OSGi Tutorial

A Step by Step Introduction to OSGi Programming Based on the Open Source Knopflerfish OSGi Framework

Sven Haiges
October 2004
sven.haiges@vodafone.com

Table of Contents

Table of contents i

List of Tables ii

List of Figures iii

1 Introduction 1

2 Installing Knopflerfish OSGi 2

3 Creating your First Bundle 5

3.1 Create a New Project for your First Bundle 5
3.2 Create the manifest.mf File 5
3.3 Create an Ant Build File 6
3.4 Create the Activator Class 7
3.5 Build and Install your First Bundle 8

4 Creating your First Service 10

4.1 Update the manifest.mf File 10
4.2 Create the Service Interface 11
4.3 Create the Service Implementation 11
4.4 Create an Activator that Registers the Service 11
4.5 Build and Install the Service Bundle 12

5 Using other Services 13

5.1 Update the manifest.mf File 13
5.2 Retrieve a Service – the bad way 14
5.3 Using a ServiceListener to Dynamically Bind Services 15
5.4 Using a ServiceTracker to track Services 18

List of References ix

List of Tables

List of Figures

Figure 1: Command Window after clicking the startup.bat file 3

Figure 2: The Knofperfish OSGi Desktop 4

Figure 3: KF Desktop after installing the Hello World Bundle 9

1 Introduction

This tutorial introduces you to OSGi programming based on the open source Knopflerfish OSGi framework. I chose Knopflerfish, because it is easy to install and provides a great desktop GUI, that will help you to get your first bundles deployed in an OSGi Framework.

First, the reader is quickly introduced to the installation of Knopflerfish. Second, you will create your first OSGi bundle and deploy it in this framework. Step by step, you will create more bundles, register and retrieve services and manage their dependencies. By the end of this tutorial, you should have a basic understanding of OSGi programming.

Please notice the references at the end of this document for further information about OSGi and other tutorials. The author would like to tank the maintainers of Knopflerfish for their great OSGi Framework. Please also have a look at the Knopflerfish website[1] to find more about this framework.

[1] http://www.knopflerfish.org

2 Installing Knopflerfish OSGi

Installation of Knopflerfish is really easy. Please point your web browser to http://www.knopflerfish,org and go to the download page. I recommend to download the complete framework (approx. 7 MB, including all sources and documentation.

Please make sure, that you have a current Java software development kit installed. We also assume, that you have access to an IDE such as the open source Eclipse (by the way, Eclipse is also an OSGi Framework).

Next, open the jar file you just downloaded with WinZip. You can also simply change the file extension from .jar to .zip and then extract the file in a new directory using <right-click-on-file>/<extract to...>.

To start the Knopflerfish Framework, you can now simply double-click on the framework.jar file that you will find in this directory:

```
<install-directory>/knopflerfish.org/osgi/framework.jar
```

I recommend to create a small startup.bat file. Simply copy the following line into a file called startup.bat and place it into the same directory as the framework.jar file:

```
java -jar framework.jar
```

You should see that a command window opens and soon after that the so-called Knopflerfish OSGi Desktop is starting up.

