Embedding Hamlets

Writing Web-based user interfaces for embedded devices running OSGi

René Pawlitzek (rpa@zurich.ibm.com)
Research and Development Engineer
IBM

Skill Level: Intermediate

Date: 19 Jun 2007

The open source Hamlets framework can help aid your Web development and properly separate content from presentation. The OSGi framework provides an excellent tool for development on embedded devices. Together, the two frameworks work as a team to provide browser-based interactivity to the humblest gadgets -- such as the lowly coffee maker. Read on to find out how it works.

Hamlets provide an easily used and easily understood framework for developing Web-based applications. Due to their lightweight design and modest resource requirements, they are also well suited for the embedded space. In this article, you will learn how to use Hamlets to write Web-based user interfaces for embedded devices running OSGi.

What are Hamlets?

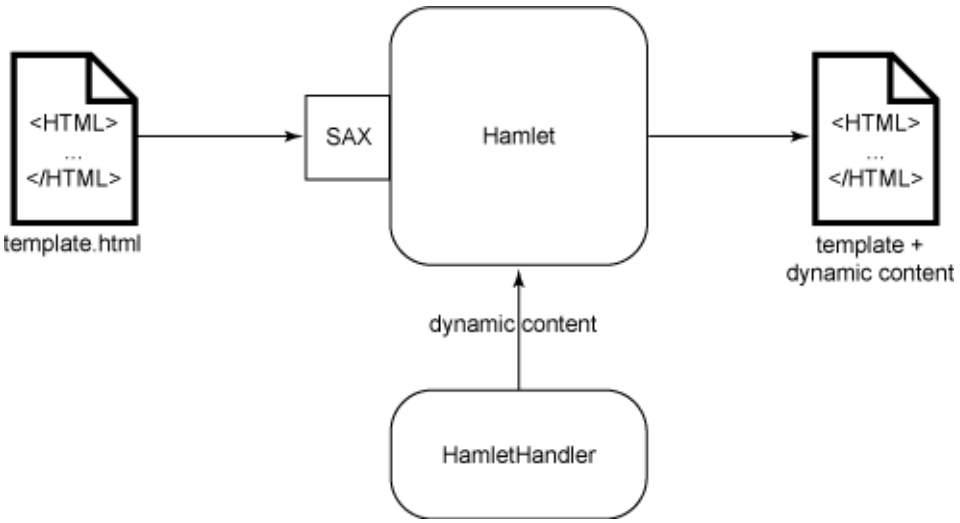
Servlets are an ideal choice for Web development for a number of reasons: portability, efficiency, safety, extensibility, and flexibility. Few viable alternatives exist that can match the power and elegance of servlets.

Despite their attractive properties, out-of-the-box servlets lack an important feature: support for the separation of content from presentation. If servlets are exclusively used for the development of Web-based applications, HTML and Java™ code inevitably end up intermingled in the same source file.

To solve this problem, I proposed an easy-to-use and easy-to-understand framework called *Hamlets* for the development of Web-based applications (see Resources for more information on Hamlets). The framework is the result of a radical software simplification effort; thus, you should be able to understand it quickly.

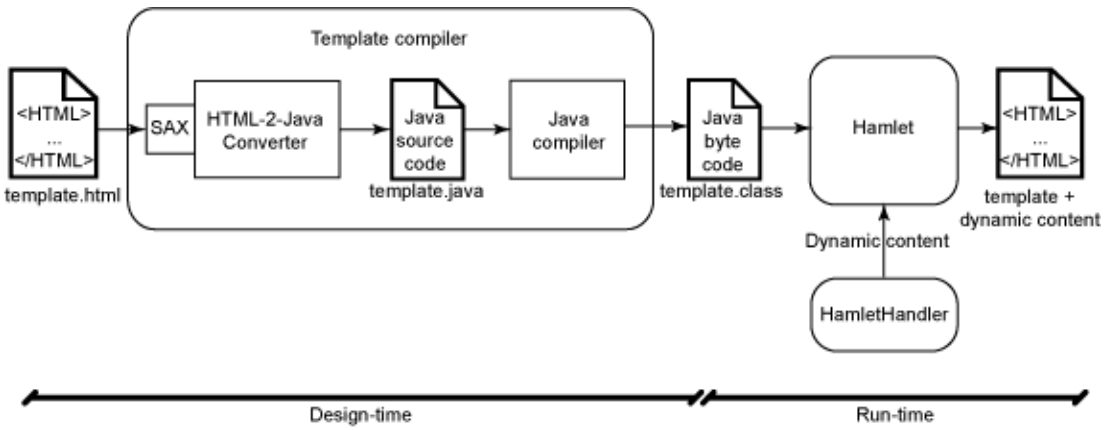
A Hamlet is a Java servlet extension that uses the Simple API for XML (SAX) to read template files. While a template file is being read, the Hamlet uses a small set of callback functions (implemented by a HamletHandler) to add dynamic content to those places in the template that are marked with special tags and IDs (see Figure 1).

