OSGi Tutorial v2

A Step by Step Introduction to OSGi Programming

Based on the open source
Knopflerfish OSGi Framework

www.knopflerfish.org

Originally by Sven Haiges 2004 (sven.haiges@vodafone.com)
Updated by Erik Wistrand 2009 (wistrand@makewave.com)

Table of Contents

1 Introduction

1.1 Original introduction by Sven Haiges

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 (www.knopflerfish.org) to find more about this framework.

1.2 Updated introduction by Erik Wistrand

First, thanks to Sven for writing the original tutorial!

Since 2004, OSGi adaption has become much more widespread and the specification has evolved to version 4. This updated tutorial reflects this by making sure the examples work on the latest (2.2) release on the KF framework and by adding a chapter about Declarative Services. Screen snapshots has also been updated, and full source to all examples are available for download at the KF website.

This tutorial including all the examples are available at the Knopflerfish subversion repository:

https://www.knopflerfish.org/svn/knopflerfish.org/trunk/docs/kf_osgi_tutorial/

2 Installing Knopflerfish OSGi

Installation of Knopflerfish is really easy. Point your web browser to http://www.knopflerfish,org and go to the download page.

I recommend downloading the complete framework (approx. 18 MB, including all sources and documentation). Make sure that you have a current Java software development kit installed. I also assume that you have access to an IDE such as the open source Eclipse (by the way, Eclipse is also an OSGi Framework).

The downloaded file will have a name similar to `knopflerfish_osgi_<version>.jar`

Next, start the downloaded jar file by double-clicking on it, or by running

```
java -jar <downloaded jar file>
```

if you don't have a desktop environment, you can start the installation in batch mode using

```
java -jar <downloaded jar file> -batch
```

The default installation directory is the current directory. When installation is complete, you will have a new directory

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

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

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

You can also type the command

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

You may see a command window opening and soon after that the Knopflerfish OSGi Desktop is starting up.

Note: if you get an OS/firewall warning about Java trying to act as server, this is because the default startup includes an OSGi web server. The KF Desktop may also try to access update informationon the KF website

