create a CIS requestor from the CSS and CIS identities</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean equals(Object obj)</td>
</tr>
<tr>
<td>Customised equivalence relation</td>
</tr>
<tr>
<td>IIdentity getCisRequestId()</td>
</tr>
<tr>
<td>Identity of the CIS requestor</td>
</tr>
<tr>
<td>int hashCode()</td>
</tr>
<tr>
<td>Customised hash function</td>
</tr>
<tr>
<td>String toString()</td>
</tr>
<tr>
<td>Customised textual representation</td>
</tr>
<tr>
<td>String toXMLString()</td>
</tr>
<tr>
<td>Return a string (XML formatted) representing the object.</td>
</tr>
</tbody>
</table>

### 3.7.7 Class RequestorService

public class RequestorService extends Requestor implements Serializable. This class is used to represent a 3P Service (provided by a CSS) requesting resources

<table>
<thead>
<tr>
<th>Constructor Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>RequestorService(IIdentity requestorId, org.societies.api.schema.servicelifecycle.model.ServiceResourceIdentifier requestorServiceId)</td>
</tr>
<tr>
<td>Create a 3P service requestor from the CSS and 3P service identities</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean equals(Object obj)</td>
</tr>
<tr>
<td>Customised equivalence relation</td>
</tr>
<tr>
<td>org.societies.api.schema.servicelifecycle. getRequestorServiceId()</td>
</tr>
</tbody>
</table>
3.7.8 Enum IdentityType

Enum Constant Values: CIS, CSS, CSS_LIGHT, CSS_RICH.

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>static IdentityType</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>static IdentityType[]</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

3.8 Personalisation

The SOCIETIES platform provides personalisation functionality that enable Third Party services to customise their appearance and functionality for their users to provide better value for service. The SOCIETIES platform supports both explicit as well as proactive personalisation.

3.8.1 Explicit Personalisation

Preferences are created both manually by the user as well as by monitoring the user's behaviour and inferring user preferences from their behaviour. The monitoring of the user's behaviour depends on information sent to the SOCIETIES framework components by the Third Party services that the user interacts with.

Third Party services send user behaviour information to the societies platform in the form of Action objects:

1. interface: org.societies.api.personalisation.model.IAction
2. class: org.societies.api.personalisation.model.Action

These objects embed information about the Third Party service and the action that was performed by the user in that service.

For example, the action representing the muting of the volume for a Media Player Third Party service could be represented as an instance of Action that is created with the following parameters passed to the constructor:

```java
public Action(ServiceResourceIdentifier serviceID, String serviceType, String parameterName, String value)
IAction action = new Action(myServiceID, "mediaPlayer", "volume","0");
```

User Actions are sent to the SOCIETIES platform from Third Party services using the method:

```java
org.societies.api.useragent.monitoring.IUserActionMonitor.monitor(IIdentity owner, IAction action)
```
This information is collected and analysed by the platform from Third Party services to create user preferences and user intent models by combining this information with other context information that describes the environment and activities of the user at the time the action was performed by the user.

All of the background processing is completely hidden from the Third Party Service developer, which simply queries the Personalisation subsystem for the values of preferences or intents it is interested on.

### 3.8.2 Proactive Personalisation

User preferences and user intents are not static but they depend heavily on the current context of the user. As the context of the user changes, so do the user preferences and intents that must be applied in the system to keep it customised to the user's wishes. The SOCIETIES platform provides a mechanism for Third Party services to be informed of the changes in the preferences and intents that should be applied to them. Hence, for a Third Party service to be proactively personalised it must implement the interface:

```java
org.societies.api.personalisation.model.IActionConsumer
```

The Third Party service developer must make sure that the following methods return a valid serviceId and serviceType respectively in order for the SOCIETIES platform to be able to locate the appropriate service when it needs to:

```java
ServiceResourceIdentifier IActionConsumer.getServiceIdentifier()
String IActionConsumer.getServiceType()
```

When new preference or intent information has to be sent to a Third Party service to personalise it, these two methods are called to locate the service.

Then, when the system has acquired the service, it calls the method:

```java
boolean IActionConsumer.setIAction(IIdentity userId, IAction action)
```

where the userId is the identity of the user for whom the personalisation has to be performed and the action is the object that specifies what should be personalised and how.

A Third Party service can request user preference and user intent information from the SOCIETIES platform explicitly by using the `org.societies.api.personalisation.mgmt.IPersonalisationManager` interface.

The following two methods allow a Third Party service to request a user preference and a user intent action respectively from the Personalisation system:

```java
public Future<IAction> getPreference(
    Requestor requestor, IIdentity ownerID, String serviceType,
    ServiceResourceIdentifier serviceID, String preferenceName
);
public Future<IAction> getIntentAction(
    Requestor requestor, IIdentity ownerID,
    ServiceResourceIdentifier serviceID, String preferenceName
);
```

The `requestor` parameter defines the identity of the Third Party service that is requesting the preference information. The Requestor class has a subclass `org.societies.api.identity.RequestorService` which should be used to instantiate the identity of the Third Party service. If the requestor parameter is not properly defined, the Personalisation system will not return valid preference information.

The `ownerID` parameter defines the identity of the user for which the Third Party service is requesting the information. Currently, as the platform does not support multiple identities, the Third Party service should utilise the `org.societies.api.identity.IIdentityManager.getThisNetworkNode()` to retrieve the identity of the user on the current node as long as the service is running on a user's node.

The `serviceType` parameter is defined by the Third Party service developer in the service meta data.
The `serviceID` parameter is defined by the platform and is retrieved and saved by the 3P service during initialisation by invoking the static method “`generateServiceResourceIdentifier(IIdentity identity, java.lang.Class<?> callingClass)`” provided by the `org.societies.api.internal.servicelifecycle.ServiceModelUtils` class.