Figure 1. A Hamlet uses SAX for reading content from a template file and calls a HamletHandler to add dynamic content



A template compiler can be used to accelerate Hamlets. The template compiler uses SAX to read template files and converts the content into Java source code. Subsequently, it calls the standard JDK Java compiler to generate Java bytecode. At runtime, this code is executed by the Hamlet and calls a HamletHandler to add dynamic content, as illustrated in Figure 2. For detailed information on how template compilation works, see Resources.

Figure 2. A template compiler converts the content of a template file into Java bytecode, which is executed by the Hamlet and calls a HamletHandler to add dynamic content



Hamlets now open source

The Hamlet framework has been available at IBM alphaWorks since September of 2005. On March 7, 2007, IBM decided to release the Hamlets code under a BSD license to help spread the technology. The framework is now hosted at Sourceforge; see Resources for a link.

The project name *Hamlets* used in this publication refers to an internal project for the development of a servlet-based framework for separation of content and presentation in Web-based applications. Version 1.4 of the Hamlets framework (officially known as the *IBM Servlet-Based Content Creation Framework*) is available from IBM alphaWorks (see Resources for a link).

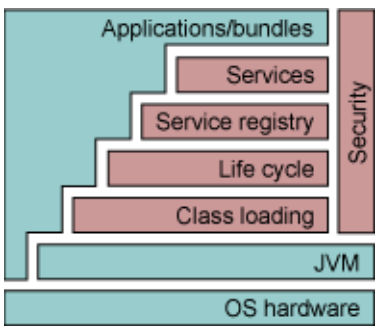
What is OSGi?

The OSGi Alliance (formerly known as the Open Services Gateway initiative) is an independent nonprofit organization formed in 1999. It defines open specifications for a Java technology-based service platform that can be remotely managed. The specification consists of two parts: the OSGi framework and a set of standard service definitions.

The OSGi framework implements a runtime environment for applications, called *bundles,* which execute together in a single Java Virtual Machine (JVM), as illustrated in Figure 3. Bundles can be installed, started, stopped, updated, and uninstalled dynamically without a reboot. A service registry allows bundles to be notified when services appear or disappear. Bundles can then adapt accordingly. The framework was originally targeted at Internet gateways with home automation applications, but it has also been successfully used in other domains, such as the automotive industry, consumer electronics, and in the desktop application space. For more detailed information, read the technical white paper entitled "About the OSGi Service Platform" (see Resources).

A large number of services and libraries have been developed for the OSGi framework. Hamlets are an ideal addition to the framework because they allow the creation of Web-based applications in which presentation (HTML code) and logic (Java code) are strictly separated.

Figure 3. OSGi architecture



The Hello bundle

The most basic OSGi application is shown in Listing 1. It consists of an `Activator` class that implements the `start()` and `stop()` methods of the `BundleActivator` interface. When the application is installed and started, the OSGi framework calls the `Activator`'s `start()` method to print "Hello World!" The `Activator`'s `stop()` method is invoked when the application is stopped.