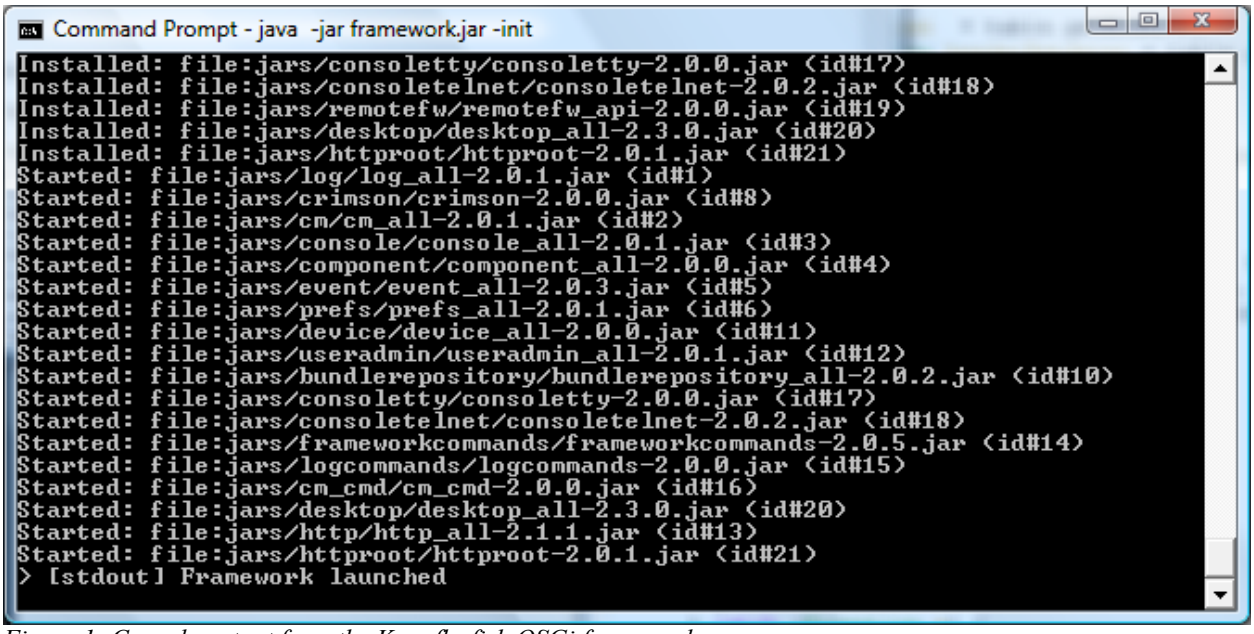

Figure 1: Console output from the Knopflerfish OSGi framework

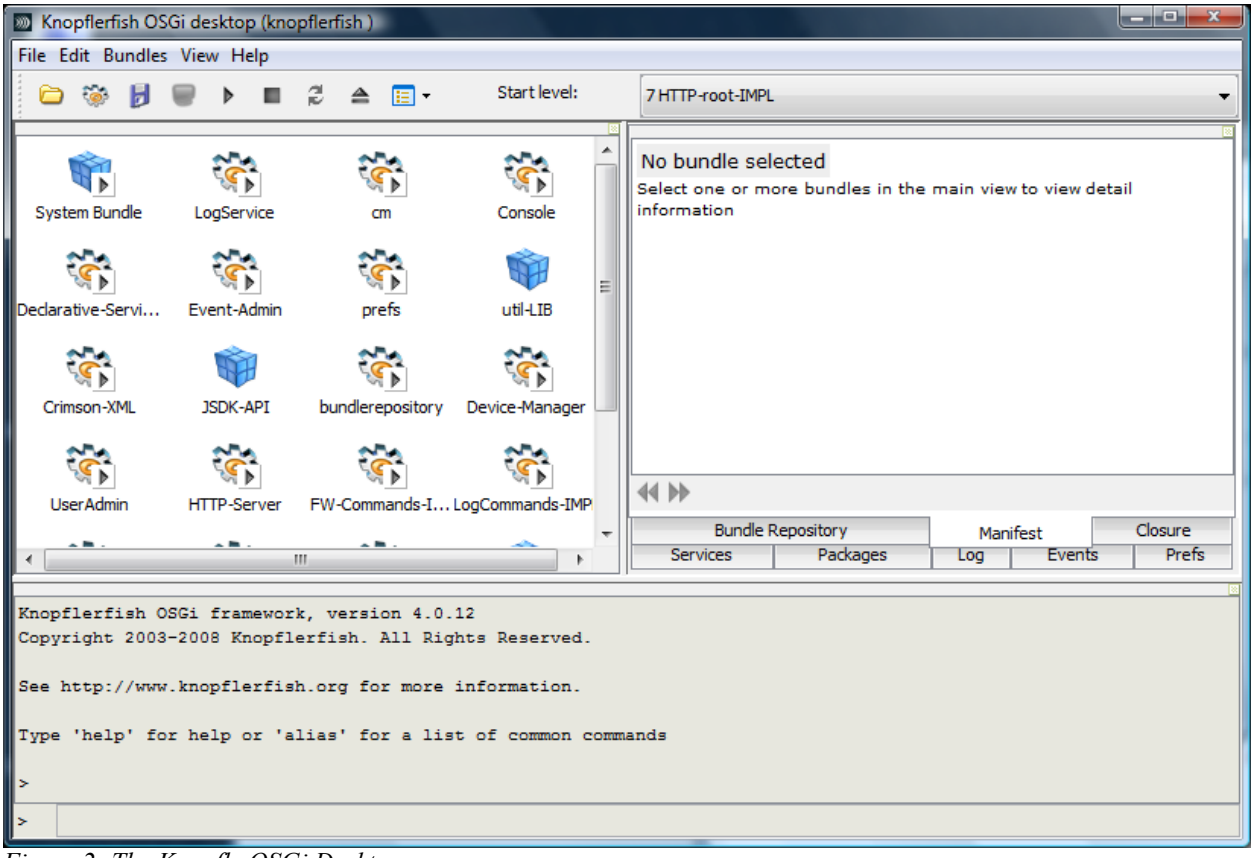

Figure 2: The Knopfle OSGi Desktop

Congratulations!

