INTEGRATION TEST FOR JEMMA PLATFORM WITH PAX EXAM TOOLKIT

Authors:
Patrick Facco
CSP “Innovazione nelle ICT” s.c. a r.l.
(patrick.facco@csp.it)

Ferdinando Ricchiuti
CSP “Innovazione nelle ICT” s.c.a r.l.
(ferdinando.ricchiuti@csp.it)

Index

Introduction

This tutorial describes how to set up an integration test for the Jemma Platform using PaxExam and the JUnit framework.
The tutorial also provides an integration test as an example. This single case has been designed to be extensible to accomplish a number of integration tests.
The general approach used here, is to inject some data in the platform and read the responses from the serial port to simulate the interaction on the wireless network. This test detects the correct reaction of the full stack of the framework started by the injected data packets. In order to simplify the creation of the test packets, this sample software provides a way to make raw dump of packets received and transmitted by the wireless device.
The main steps in creating a test using the provided example are the following:

1. set-up the hardware to test;
2. start the jemma platform in logging mode, using the properly configuration;
3. create different dumps for different use-cases (e.g. devices discovery, device join, ecc...) and save the related dumps;
4. write a JUnit test for a given use-case by injecting the related dump.

You may perform different types of checks, for example you can check if it is raised at least one event of a certain type, or print all events that are raised and check the correctness of the sequence.
In this example, after recording the traffic related to an operation of join of a Zigbee device, we will use the packets saved to check if it is raised at least one "*ZigBee Device Profile message*" event.

Required Tools

The system requirements for running the test are:

- Oracle java jdk version 1.7 or higer
- maven version 2.2.1 or higher
- eclipse kepler or netbeans 8.0

Packets Dump

To enable packets dump you must set two variables in the javagal's configuration file (config.properties):

```
#enable dump
dump = true

#directory path where dump files are saved
dumpDir = /my/favourite/path
```

If you set this pair of configurations and launch jemma platform normally all the traffic that passed through the serial port is saved as a binary file in the directory pointed by the value of "dumpDir".

The name of each files saved using the following convention:

```
<unixepochtimestamp>-<r/w>.bin
```

For example: 1426169449329-r.bin.

The first part is the timestamp of the file creation in unix epoch format, the second part is the charachter "w" if the packet is written or "r" if the packet is readed.

Create a Pax Exam Integration Test

First of all you must include the following dependencies for pax exam and javaGAL inside your pom file:

Group id	**Arifact id**	**version**
org.eclipse.osgi	org.eclipse.osgi	3.10.0
org.eclipse.osgi	org.eclipse.osgi.services	3.4.0
javax.xml	javax.xml	1.3.4
org.rxtx	org.rxtx	1.0.0
jssc[1]	jssc	2.8.0
org.apache.commons	commons-lang3	3.3.2
org.ops4j.pax	runner	1.8.7-SNAPSHOT
junit	junit	4.8.1
org.ops4j.pax.exam	pax-exam-container-paxrunner	2.3.0
org.ops4j.pax.exam	pax-exam-junit4	2.3.0
org.ops4j.pax.exam	pax-exam-link-assembly	2.3.0
org.ops4j.pax.exam	pax-exam-testforge	2.3.0
org.ops4j.pax.runner	pax-runner-no-jcl	1.7.6
org.restlet.osgi	org.restlet.ext.servlet	2.1.6
org.osgi	org.osgi.core	4.1.0
org.eclipse.equinox	org.eclipse.equinox.common	3.6.0.v20100503
javax.inject	javax.inject	1

The test is provided in a separated maven project as an integration test. In this new Maven project, we use Pax Exam 2.3 with a native container which uses the OSGi FrameworkFactory API to look up an OSGi framework for running the tests. The Native Container launches the OSGi framework in the same VM. We basically use the dependencies from the Pax Exam and the javaGAL documentation.

1 This dependency is mandatory if you want to run the test under Windows. If you run the test under Linux environment you must not include jssc.

To run correctly the javaGAL, you need to pass some parameters to the virtual machine. The parameters used for the test are listed below:

```
eclipse.ignoreApp = true
osgi.noShutdown = true
it.telecomitalia.ah.driver.autoinstall = true
zgd.dongle.uri = /dev/ttyUSB0
zgd.dongle.speed = 115200
zgd.dongle.type = freescale
osgi.instance.area = /tmp
org.energy_home.jemma.ah.configuration.file = EmptyConfig
org.ops4j.pax.logging.DefaultServiceLog.level = DEBUG
```

All bundles we want to load on OSGi and all virtual machine parameters are loaded by a method annotated as *@Configuration*.
This method returns an "*Option*" array used by the PaxExam core to load correctly all the bundles and their dependencies.

For the bundles of JUnit framework and the OSGi equinox platform there are two methods that return all the bundles and dependencies needed. This two methods, *junitBundles()* and *equinox()*, are facilities provided by PaxExam platform.
If you want to test your application in another OSGi platform you can load all the required bundles using the namesake method (e.g. *felix()*).

```
@Configuration
public Option[] config()
{
     return options(
         junitBundles(),
         systemProperty("eclipse.ignoreApp").value("true"),
         systemProperty("osgi.noShutdown").value("true"),
         systemProperty("zgd.dongle.uri").value("/dev/ttyUSB0"),
         systemProperty("zgd.dongle.speed").value("115200"),
         …
         …
         equinox(),
         mavenBundle("org.eclipse.osgi",
               "org.eclipse.osgi.services",
               "3.4.0"),
         mavenBundle("org.ops4j.pax.logging",
               "pax-logging-api",
               "1.7.2"),
         …
         …
         mavenBundle("org.energy-home", "jemma.osgi.javagal",
"2.0.1")
        );
    }
```

To lookup the context and obtain a reference for the javaGAL factory you need to inject two variables using the @Inject annotation.
This annotation provides the external initialization of variables through the mechanism of inversion of control.

```
//BundleContext Injection
@Inject
BundleContext bc;

//GAL Factory Injection
@Inject
GalExtenderProxyFactory galfactory;
```

When the main variables are initializated and all bundles are correctly deployed you can write the test methods as a generic JUnit test case.
The first test is aimed to check if the context is injected without errors.

```
@Test
public void checkInjection()
{
     assertNotNull(bc);
}
```

This method is annotated as *@Test* as a generic JUnit test method.