Listing 1. Activator for the Hello bundle

```
package com.ibm.zurich.HelloBundle;


import org.osgi.framework.*;


public class Activator implements BundleActivator {


  public void start (BundleContext aContext) {
    System.out.println ("Hello World!");
  } // start


  public void stop (BundleContext aContext) {
  } // stop


} // Activator
```

The *bundle* is the delivery and development unit of OSGi applications. A bundle is a jar file that contains the application's bytecode and its resources. In addition, it contains a file -- the *manifest* -- that describes the bundle. Listing 2 describes the Hello bundle.

Listing 2. Manifest for the Hello bundle

```
Manifest-Version: 1.0
Bundle-Name: Hello Bundle
Bundle-SymbolicName: hellobundle
Bundle-Version: 1.0.0
Bundle-Description: This bundle prints 'Hello World!'.
Bundle-Vendor: Rene Pawlitzek
Bundle-Activator: com.ibm.zurich.HelloBundle.Activator
Bundle-Category: example
Import-Package: org.osgi.framework
```

The manifest contains the bundle's name, version, description, vendor, category, and *activator,* which is the name of the class that implements the `ActivatorBundle` interface. The manifest also names the import packages. Refer to the OSGi Service Platform specification (see Resources) for more information about the manifest file format and syntax.

Bundles can conveniently be built using Ant scripts. I used my favorite IDE and the script in Listing 3 to build hellobundle.jar.

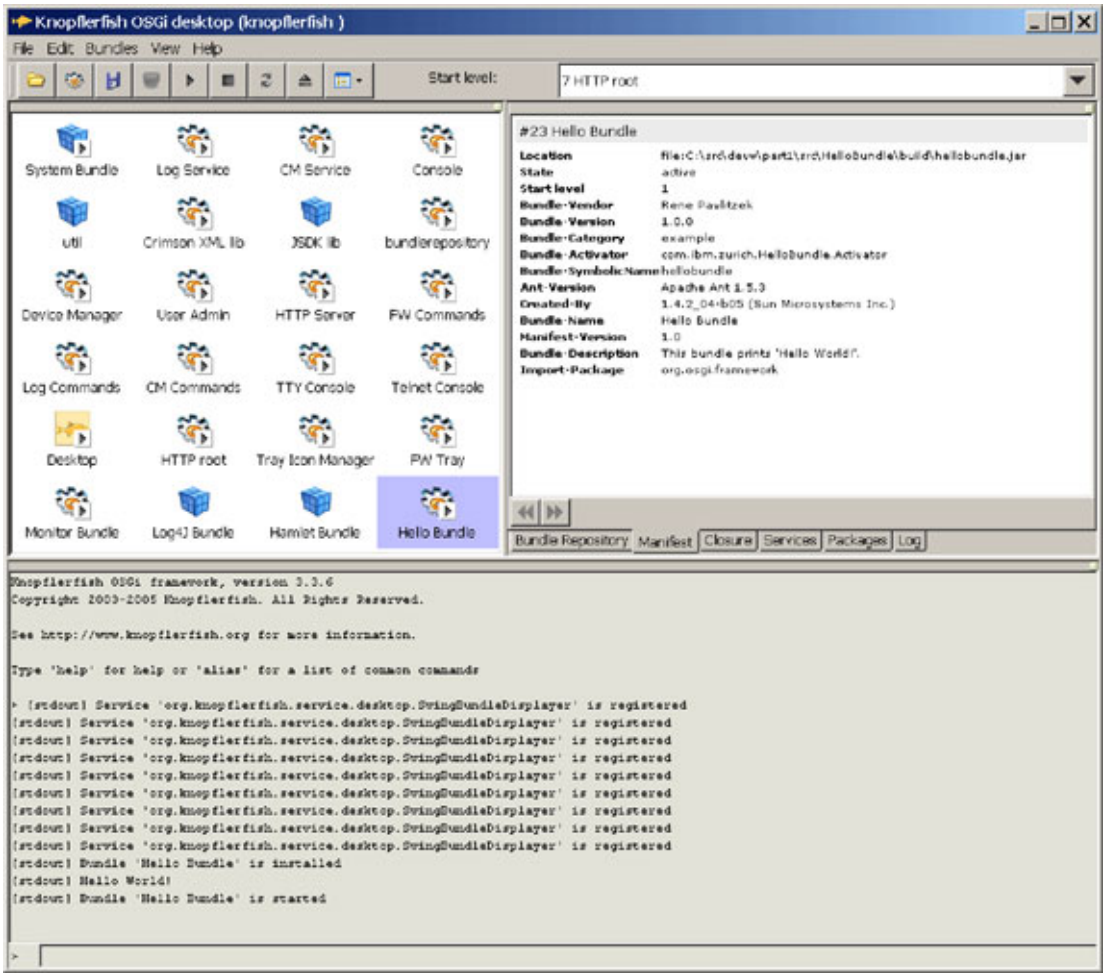
Listing 3. Ant script to build the Hello bundle