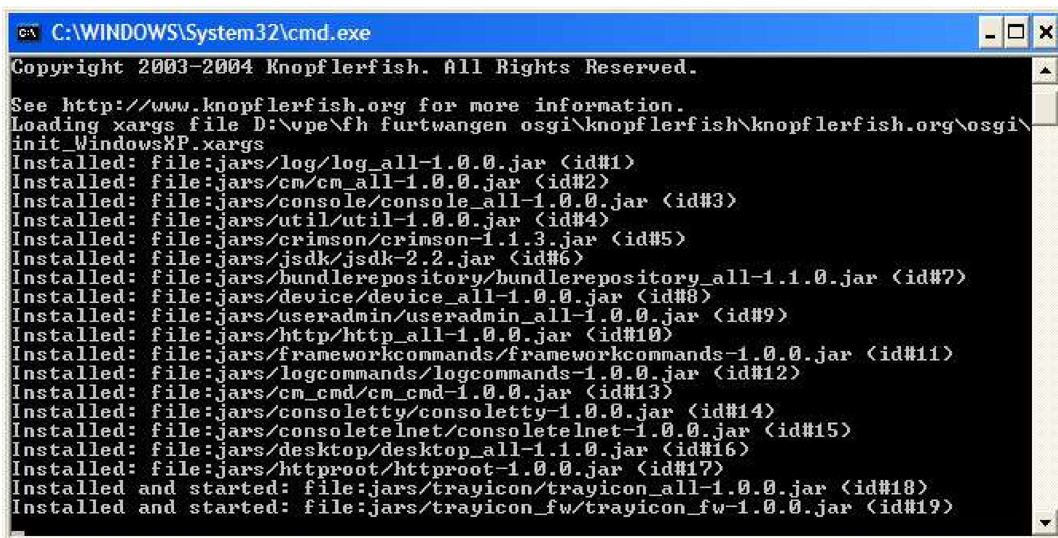

Figure 1: Command Window after clicking the startup.bat file

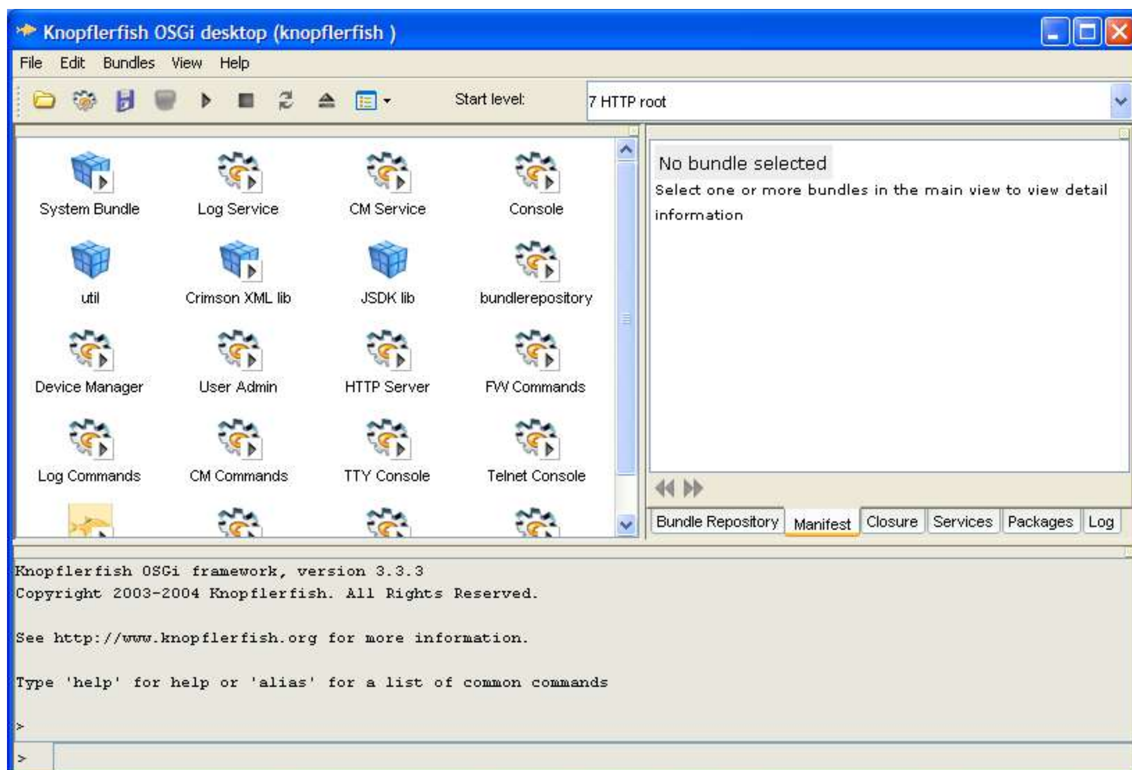

Figure 2: The Knofperfish OSGi Desktop

Congratulations! You successfully installed Knopflerfish and started up the framework for the first time. You will find more information about Knopflerfish startup options and the Knopflerfish Desktop on the KF website: http://www.knopflerfish.org.

The Knopflerfish Desktop let's you manage the KF framework. It is the visible part of the management agent for your framework. For example, you can install new bundles, start and stop them, update bundles or uninstall them.

3 Creating your First Bundle

In this chapter, you will create your first OSGi bundle and deploy it in the Knopflerfish framework. We assume that you have access to an IDE, such as Eclipse.