You successfully installed Knopflerfish and started up the framework for the first time. You can 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.

Figure 3: Desktop toolbar

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. The source code for all of these examples are available on the KF web sites, including Ant build files. See chapter 6 for details

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 registers a service with the OSGi framework, other bundles may use your published service.

Your first bundle will simply create a background thread which 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 output (I recommend `src` and `out`). 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 manifest file named `manifest.mf` that describes your bundle. It will be used by the framework to get information about you bundle and to deploy it successfully.

Add the following text to the `manifest.mf` file:

```
Manifest-Version: 2.0
Bundle-Name: simplebundle
Bundle-SymbolicName: simplebundle
Bundle-Version: 1.0.0
Bundle-Description: Demo Bundle for the KF OSGi tutorial
Bundle-Vendor: Knopflerfish
Bundle-Activator: org.knopflerfish.tutorial.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 `org.knopflerfish.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">

  <property name="kf.dir" location="../.."/>

  <property name="framework.jar"
             location="${kf.dir}/knopflerfish.org/osgi/framework.jar"/>

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

  <target name="init">
     <mkdir dir="out/classes"/>
  </target>

  <target name="compile">
     <javac destdir       = "out/classes"
            debug         = "on"
            srcdir        = "src">
       <classpath>
          <pathelement location="${framework.jar}"/>
       </classpath>
     </javac>
  </target>

  <target name="jar">
     <jar basedir  = "out/classes"
          jarfile  = "out/${ant.project.name}.jar"
          compress = "true"
          includes = "**/*"
          manifest = "manifest.mf"/>
  </target>

  <target name="clean">
     <delete dir = "out"/>
  </target>
</project>
```

Note: You have to set the kf.dir property to the installation dir mentioned on page4.

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

3.4 Create the Activator class

Most bundles 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. See the OSGi R4 specification, chapter 4 "Life Cycle Layer" for details.

Create a `org.knopflerfish.tutorial.simplebundle.impl` package. In OSGi programming you typically separate the service interfaces from their implementation. As our first bundle will not register any services, the

`org.knopflerfish.tutorial.simplebundle` package will be empty. The `impl` subpackage will store the Activator class that starts our bundle.

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

```
package org.knopflerfish.tutorial.simplebundle.impl;

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

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 two seconds.

```
package org.knopflerfish.tutorial.simplebundle.impl;

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

  public HelloWorldThread() {
    super("Hello World thread");
  }

  public void run() {
    while (running) {
      System.out.println("Hello World!");
      try {
        Thread.sleep(2000);
      } catch (InterruptedException e) {
        System.out.println("HelloWorldThread ERROR: " + e);
      }
    }
    System.out.println("thread stopped");
  }

  public void stopThread() {
    System.out.println("stopping thread");
    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 when the bundle is stopped. We also add some debugging code to see when the bundle ist started and stopped:

```
package org.knopflerfish.tutorial.simplebundle.impl;

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

```
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;
    thread = new HelloWorldThread();
    thread.start();
  }

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

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 ./out directory. Open Knopflerfish and choose **File>Open Bundle**. Choose the simplebundle.jar file and install the bundle. The bundle is automatically selected and the icon view is scrolled to its icon, but the bundle is not yet started.

Tip: Drag & Drop of the jar file into the desktop works on Windows platforms.

Start the bundle using the start icon in the desktop, of the **Bundle>Start** menu item.

Now, every five seconds, the bundle prints out a new "Hello World" to the console. Try to start and stop the bundle using the buttons of the Knopflerfish OSGi Desktop.