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

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


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


  <target name="compile">
    <javac destdir="./classes" srcdir="." />
  </target>


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


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


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


</project>
```

Prior to deployment, you can use WinZip or a similar tool to inspect hellobundle.jar. You should see the MANIFEST.MF and Activator.class files.

There are a number of implementations for the OSGi Service platform specification. Among them are Knopflerfish, Eclipse Equinox, and Apache Felix, three open source implementations (see Resources). I chose to use Kopflerfish 1.3.5 for all examples in this article. If you load hellobundle.jar from the Knopflerfish OSGi desktop, you should see the "Hello World!" printout in the console (the bottom part of the window shown in Figure 4).

Figure 4. Knopflerfish OSGi desktop running the Hello bundle



If you've come this far, you have already learned how to create, deploy, and run an OSGi application.

The Monitor bundle

Now I'll create a more useful program that monitors the life cycle of bundles and services within the OSGi framework. When an application registers a listener, it is notified by the framework when bundles are installed, started, stopped, or uninstalled. Similarly, an application can be notified when services (provided by bundles) are registered, unregistered, or modified. Well-behaved software never assumes that a service is available at all times. Instead, it registers a listener to receive notification events and adapts accordingly.

The `Activator` class for the Monitor application implements `BundleActivator` and two additional interfaces: `BundleListener` and `ServiceListener`. As you saw before with the Hello bundle, the OSGi framework calls the `start()` method when the Monitor bundle is started. Instead of printing "Hello world!", in this app you use the bundle context to register a bundle and a service listener with `addBundleListener()` and `addServiceListener()`. The listeners are removed in the `stop()` method with

`removeBundleListener()` and `removeServiceListener()` when the Monitor bundle is stopped.

The `serviceChanged()` and `bundleChanged()` methods (both implemented by the `Activator` class) receive notification events from the OSGi framework when a service is registered, unregistered, or modified, and when a bundle is installed, started, stopped, or uninstalled. The `ServiceEvent` and `BundleEvent` classes contain detailed information about a change, including a reference to the service or the bundle. `serviceChanged()` and `bundleChanged()` print this information. All this is illustrated in Listing 4.

Listing 4. Activator for the Monitor bundle

```
package com.ibm.zurich.MonitorBundle;


import org.osgi.framework.*;


public class Activator implements BundleActivator, ServiceListener, BundleListener {


  public void start (BundleContext aContext) throws Exception {
    aContext.addBundleListener (this);
    aContext.addServiceListener (this);
  } // start


  public void stop (BundleContext aContext) {
    aContext.removeServiceListener (this);
    aContext.removeBundleListener (this);
  } // stop


  /* ----- implementation of ServiceListener ----- */


  public void serviceChanged (ServiceEvent aEvent) {
    String action = null;
    switch (aEvent.getType ()) {
      case ServiceEvent.MODIFIED:
        action = "modified";
        break;
      case ServiceEvent.REGISTERED:
        action = "registered";
        break;
      case ServiceEvent.UNREGISTERING:
        action = "unregistered";
        break;
    } // switch
    if (action != null) {
      ServiceReference ref = aEvent.getServiceReference ();
      String classes[] = (String[]) ref.getProperty ("objectClass");
      System.out.println ("Service '" + classes[0] + "' is " + action);
    } // if
  } // serviceChanged