In OSGi programming, the elements that can be installed in a framework are called bundles. Bundles are simply jar files, that typically contain the Java class files of the service interfaces, their implementation and some more meta information in a META-INF/manifest.mf file. Services are Java interfaces and once your bundle registered a servcie with the OSGi framework, other bundles may use your "published" service.

Your first bundle will simply create a background thread that prints out "Hello World" every second.

3.1 Create a New Project for your First Bundle

Open your IDE, such as Eclipse, and create a new Java Project. Name it simplebundle. Create separate folders for your source code and the generated classes (I recommend src and classes). Make sure that you import the framework.jar file in your Java build path. Otherwise, you will not be able to access the OSGi classes and interfaces provided by Knopflerfish.

3.2 Create the manifest.mf File

Next, add a META-INF directory to your project. This folder will contain the manifest.mf file that describes your bundle. It is later used by the framework to get information about you bundle and to deploy it successfully.

Add the following text to your manifest.mf file:

```
Manifest-Version: 1.0
```

```
Bundle-Name: simplebundle
Bundle-SymbolicName: simplebundle
Bundle-Version: 1.0.0
Bundle-Description: Demo Bundle
Bundle-Vendor: Vodafone Pilotentwicklung GmbH
Bundle-Activator: de.vpe.simplebundle.impl.Activator
Bundle-Category: example
Import-Package: org.osgi.framework
```

The most important properties here are Bundle-Activator and Import-Package. Bundle-Activator tells the framework which class is your Activator class, this is a kind of "main" class for your bundle. In our example, we will later create a de.vpe.simplebundle.impl.Activator class and this class will be launched by the framework once we deploy and start the bundle.

The Import-Package property tells the framework that our bundle needs to have access to all classes contained in the org.osgi.framework package. Generally, every bundle that you create needs to have access to the classes of the OSGi framework.

3.3 Create an Ant Build File

We will use Ant to build the project. Create a build.xml file in the top of your directory structure and add the following targets:

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

<project name="simplebundle" default="all">

  <target name="all" depends="init,compile,jar"/>

  <target name="init">
       <mkdir dir="./classes"/>
       <mkdir dir="./build"/>
  </target>


  <target name="compile">

      <javac destdir       = "./classes"
             debug         = "on"
             srcdir        = "./src"
      >
     </javac>
  </target>

  <target name="jar">
         <jar basedir  = "./classes"
              jarfile  = "./build/simplebundle.jar"
              compress = "true"
              includes = "**/*"
              manifest = "./meta-inf/MANIFEST.MF"
          />

  </target>

  <target name="clean">
       <delete dir = "./classes"/>
       <delete dir = "./build"/>
```

```
  </target>

</project>
```

You can now test run the build.xml file. In Eclipse, simply right-click on the build.xml file and choose Run>Ant Build. The build file should complete successfully. If not, please check your directory structure and make changes where necessary.

3.4 Create the Activator Class

Most bundles do have an Activator class, specified in the bundle's manifest.mf file. The Activator class implements the BundleActivator interface. This interface requires the implementation of two methods, start() and stop(), which are used by the framework to manage your bundle.

Create a de.vpe.simplebundle.impl package. In OSGi programming, you typically separate the service interfaces from their inplementation. As our first bundle will not register any services, our de.vpe.simplebundle package will be empty. The impl sub-package will store the Activator class, that starts our bundle.

Now create a class called Activator that implements the BundleActivator interface:

```
package de.vpe.simplebundle.impl;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

/**
 * @author Sven Haiges | sven.haiges@vodafone.com
 */
public class Activator implements BundleActivator {

  public static BundleContext bc = null;

  public void start(BundleContext bc) throws Exception {
       Activator.bc = bc;

  }

  public void stop(BundleContext bc) throws Exception {
       Activator.bc = null;
  }

}
```

Notice that the start() and stop() methods receive an BundleContext object. You should always store this object once you get it and set the reference back to null when the bundle is stopped. That way, the Garbage Collector can do its work and free unused resources.

Next, we create a Thread subclass, that will print out “Hello World” every five seconds.