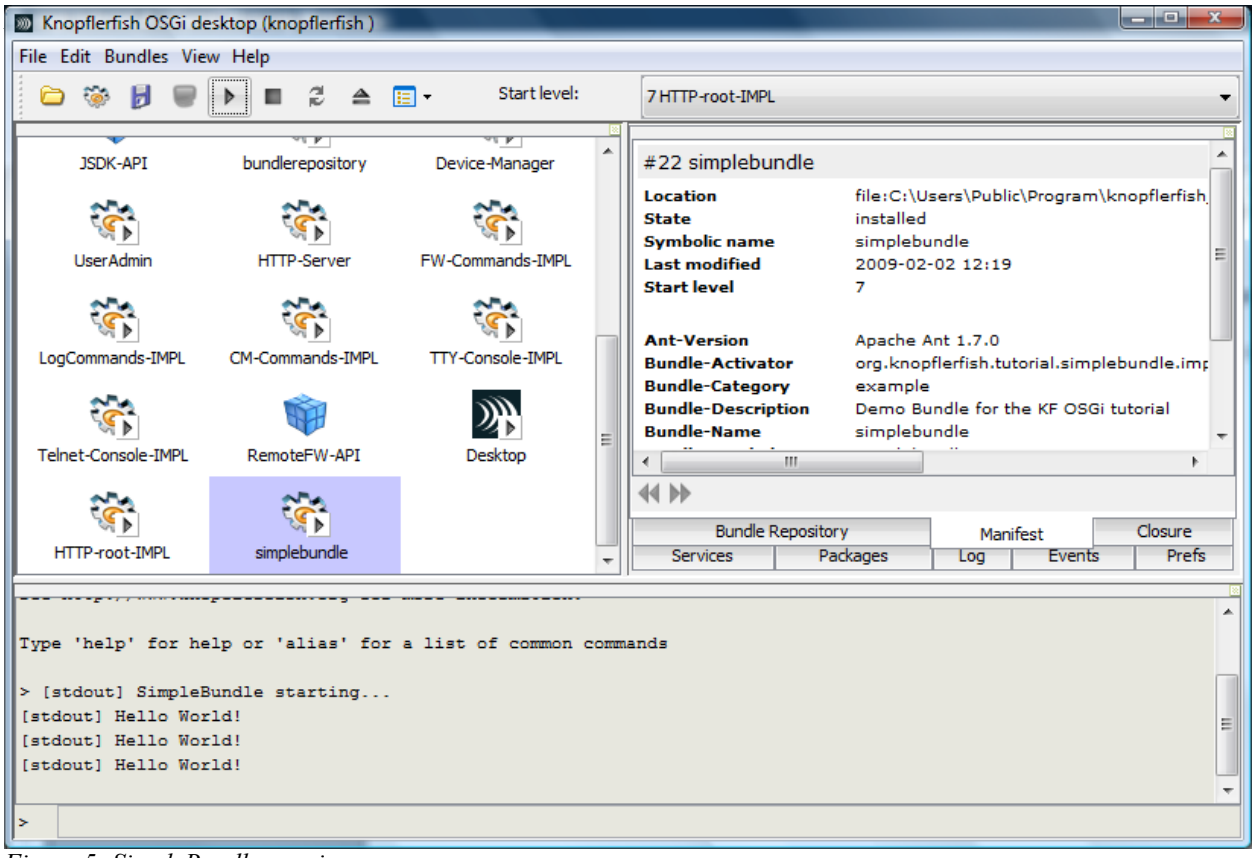

Figure 5: SimpleBundle running

Congratulations!
You just created and installed your first 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 DateService. 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 must add an `Export-Package` property. Otherwise, other services won't 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: 2.0
Bundle-Name: dateservice
Bundle-SymbolicName: org.knopflerfish.tutorial.dateservice
Bundle-Version: 1.0.0
Bundle-Description: Demo Service Bundle for the KF OSGi tutorial
Bundle-Vendor: Knopflerfish
Bundle-Activator: org.knopflerfish.tutorial.dateservice.impl.Activator
Bundle-Category: example
Import-Package: org.osgi.framework
Export-Package: org.knopflerfish.tutorial.dateservice
```

4.2 Create the service interface

Notice that we also renamed the packages to org.knopflerfish.tutorial.dateservice. Create a Java interface called `DateService`:

```
package org.knopflerfish.tutorial.dateservice;
import java.util.Date;

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

4.3 Create the service implementation

Next, we create the `DateService` implementation in the impl subpackage:

```
package org.knopflerfish.tutorial.dateservice.impl;

import java.text.DateFormat;
import java.util.Date;
import org.knopflerfish.tutorial.dateservice.DateService;


public class DateServiceImpl implements DateService {
  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 date service

Finally we have to register our service. This will be done 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, which acts as glue between our bundle and the framework.