  /* ----- implementation of BundleListener ----- */


  public void bundleChanged (BundleEvent aEvent) {
    String action = null;
```

```
    switch (aEvent.getType ()) {
      case BundleEvent.INSTALLED:
        action = "installed";
        break;
      case BundleEvent.STARTED:
        action = "started";
        break;
      case BundleEvent.STOPPED:
        action = "stopped";
        break;
      case BundleEvent.UNINSTALLED:
        action = "uninstalled";
        break;
    } // switch
    if (action != null) {
      Bundle bundle = aEvent.getBundle ();
      String name = (String) bundle.getHeaders().get (Constants.BUNDLE_NAME);
      System.out.println ("Bundle '" + name + "' is " + action);
    } // if
  } // bundleChanged


} // Activator
```

The manifest and Ant build script for the Monitor bundle are almost identical to their Hello bundle counterparts, and thus are not illustrated here.

The HelloServlet bundle

The OSGi framework features an HTTP service that allows the execution of servlets. The next bundle provides and registers a Hello servlet, which then becomes available through HTTP. The servlet's output (the "Hello world!" greeting) will be displayed in a browser. I will use a `ServiceTracker` object (which is based on the previously presented `ServiceListener` interface) to receive notification events when the HTTP service changes.

The `Activator` class for the HelloServlet bundle implements the `start()` and `stop()` methods of the `BundleActivator` interface. In `start()`, a HelloServlet instance is created and stored for later use in an instance variable (`servlet`). Furthermore, a `ServiceTracker` object is instantiated and opened to track the HTTP service. The `ServiceTracker` constructor is called with the bundle context (`context`), the name of the service to track (`HttpService.class.getName()`), and the current `Activator` instance (`this`). The `Activator` class implements the three methods of the `ServiceTrackerCustomizer` interface (`addingService()`, `modifiedService()`, and `removedService()`) that are invoked by the service tracker when the HTTP service is added, modified, or removed. This is illustrated in Listing 5.

Listing 5. Activator for the HelloServlet bundle

```
package com.ibm.zurich.HelloServletBundle;


import org.osgi.framework.*;
import org.osgi.util.tracker.*;
import org.osgi.service.http.*;
```

```
public class Activator implements BundleActivator, ServiceTrackerCustomizer {


  public static BundleContext context = null;


  private  HttpService   httpService;
  private  HelloServlet  servlet;


  public void start (BundleContext aContext) {
    context = aContext;
    servlet = new HelloServlet ();
    ServiceTracker tracker =
      new ServiceTracker (context, HttpService.class.getName (), this);
    tracker.open ();
  } // start


  public void stop (BundleContext aContext) {
    unregisterServlet (httpService);
    servlet = null;
    context = null;
  } // stop


  private void registerServlet (HttpService aHttpService) {
    try {
      if (aHttpService != null)
        aHttpService.registerServlet ("/hello", servlet, null, null);
        System.out.println ("Registered /hello servlet");
    } catch (Exception e) {
      System.out.println ("Unable to register servlet");
    } // try
  } // registerServlet


  private void unregisterServlet (HttpService aHttpService) {
    try {
      if (aHttpService != null)
        aHttpService.unregister ("/hello");
        System.out.println ("Unregistered /hello servlet");
    } catch (Exception e) {
      System.out.println ("Unable to unregister servlet");
    } // try
  } // unregisterServlet


  /* ----- implementation of ServiceTrackerCustomizer ----- */


  public Object addingService (ServiceReference aRef) {
    httpService = (HttpService) context.getService (aRef);
    registerServlet (httpService);
    return httpService;
  } // addingService


  public void modifiedService (ServiceReference aRef, Object aObj) {
    httpService = (HttpService) context.getService (aRef);
    registerServlet (httpService);
  } // modifiedService


  public void removedService (ServiceReference aRef, Object aObj) {
  } // removedService
```