```
package de.vpe.simplebundle.impl;

/**
 * @author Sven Haiges | sven.haiges@vodafone.com
```

```
 */
public class HelloWorldThread extends Thread {
  private boolean running = true;

  public HelloWorldThread() {

  }

  public void run() {
       while (running) {
            System.out.println("Hello World!");

            try {

  Thread.sleep(5000);
            } catch (InterruptedException e) {

  System.out.println("HelloWorldThread ERROR: " + e);
            }
       }
  }

  public void stopThread() {
       this.running = false;
  }
}
```

Finally, we have to create a new thread while the bundle is started (in the start() method) and we have to stop the thread once the bundle is stopped. We also add some debugging code to see when the bundle ist started and stopped:

```
public class Activator implements BundleActivator {

  public static BundleContext bc = null;

  private HelloWorldThread thread = null;

  public void start(BundleContext bc) throws Exception {
       System.out.println("SimpleBundle starting...");
       Activator.bc = bc;
       this.thread = new HelloWorldThread();
       this.thread.start();

  }

  public void stop(BundleContext bc) throws Exception {
       System.out.println("SimpleBundle stopping...");
       this.thread.stopThread();
       this.thread.join();
       Activator.bc = null;
  }

}
```

3.5 Build and Install your First Bundle

Again, build the project using the build.xml file. You should now see a simplebundle.jar file in the ./build directory. Open Knopflerfish and choose File>Open Bundle. Choose the

simplebundle.jar file and install the bundle. The bundle is automatically activated and you should see a new icon in the upper left window. Every five seconds, the bundle prints out a new Hello World. Try to start and stop the bundle using the buttons of the Knopflerfish OSGi Desktop.

Congratulations, you just created your first bundle!

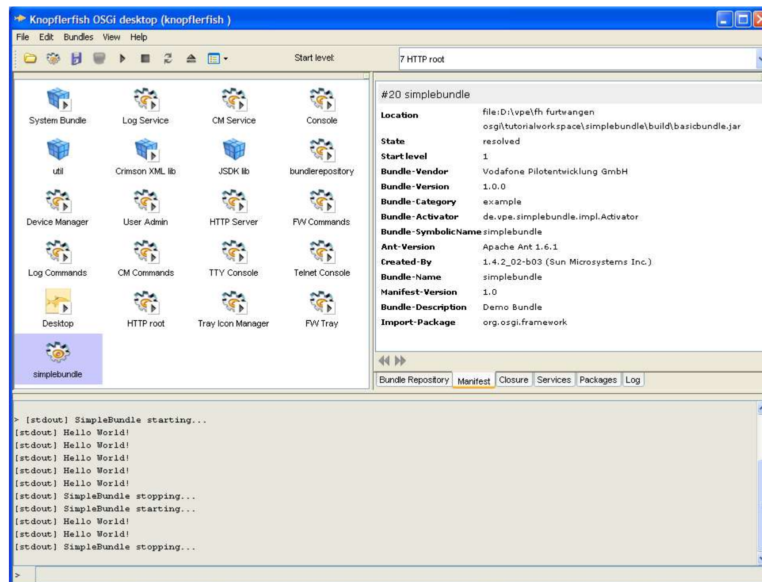

Figure 3: KF Desktop after installing the Hello World Bundle

4 Creating your First Service

This chapter will help you to create your first service. Again, we need to create a bundle, but this time our interface package will not be empty. It will contain the service interface, which is a simple Java interface.

First, copy your SimpleBundle project and name it DateBundle. The service that you will create will format a given Date object and return the formatted date. Make sure that you change the build.xml file and the manifest.mf file to meet the names of the newly created bundle.

4.1 Update the manifest.mf File

There is one small change in the manifest.mf file: we have to add an Export-Package property. Otherwise, other services will later not be able to retrieve the service interface and thus will not be able to use our service.

Make sure that your manifest.mf file for the new project looks like this:

```
Manifest-Version: 1.0
Bundle-Name: firstservice
Bundle-SymbolicName: firstservice
Bundle-Version: 1.0.0
Bundle-Description: Demo Bundle
Bundle-Vendor: Vodafone Pilotentwicklung GmbH
Bundle-Activator: de.vpe.firstservice.impl.Activator
Bundle-Category: example
Import-Package: org.osgi.framework
Export-Package: de.vpe.firstservice
```

4.2 Create the Service Interface

Notice, that we also renamed the packages to de.vpe.firstservice. Create a Java interface called FirstService:

```
package de.vpe.firstservice;
import java.util.Date;

public interface FirstService {
  public String getFormattedDate(Date date);
}
```

4.3 Create the Service Implementation

Next, we create the FirstService Implementation in the impl subpackage:

```
package de.vpe.firstservice.impl;

import java.text.DateFormat;
import java.util.Date;

import de.vpe.firstservice.FirstService;

public class FirstServiceImpl implements FirstService {

  public String getFormattedDate(Date date) {
      return  DateFormat.getDateInstance(DateFormat.SHORT).format
(date);
  }

}
```

The implementation of our service interface is rather easy, but this is OK for now. The implementation simply returns a formatted date in short style.

4.4 Create an Activator that Registers the Service

Finally we have to register our service. This will be achieved in the start() method of our Activator class. We first create a service implementation and then register this service under the name of the service interface. All registering operations are done via methods in the BundleContext object. This object is the glue between our bundle and the framework.

```
package de.vpe.firstservice.impl;

import java.util.Hashtable;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
```

```
import org.osgi.framework.Constants;
import org.osgi.framework.ServiceRegistration;

import de.vpe.firstservice.FirstService;

public class Activator implements BundleActivator {

  public static BundleContext bc = null;

  public void start(BundleContext bc) throws Exception {
       System.out.println(bc.getBundle().getHeaders().get
(Constants.BUNDLE_NAME) + " starting...");
       Activator.bc = bc;
  FirstService service = new FirstServiceImpl();
  ServiceRegistration registration = bc.registerService(
FirstService.class.getName(), service, new Hashtable());
       System.out.println("Service registered: FirstService");
  }

  public void stop(BundleContext bc) throws Exception {
       System.out.println(bc.getBundle().getHeaders().get
(Constants.BUNDLE_NAME) + " stopping...");
       Activator.bc = null;
  }
}
```

The registerService method of the BundleContext receives three parameters: the first parameter is the name of the service interface. The second is the service implementation. The third parameter can be used to supply additional information about the service as key/value pairs.

4.5 Build and Install the Service Bundle

Again, build the bundle using the build.xml file. Make sure that you changed the name of the bundle jar file to something like firstservicebundle.jar. Install it using the KF Desktop. You should see the debug messages that you added to the code.

The next bundle that you will create will use the service that you just registered.

5 Using other Services

Copy the FirstService project and rename it to FirstServiceUser. Don't forget to rename the file names and property names in the build.xml file, the manifest.mf file and also change the package names of the new project.

The bundle that you will create will only use services, so again, we will have an empty service package. The only class that you have to create for this bundle is a new Activator class. This Activator will look up the FirstService and use it.

5.1 Update the manifest.mf File

Your manifest for the FirstService user bundle should look like this:

```
Manifest-Version: 1.0
Bundle-Name: firstserviceuser
Bundle-SymbolicName: firstserviceuser
Bundle-Version: 1.0.0
Bundle-Description: Demo Bundle
Bundle-Vendor: Vodafone Pilotentwicklung GmbH
Bundle-Activator: de.vpe.firstserviceuser.impl.Activator
Bundle-Category: example
Import-Package: org.osgi.framework,de.vpe.firstservice
```

Notice that we added a comma and a new package name to the Import-Package header. We now declare, that our bundle needs to have access to the de.vpe.firstservice package. A framework will always check that this package is available to the bundle before the Activator is started.

5.2 Retrieve a Service – the bad way

Whenever you retrieve a service, you must understand that an OSGi Framework is a quite dynamic place where services might be available or not. It is very important to double-check that you really retrieved a valid service implementation and not null, whenever you get a service. Soon after using the service, you should also "unget" it, which means that the framework is informed that you do not use the service any longer.

Our first example, the easiest one but also the worst regarding code quality, retrieves the FirstService from the BundleContext and uses the service. We will show you later why this code is problematic in several ways.