```
package org.knopflerfish.tutorial.dateservice.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 org.knopflerfish.tutorial.dateservice.DateService;

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;

    DateService         service       = new DateServiceImpl();
    ServiceRegistration registration =
      bc.registerService(DateService.class.getName(),
                         service,
                         new Hashtable());
    System.out.println("Service registered: DateService");
  }

  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. In this example we don't add any key/value pairs.

Note the small trick of getting the information from the bundle's manifest headers to display a meaningful debug text.

4.5 Build and install the date 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 `dateservice.jar`. Install and start 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 DateService project and rename it to DateServiceUser. 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 DateService and use it.

The Desktop's graph view may be used to view the final result. It displays relationships between bundles by painting links from a central bundle to the depending bundles.

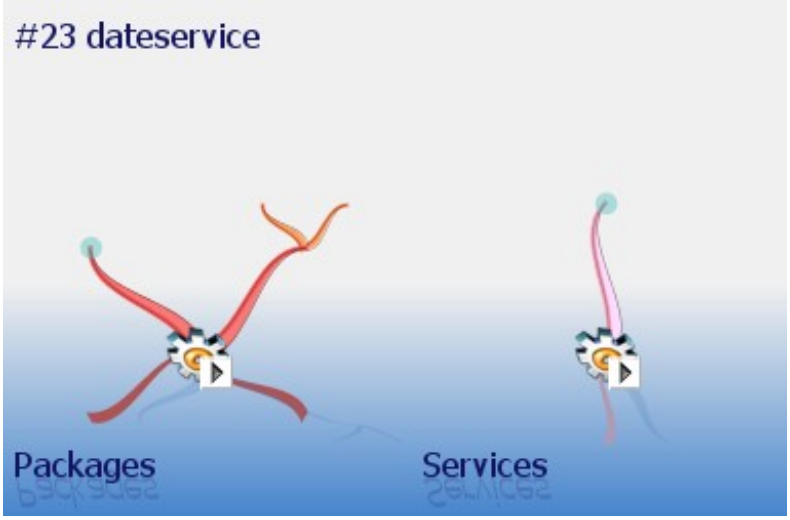

Figure 6: The dateservice bundle exporting packages and a service

5.1 Update the manifest.mf file

Your manifest for the DateService user bundle should look like this

```
Manifest-Version: 2.0
Bundle-Name: dateserviceuser
Bundle-SymbolicName: org.knopflerfish.tutorial.dateserviceuser
Bundle-Version: 1.0.0
Bundle-Description: Demo Date Service User Bundle for the KF OSGi tutorial
Bundle-Vendor: Knopflerfish
Bundle-Activator: org.knopflerfish.tutorial.dateserviceuserbad.impl.Activator
Bundle-Category: example
Import-Package: org.osgi.framework,org.knopflerfish.tutorial.dateservice
```

Notice that we added a comma as separator and a new package name to the `Import-Package` header. We thus declare that our bundle needs to have access to the `org.knopflerfish.tutorial.dateservice` package. A framework will always check that this package is available to the bundle before the Activator is started. If the package is not available, the bundle cannot be resolved. See the OSGi R4 specification, chapter 3, Module

Layer for more information.