```
} // Activator
```

The tracker object will eventually call the `addingService()` method to inform you that the HTTP service is available. Take this opportunity to register the Hello servlet that was previously created in `start()`. The servlet is now ready for use and locally accessible at the URL http://localhost:8080/hello, as you can see in Figure 5.

Figure 5. HelloServlet bundle saying hello to the world

The servlet's output is generated by a sequence of `println()` statements in the `doGet()` method of the `HelloServlet` class, as you can see in Listing 6. Embedding HTML in Java code is acceptable for very small programs. However, for larger applications, it results in severe maintenance problems. Hamlets solve this problem, as you will see in the Coffee Machine example later in this article.

Listing 6. Hello servlet

```
package com.ibm.zurich.HelloServletBundle;


import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;


public class HelloServlet extends HttpServlet {


  public void doGet (HttpServletRequest req, HttpServletResponse res) throws
    ServletException {

    try {
```

```
      res.setContentType ("text/html");
      PrintWriter out = res.getWriter ();
      out.println ("<HTML>");
      out.println ("<HEAD><TITLE>Hello world!</TITLE></HEAD>");
      out.println ("<BODY>");
      out.println ("Hello world!");
      out.println ("</BODY>");
      out.println ("</HTML>");
      out.flush ();
    } catch (Exception e) {
      throw new ServletException (e);
    } // try

  } // doGet


} // HelloServlet
```

Now assume that the current HTTP service is replaced with a more sophisticated implementation. Obviously, the tracker object will call the `removedService()` method followed by the `addingService()` method. It is not necessary to unregister the Hello servlet in `removedService()`, but you must re-register the servlet with the new improved HTTP service to make it available again. Similarly, you re-register the Hello servlet in `modifiedService()` whenever the HTTP service is modified. The Hello servlet is unregistered in the `stop()` method, which is called when the bundle is stopped.

The Ant build script and the manifest for the HelloServlet bundle are mostly identical to their counterparts for the other bundles that you saw above. Note that the manifest, shown in Listing 7, names a few additional import packages.

Listing 7. Hello servlet

```
Manifest-Version: 1.0
Bundle-Name: Hello Servlet Bundle
Bundle-SymbolicName: helloservletbundle
Bundle-Version: 1.0.0
Bundle-Description: This bundles provides a servlet to print 'Hello World!'.
Bundle-Vendor: Rene Pawlitzek
Bundle-Activator: com.ibm.zurich.HelloServletBundle.Activator
Bundle-Category: example
Import-Package: org.osgi.framework, org.osgi.util.tracker, org.osgi.service.http,
javax.servlet, javax.servlet.http
```

The Hamlet bundle

Next, you'll create the Hamlet bundle to make the Hamlet framework (which comes in the form of a jar library called hamlet.jar) available to the OSGi infrastructure. This allows you to separate HTML from Java code in your bundles. No activator class is required for this bundle. The Ant build script, shown in Listing 8, simply packages hamlet.jar and MANIFEST.MF into a jar file called hamletbundle.jar.

Listing 8. Ant script to build the Hamlet bundle

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

<project name="HamletBundle" default="all">
```

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


  <target name="jar">
    <jar basedir="./jar" jarfile ="./build/hamletbundle.jar"
      includes="hamlet.jar" manifest="./meta-inf/MANIFEST.MF" />
  </target>