The second method is optional but very useful to check if all bundles are completly loaded and all dependencies are satisfied.
The idea is to print the names, the ids, the versions and the state of all the bundles deployed on the OSGi platform.
A bundle can be in one of the following six states: *UNINSTALLED*, *INSTALLED*, *RESOLVED*, *STARTING*, *STOPPING* and *ACTIVE*.
We check, for simplicity, if the bundles are active, installed or resolved.
The meaning of these three states are the following:

- **ACTIVE:** The bundle is running without problems.
- **INSTALLED:** The bundle is installed in OSGi platform but not yet resolved. You must deploy all the dependencies required.
- **RESOLVED:** The bundle is resolved but not yet started.

The first step is to obtain the bundles list from the bundle context. Once we have the whole list of bundles, we can print all the information described above.
If all bundles are in *ACTIVE* state the test can go ahead.
No exceptions are thrown in this method, because we want only to print an overview of usefull informations about the bundles deployed.

```
@Test
public void getBundleslist()
{
     System.out.println("Printing bundle list..");

     for(Bundle b : bc.getBundles())
     {
        System.out.print("Bundle " +
          b.getBundleId() + ":" +
          b.getSymbolicName() +
          " is " + b.getVersion());

        if(b.getState() == Bundle.ACTIVE)
           System.out.println("BUNDLE ACTIVE");

        if(b.getState() == Bundle.INSTALLED)
           System.out.println("BUNDLE INSTALLED");

        if(b.getState() == Bundle.RESOLVED)
           System.out.println("BUNDLE RESOLVED");
     }
}
```

The last method is aimed to write one or many packets through the serial port and verify if the response from the device raises the aspected events. The event fired is catched and checked by an appropriate class listener.
The listener implemented for catching the events is described by the interface *GatewayEventListenerExtended.java* into the package *org.energy_home.jemma.zgd* of javaGAL.
The *GatewayEventListenerExtended* is an implementation of a ZigBee Device specification described in the document "*Network Device: Gateway Specification*" provided by the "*ZigBee Alliance*"[2].
The implementation of the listener for the events, in this example, is very simple: a string that rapresents the callback launched is setted and printed.
In this way you can read immediatly on the log which callback is called and you can perform some operation on the string rapresentation like counting, comparison, etc...
The packets, written on the serial port, are read from the same directory used for save the dump. In particular only the packets marked as "written" are selected.
The packets are also ordered by timestamp in order to respect the sequentiality of the dialogue.
In this example we write some packets dumped and we count how many times a "*Zigbee Device Profile*" callback is triggered in order to demonstrate a simple operation of event count.
At the end an assertion checks if we have received at least one ZigBee Device Profile message.

2 The document is available and free downloadable at the following url: https://docs.zigbee.org/zigbee-docs/dcn/11/docs-11-5552-00-00mg-zigbee-network-device-zigbee-gateway-standard-version-1.pdf.

```
@Test
public void checkMessage()
{
     try
     {
          //Creates a new listener
          GatewayEventListenerForTest gel =
             new GatewayEventListenerForTest();
          //Register the listener on the GAL injected
          galfactory.createGatewayInterfaceObject()
                .setGatewayEventListener(gel);
          //count of callbacks
          int count = 0;
         //grab the directory where dumps are saved
         File dumpDirW = new File(
            folderPath + File.separator + 'w');
            if(!dumpDirW.exists())
            {
               //...print error and exit...
            }
            //Grab and sort all files marked as "w"
            String[] filesW = dumpDirW.list();
            Arrays.sort(filesW);
            //write each binary packet on the serial port
            for(String filesW1 : filesW)
            {
                //write packet using an utility method
                galfactory.writeWrapper(
                   fileTobyte(dumpDirW +
                      File.separator + filesW1));

                //sleep 1 second, waiting for response
                Thread.sleep(1000);

                //count profile callbacks
                if(gel.getCallback().equals("notifyZDPCommand"))
                {
                    count++;
                }
            }
            // we have received at least one
            //ZigBee Device Profile message?
            assertTrue(count > 0);
        } //...catcher for exceptions...
    }
```

Launch the Test

All the test written run automatically during the javaGAL bundle compilation whit the following maven command:

```
mvn clean install
```

If you want to skip the test you must use the following command:

```
mvn clean install -DskipTests
```

If you want to launch the test within an IDE, such as eclipse[3] or NetBeans[4], you simply have to launch the class called *JavaGALTest* as a JUnit test.

Obtain the code

All the code of this tutorial is available on github.
You can obtain the code cloning the following repository:

https://github.com/p3dr16k/javagalpaxexam.git

Alternately you can download a zip file at the following url:

https://github.com/p3dr16k/javagalpaxexam/archive/master.zip

3 https://eclipse.org/
4 https://netbeans.org/