5.2 Retrieve a service – the bad way

Whenever you retrieve a service, you should understand that an OSGi Framework is a quite dynamic place where services might be appear and disappear at any time. It is very important to double-check that you really retrieved a valid service object 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 DateService from the BundleContext and uses the service. We will show you later why this code is problematic in several ways.

```
package org.knopflerfish.tutorial.dateserviceuser.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 org.knopflerfish.tutorial.dateservice.DateService;

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(DateService.class.getName());

    DateService service = (DateService)bc.getService(reference);
    System.out.println("Using DateService: 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 DateService and use it.

You can now build the project and install it using the KF desktop. If your DateService is started, everything will be fine and you will see the debug output. The problem is that there is no guarantee that the DateService is actually available. Try the following: stop both bundles and then first start the bundle that uses the DateService.

You will probably get a `NullPointerException`, because the `getServiceReference()` returned null (no service was yet registered, so the framework could not give you what you asked for).

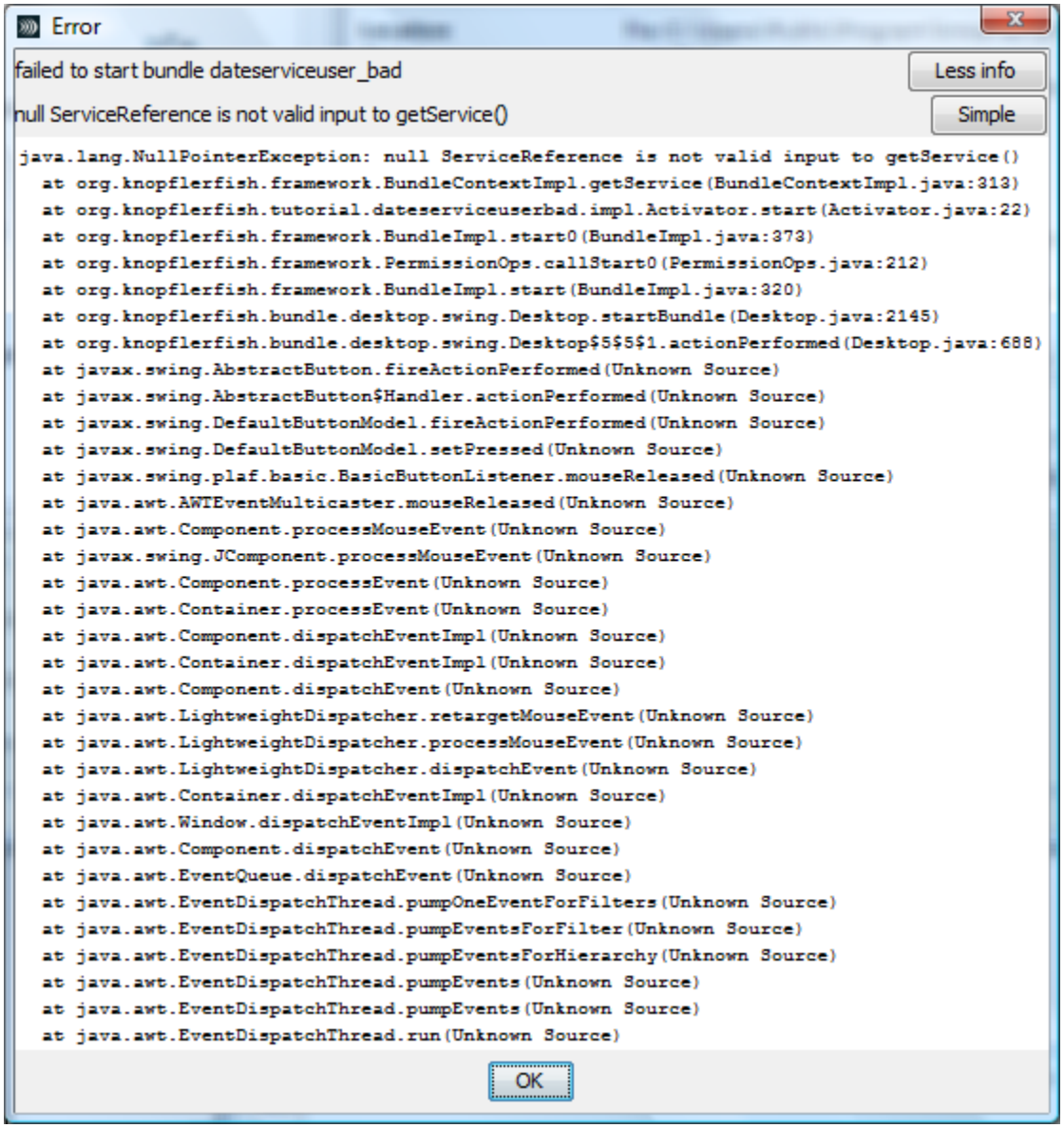

Figure 7: NullPointerException when no service is available

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

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

if (reference != null) {
  DateService service = (DateService)bc.getService(reference);
  System.out.println("Using DateService: 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, nothing happens and the bundle will never use the 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 (see OSGi R4 spec, section 3.2.6 "Filter Syntax") and tells the framework only to send service events concerning the DateService interface to us. Notice that we do not retrieve the DateService directly in the start() method. Instead, we do a little trick and obtain all services that match our filter. Then, we'll fake ServiceRegistered events for every service that is found (in our example, this will just be one service and thus one ServiceEvent). Add a ServiceListener implementation instance to your Activator.