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


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


</project>
```

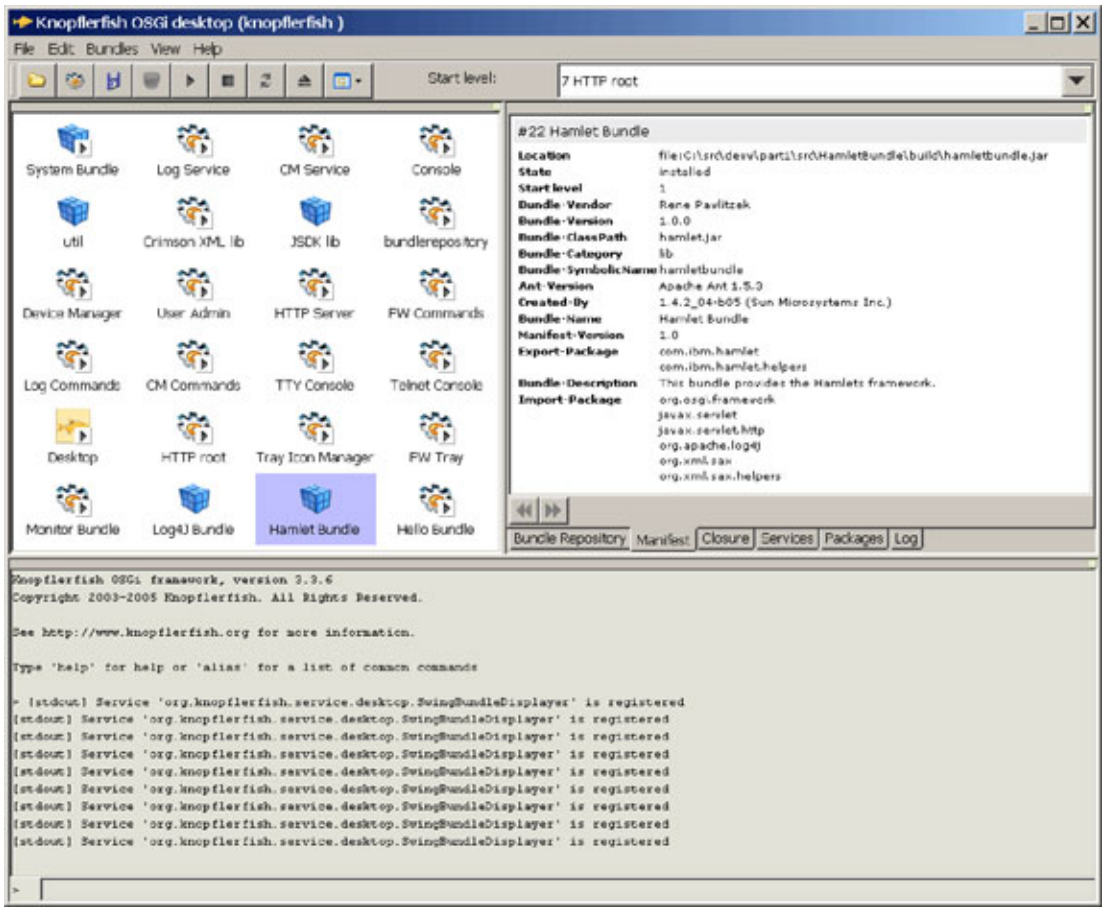
The manifest for hamletbundle.jar, shown in Listing 9, names not only the import packages but also the export packages. Note that the bundle category is now `lib`.

Listing 9. Manifest for the Hamlet bundle