```
package de.vpe.firstserviceuser.impl;

import java.util.Date;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.Constants;
import org.osgi.framework.ServiceReference;

import de.vpe.firstservice.FirstService;

public class Activator implements BundleActivator {

   public static BundleContext bc = null;

   public void start(BundleContext bc) throws Exception {
        System.out.println(bc.getBundle().getHeaders().get(
   Constants.BUNDLE_NAME)
   + " starting...");
        Activator.bc = bc;

        ServiceReference reference = bc.getServiceReference
(FirstService.class.getName());
        FirstService service = (FirstService)bc.getService(reference);
        System.out.println("Using FirstService: formatting date: " +
service.getFormattedDate(new Date()));
        bc.ungetService(reference);
   }

   public void stop(BundleContext bc) throws Exception {
        System.out.println(bc.getBundle().getHeaders().get
             Constants.BUNDLE_NAME)
             + " stopping...");
        Activator.bc = null;
   }
}
```

We first retrieve a ServiceReference from the BundleContext. The getServiceReference() method simply asks for the name of the service interface that we would like to use. Once we have a ServiceReference, we use the getService() method to acquire the service implementation object, cast it to FirstService and use it.

You can now build the project and install it using the KF desktop. If your FirstService is started, everything will be fine and you will see the debug output.

The problem is that there is no guarantee that the FirstService is actually available. Try the following: stop both services and then first start the bundle that uses the FirstService. You will probably get a NullPointerException, because the mehtod getServiceReference

returned null (no service was yet registered, so the framework could not give you what you asked for).

A better solution would be to check if the return value is null:

```
ServiceReference reference = bc.getServiceReference
  (FirstService.class.getName());

if (reference != null)
{
  FirstService service = (FirstService)bc.getService(reference);
  System.out.println("Using FirstService: formatting date: " +
service.getFormattedDate(new Date()));
  bc.ungetService(reference);

}
else
{
  System.out.println("No Service available!");
}
```

This solves the problem with the NullPointerException, but: if our service is not available at startup, we can never use this service! Somehow we should regularly check if the service is available or not. Or even better: the framework should inform us, as soon as a suitable service is available. You can achieve this, using ServiceListeners.

5.3 Using a ServiceListener to Dynamically Bind Services

Using the BundleContext, it is possible to register a ServiceListener with the framework. With an optional filter object, you can exactly specify for which services you want to receive ServiceEvents. Service events are sent out by the framework whenever a new service registers, unregisters or modifies its properties.

The next example shows the modified start()-method code. First, a ServiceListener is registered with the framework. The filter string is in LDAP style and tells the framework only to send service events concerning the FirstService interface to us. Notice, that we do not retrieve the FirstService directly in the start() method. Instead, we do a little trick and obtain all Services that match our filter. Then, we send out ServciceRegistered events for every service that is found (in our example, this will just be one service and thus one ServiceEvent). Be sure to add the ServiceListener interface to you Activator. This forces you to add a serviceChanged method (see below).

```
public void start(BundleContext bc) throws Exception
{
System.out.println("start " + getClass().getName());
Activator.bc = bc;


String filter = "(objectclass=" + FirstService.class.getName() + ")";
bc.addServiceListener(this, filter);
ServiceReference references[] = bc.getServiceReferences(null, filter);

for (int i = 0; references != null && i < references.length; i++)
{
   this.serviceChanged(new ServiceEvent(ServiceEvent.REGISTERED,
```

```
      references[i]));
}
}
```

The serviceChanged method will receive all ServiceEvents for FirstService service changes. The method I implemented starts to use a service once a FirstService Registers (it starts a thread that uses the service every second or so) and stops once the service unregisters. If the service changes (e.g. the properties of the service changed), we stop using the service, obtain a new reference to the service and start again.

```
  public void serviceChanged(ServiceEvent event) {
       switch (event.getType()) {
       case ServiceEvent.REGISTERED:
            log("ServiceEvent.REGISTERED");
            this.service = (FirstService) Activator.bc.getService(event

       .getServiceReference());
            this.startUsingService();
            break;
       case ServiceEvent.MODIFIED:
            log("ServiceEvent.MODIFIED received");
            this.stopUsingService();
            this.service = (FirstService) Activator.bc.getService(event

       .getServiceReference());
            this.startUsingService();
            break;
       case ServiceEvent.UNREGISTERING:
            log("ServiceEvent.UNREGISTERING");
            this.stopUsingService();
            break;

       }
  }

  private void stopUsingService() {
       this.thread.stopThread();
       try {
            this.thread.join();
       } catch (InterruptedException e) {
            e.printStackTrace();
       }
       this.service = null;
  }

  private void startUsingService() {
       this.thread = new ServiceUserThread(this.service);
       this.thread.start();

  }

  private void log(String message) {
       System.out.println(Activator.bc.getBundle().getHeaders().get(

  Constants.BUNDLE_NAME)

  + ": " + message);
  }
```

Again, you can now build your project and install the bundle jar file (or update it). You will see that our bundle starts to use the FirstService as soon as it is available. Try to stop the FirstService, while the bundle is active. You will see that the bundle stops using the

service and waits until it becomes again available. Although the ServiceUserThread class is just a basic thread, here is the code for it.