```
public void start(BundleContext bc) throws Exception  {
   Activator.bc = bc;
   log("starting");

   String filter = "(objectclass=" + DateService.class.getName() + ")";
   bc.addServiceListener(listener, filter);

   ServiceReference references[] = bc.getServiceReferences(null, filter);
   for (int i = 0; references != null && i < references.length; i++)
     {
       listener.serviceChanged(new ServiceEvent(ServiceEvent.REGISTERED,
                                                references[i]));
     }
 }
```

The `listener.serviceChanged()` method will receive all ServiceEvents for DateService service changes. The method I implemented starts to use a service once a DateService 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.

```
ServiceListener listener = new ServiceListener() {
    public void serviceChanged(ServiceEvent event) {
      switch (event.getType()) {
      case ServiceEvent.REGISTERED:
        log("ServiceEvent.REGISTERED");
        dateService =
            (DateService)Activator.bc
            .getService(event.getServiceReference());
        startUsingService();
        break;
      case ServiceEvent.MODIFIED:
        log("ServiceEvent.MODIFIED received");
        stopUsingService();
        dateService =
            (DateService)Activator.bc
            .getService(event.getServiceReference());
        startUsingService();
        break;
      case ServiceEvent.UNREGISTERING:
        log("ServiceEvent.UNREGISTERING");
        stopUsingService();
        break;
      }
    }

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

    private void startUsingService() {
      thread = new ServiceUserThread(dateService, "listener example");
      thread.start();
    }
  };
```

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 DateService as soon as it is available. Try to stop the DateService, 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 DateService  dateService = null;
  private boolean      running = true;

  public ServiceUserThread(DateService dateService, String threadName) {
    super(threadName);
    this.dateService = dateService;
  }

  public void run() {
    String formattedDate = null;

    while (running) {
      Date date = new Date();
      try {
        formattedDate = dateService.getFormattedDate(date);
      } catch (RuntimeException e) {
        System.out.println("RuntimeException occured during service usage: "
                           + e);
      }
      System.out.println(getName() + ": converted date has value: "
                         + formattedDate);
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {
        System.out.println("ServiceUserThread ERROR: " + e);
      }
    }
  }

  public void stopThread() {
    System.out.println("stopping " + this);
    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 {
   Activator.bc = bc;
   log("starting");
   tracker =  new ServiceTracker(bc,
                                 DateService.class.getName(),
                                 customizer);
   tracker.open();
 }

 public void stop(BundleContext bc) throws Exception {
   log("stopping");
   tracker.close();
   Activator.bc = null;
 }
```