The `preferenceName` parameter is also defined by the Third Party service developer and it is the service that is able to understand the semantics of this parameter and apply it properly since it was the service that sent this information to the platform using the method

```
org.societies.api.useragent.monitoring.IUserActionMonitor.monitor(IIdentity owner, IAction action)
```

### 3.9 Registering a Third Party Service on a Cloud/Rich Node

On the cloud/rich SOCIETIES node, the Service Discovery core component has a listener registered with the OSGI registry for deployed bundles being added/removed from the OSGI Registry itself. To allow your Third Party Service to be picked up by Service Discovery component, it will need to contain a `TargetPlatform` property of "SOCIETIES" (See Figure 32).

![Figure 32 Third Party Service Registration](image)

Below is the `bundle-context-osgi.xml` file for the example Calculator Service:

```
<osgi:service ref="CalcService"
interface="org.societies.example.calculator.ICalc" >
  <osgi:service-properties>
    <entry key="TargetPlatform" value="SOCIETIES" />
    <entry key="ServiceProvider" value="ICT-SOCIETIES" />
    <entry key="ServiceMetaModel" value-ref="serMetamodel" />
  </osgi:service-properties>
</osgi:service>

<bean id="serMetamodel"
class="org.societies.api.schema.servicelifecycle.model.Service">
  <property name="authorSignature" value="johndoe" />
  <property name="serviceDescription" value="Basic calculator service, supports Add/Subtract" />
  <property name="serviceName" value="Calculator Service" />
</bean>
```
The *serMetaModel* bean contains part of the ServiceModel's metadata for that service. It does not contain all of the Service Model Metadata because metadata such as version and bundle ID, just to name a few, are part of the implementation and are read directly from the OSGI registry at runtime. The ServiceResourceIdentifier value is also generated at runtime.

Service Discovery offers also the functionality to query both locally and remotely for a list of services offered by a given SOCIETIES node. The Identity of the target CSS or CIS is the required parameter for remote calls:

```java
public interface IServiceDiscovery {

    /**
     * Based on a provided Node identity, return all services shared by
     * the specified CSS/CIS to other CSS's or CIS’s
     * @param node Identity
     * @return the retrieved List of services
     * @throws ServiceDiscoveryException
     */
    public Future<List<Service>> getServices(IIdentity node) throws ServiceDiscoveryException;

    /**
     * This method returns all services for the current node
     * @return the retrieved List of services
     * @throws ServiceDiscoveryException
     */
    public Future<List<Service>> getLocalServices() throws ServiceDiscoveryException;

```
4 Conclusions and Future Work

In this deliverable, the main APIs that are of interest for third party developers have been described in detail. The document provides technical guidelines on setting up and configuring development environment and tools necessary for integrating and testing 3rd party services with the SOCIETIES platform. Following this, a detailed description was provided for some of the existing set of APIs targeted at 3rd party services. This API set is composed by Java Interfaces, XML Schemas and Android Interfaces. The Java Interfaces are designed to be invoked locally from 3rd party services which are deployed within the same runtime as the SOCIETIES cloud/rich nodes. The XML schema APIs allow for remote standards based communication, via XMPP, regardless of the implementation technology of the 3rd party service, thus providing a platform agnostic API. Finally, the Android Interfaces provide a local invocation point for Android applications to utilise the features of the SOCIETIES APIs thus reducing the impact of unnecessary remote messages from Android devices (i.e. light client node) to Cloud nodes. At the time of writing of this first deliverable version, some interfaces were not fully implemented or documented but these will be included in the final version of this deliverable (D6.2).

The most important technical aspect that we wish to highlight is the fact that the binding of functional interfaces to XML messaging (provided by the communication framework) by means of precise XSD Schemas, allows the SOCIETIES platform to be accessed by any implementation technology that is able to send and receive XML messages that comply with those schemas. The technical design of SOCIETIES allows 3rd party service developers to adopt numerous deployment models including integrating with the SOCIETIES platform on Virgo and Android nodes.

Future work includes building on the technical support for 3rd party service developers which will be provided through online collaboration tools (guidelines & tutorials, bug reporting, forums, etc - see Figure 33 and Figure 34). The 3rd party developer documentation will also be embedded within the Eclipse IDE to enable ease of access when developing. This will be reported on in the final version of this deliverable, D6.2.

Figure 33 Online SOCIETIES Forums
**Figure 34 Online Bug-tracking Tool**

![Image of Online Bug-tracking Tool](image_url)

<table>
<thead>
<tr>
<th>Tracker</th>
<th>open</th>
<th>closed</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bug</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Feature</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Support</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Use Case</td>
<td>5</td>
<td>-</td>
<td>5</td>
</tr>
<tr>
<td>Task</td>
<td>-</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Version</th>
<th>open</th>
<th>closed</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1 - 30 Apr 2012</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0.2 - 31 May 2012</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0.3 - 30 June 2012</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0.4 - 31 July 2012</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Priority</th>
<th>open</th>
<th>closed</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Normal</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>High</td>
<td>2</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>Urgent</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Immediate</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subproject</th>
<th>open</th>
<th>closed</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Integration Tests</td>
<td>93</td>
<td>35</td>
<td>119</td>
</tr>
<tr>
<td>Behaviour Tests</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Capability Tests</td>
<td>57</td>
<td>5</td>
<td>62</td>
</tr>
<tr>
<td>Dependability and Security Tests</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Performance Tests</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Scalability Tests</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Assignee</th>
<th>open</th>
<th>closed</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alan Walsh</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Maciej Lackey</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Andre Hipe</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Barbosa</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

No data to display.