```
public class ServiceUserThread extends Thread {
  private FirstService service = null;

  private boolean running = true;

  public ServiceUserThread(FirstService service) {
       this.service = service;

  }

  public void run() {
       Date date = null;
       String formattedDate = null;

       while (running) {
            date = new Date();

            try {

  formattedDate = this.service.getFormattedDate(date);
            } catch (RuntimeException e) {

  System.out

            .println("RuntimeException occured during service usage: "


       + e);
            }

            System.out.println("ServiceUserThread: converted date has
value: "

       + formattedDate);
            try {

  Thread.sleep(1000);
            } catch (InterruptedException e) {

  System.out.println("ServiceUserThread ERROR: " + e);
            }
       }

  }

  public void stopThread() {
       this.running = false;
  }
}
```

As you have seen, there is pretty much code to write to be able to dynamically start and stop using other services. Fortunately, there is a utility class available, that helps you to solve this problem. The ServiceTracker class is available for you to monitor services. We will now show you how to use the ServiceTracker.

5.4 Using a ServiceTracker to track Services

A ServiceTracker object automatically tracks all ServiceEvents for a specified service and gives you the possibility to customize what should happen, once a servcie appeards or disappears. To enable this customization, you have to implement a ServiceTrackerCustomizer interface and provide it to a ServiceTracker object.

The following code is the updated version of the Activator's start() method. You will see, that most code actually was moved out of this Activator because we now use a ServiceTracker.

```
public void start(BundleContext bc) throws Exception {
   System.out.println(bc.getBundle().getHeaders().get(
            Constants.BUNDLE_NAME)
            + " starting...");
   Activator.bc = bc;

   customizer = new MyServiceTrackerCustomizer(bc);
   tracker = new ServiceTracker(bc, FirstService.class.getName(),
customizer);
   tracker.open();
}
```

Now, we take a look at the MyServiceTrackerCustomizer class, that implements the ServiceTrackerCustomizer interface:

```
public class MyServiceTrackerCustomizer implements
ServiceTrackerCustomizer {

   private ServiceUserThread thread = null;

   private BundleContext bc;

   public MyServiceTrackerCustomizer(BundleContext bc) {
        this.bc = bc;
   }

   public Object addingService(ServiceReference reference) {
        FirstService service = (FirstService) bc.getService(reference);
        if (this.thread == null) {
             this.thread = new ServiceUserThread(service);
             this.thread.start();
             return service;
        } else
             return service;
   }

   public void modifiedService(ServiceReference reference, Object
serviceObject) {
        this.thread.stopThread();
        try {
             this.thread.join();
        } catch (InterruptedException e) {
             e.printStackTrace();
        }
        FirstService service = (FirstService) bc.getService(reference);
        this.thread = new ServiceUserThread(service);
        this.thread.start();
   }

   public void removedService(ServiceReference reference, Object
serviceObject) {
```

```
        this.thread.stopThread();
        try {
            this.thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        this.thread = null;
    }
}
```

The addingService method gets the service and starts a new thread if none exists. We check if the thread is null, because only then we would like to start exactly one new thread. It could happen that many FirstServices are registered, but even then we only want to use one service in one thread.

The modifiedService method simply stops the execution of the thread and restarts with the new service. To extend the usefulness, we could actually check if the service that changed is really the service that we currently use. Only if our service changed, there is a need to restart.

Finally, the removedService method simply stops the execution of the thread. The service is not used any more, after the method returns.

List of References

Gravity: Richard S. Hall, OSGi and Gravity Service Binder Tutorial, 2004, http://oscar-osgi.sourceforge.net/tutorial/

KF: Erik Wistrand, Develop OSGi Bundles, 2004, http://www.knopflerfish.org/programming.html

OSGi Intro: OSGi Alliance, OSGi Technology, 2004, http://www.osgi.org/osgi_technology/index.asp?section=2

OSGi Platform: OSGi Initiative, OSGi Service Platform R3, March 2004, http://www.osgi.org