We also add an implementation of the `ServiceTrackerCustomizer` interface (as a new unnamed class):

```
 ServiceTrackerCustomizer customizer = new ServiceTrackerCustomizer() {
     public Object addingService(ServiceReference reference) {
       log("addingService");
       DateService service = (DateService) bc.getService(reference);
       if (thread == null) {
         thread = new ServiceUserThread(service, "tracker example");
         thread.start();
         return service;
       } else {
         return service;
       }
     }

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




     public void removedService(ServiceReference reference, Object
                                serviceObject) {
       log("removedService");
       thread.stopThread();
       try {
         thread.join();
```

```
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
        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 DateServices 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.

5.5 Using Declarative Services to track services

All of the previous methods require quite a lot of code to handle service registrations and unregistrations. To make this easier, OSGi has specified the Declarative Services component (see OSGi R4 compendium, chapter 112).

This component (which is incluced in Knopflerfish and implemented as a bundle itself) allows a bundle to declare its service dependencies in an XML file, and inject service objects directly into bundle class methods.

To use declarative services you need to create the an XML component descriptor file, component.xml and add it to the bundle jar.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<component
   xmlns = "http://www.osgi.org/xmlns/scr/v1.0.0"
   name  = "kf.tutorial.datelisten">
  <implementation
class="org.knopflerfish.tutorial.dateserviceuserdeclarative.impl.Component"/>
  <reference
     name      = "DATESERVICE"
     interface = "org.knopflerfish.tutorial.dateservice.DateService"
     bind      = "setDateService"
     unbind    = "unsetDateService"
  />
</component>
```

The above description file defines a component which has an implementation class `org.knopflerfish.tutorial.dateserviceuserdeclarative.impl.Component`, similar to a bundle activator, but it need not implement anything more than the two methods `setDateService(DateService)` and `unsetDateService(DatService)`

As soon this new component bundle is running and a service if the type `org.knopflerfish.tutorial.dateservice.DateService` registers, the bind method will automatically be called. When the service unregister, the unbind metod will be

called.

Additionally, the description file must be referenced from the bundle's manifest using the `Service-Component` header. So, if the component.xml is store in the bundle as OSGI-INF/component.xml, this manifest line should be added:

```
Service-Component: OSGI-INF/component.xml
```

The component implementation can now simply be written as

```
package org.knopflerfish.tutorial.dateserviceuserdeclarative.impl;

import org.knopflerfish.tutorial.dateservice.DateService;

public class Component {
  DateService       dateService;
  ServiceUserThread thread;

  protected void setDateService(DateService dateService) {
    this.dateService = dateService;

    if(thread == null) {
      thread = new ServiceUserThread(dateService);
      thread.start();
    }
  }

  protected void unsetDateService(DateService dateService) {
    this.dateService = null;

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

Note how *no references* to OSGi packages are needed in this example!

6 Example source code

All source code in this tutorial, including Ant build files, manifest files and component XML descriptors are available on the Knopfler web site at:

https://www.knopflerfish.org/svn/knopflerfish.org/trunk/docs/kf_osgi_tutorial/example_source

1. **simplebundle**
   the simple Hello World bundle used in chapter 3
2. **dateservice**
   the DateService interface and implementation used in chapter 4
3. **dateserviceuser_bad**
   the DateService user bundle as described in section 5.2
4. **dateserviceuser_bad2**
   the DateService user bundle as described in section 5.2, but with a null check
5. **dateserviceuser_listener**
   the DateService user bundle as described in section 5.3
6. **dateserviceuser_tracker**
   the DateService user bundle as described in section 5.4
7. **dateserviceuser_declarative**
   the DateService user declarative component as described in section 5.5

Note: all of the dateserviceuser_ bundles uses the same `ServiceUserThread` class. This class is copied into all of these bundles, but could just as easily have been put in a common utility bundle.

7 References

1. Gravity: Richard S. Hall, OSGi and Gravity Service Binder Tutorial, 2004
   http://oscar-osgi.sourceforge.net/tutorial/
2. KF: Erik Wistrand, Develop OSGi Bundles, 2004
   http://www.knopflerfish.org/programming.html
3. OSGi Intro: OSGi Alliance, OSGi Technology, 2004,
   http://www.osgi.org/osgi_technology/index.asp?section=2
4. OSGi Platform: OSGi Initiative, OSGi Service Platform R
   http://www.osgi.org