```
Manifest-Version: 1.0
Bundle-Name: Hamlet Bundle
Bundle-SymbolicName: hamletbundle
Bundle-Version: 1.0.0
Bundle-Description: This bundle provides the Hamlets framework.
Bundle-Vendor: Rene Pawlitzek
Bundle-ClassPath: hamlet.jar
Bundle-Category: lib
Export-Package: com.ibm.hamlet, com.ibm.hamlet.helpers
Import-Package: org.osgi.framework, javax.servlet, javax.servlet.http, org.apache.log4j,
org.xml.sax, org.xml.sax.helpers
```

The Hamlet bundle shows up in the bundle view when you load it from the Knopflerfish OSGi desktop, as you can see in Figure 6.

Figure 6. Knopflerfish OSGi desktop showing the Hamlet bundle



At present, the Hamlet framework depends on Log4j for logging services. Thus, you need to build and install a Log4j bundle (containing log4j.jar); this can be accomplished in the same way in which you build the Hamlet bundle. (See Resources for a link to Log4j.)

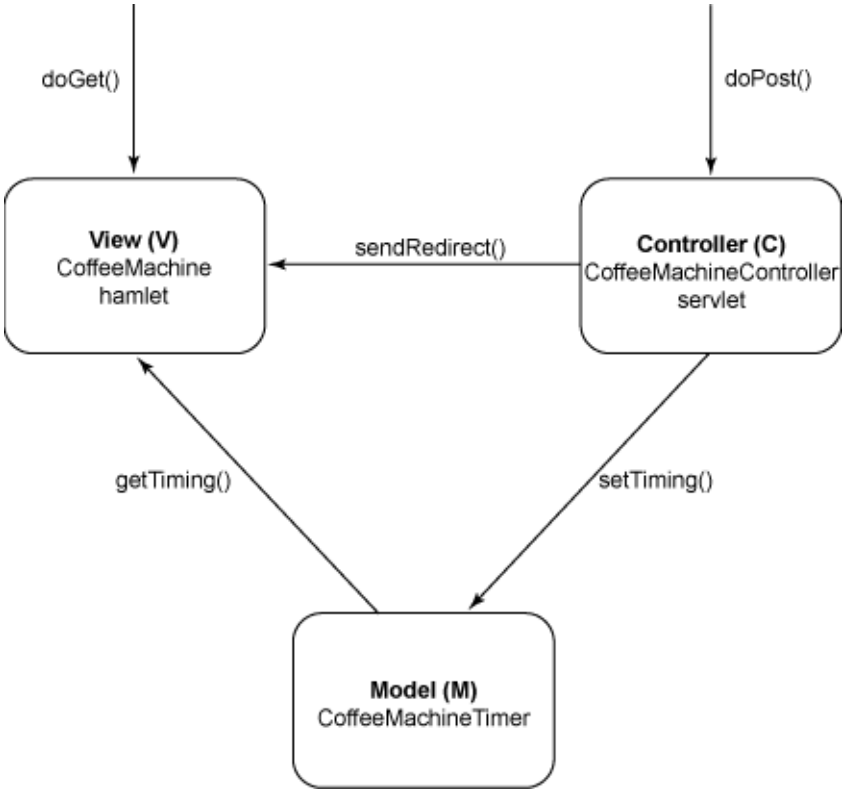
The Coffee Machine bundle

In this final example, you'll develop a bundle that provides a Web-based user interface for a network-enabled coffee machine. You can set the coffee machine's timer from anywhere in the world using a browser. You're using Hamlets to separate presentation and logic, so you will not see any HTML in the Java code, and vice versa. Furthermore, you're using the model-view-controller (MVC) design pattern for the implementation.

The `Activator` class, illustrated in Listing 10, again implements the `BundleActivator` interface with its two methods `start()` and `stop()`. The OSGi framework initially calls `start()` where logging is configured with `BasicConfigurator.configure()` and the coffee machine model (M) is initialized with `CoffeeMachineTimer.getTimer().start()`. Next, the coffee machine's view (V) and controller (C) are created. The view, an instance of `CoffeeMachine`, is a Hamlet, and the controller, an instance of `CoffeeMachineController`, is a servlet

(see Figure 7). Finally, as in the HelloServlet bundle, a `ServiceTracker` object is instantiated and opened to track the life cycle of the HTTP service. Whenever the HTTP service changes, the tracker calls `addingService()`, `modifiedService()`, and `removedService()`, which are part of the `ServiceTrackerCustomizer` interface that `Activator` implements.

Figure 7. MVC design pattern for CoffeeMachine bundle



Listing 10. Activator for the CoffeeMachine bundle

```
package com.ibm.zurich.CoffeeMachineBundle;


import javax.servlet.http.*;
import org.apache.log4j.*;
import org.osgi.framework.*;
import org.osgi.util.tracker.*;
import org.osgi.service.http.*;


public class Activator implements BundleActivator, ServiceTrackerCustomizer {


  public static BundleContext context = null;


  // log4j
  private static Category category = Category.getInstance (Activator.class.getName ());


  private  HttpService    httpService;
  private  HttpServlet    controller;
  private  CoffeeMachine  view;
```

```
public void start (BundleContext aContext) {
  context = aContext;
  BasicConfigurator.resetConfiguration ();
  BasicConfigurator.configure ();
  CoffeeMachineTimer.getTimer().start ();
  view = new CoffeeMachine ();
  controller = new CoffeeMachineController ();
  ServiceTracker tracker =
    new ServiceTracker (context, HttpService.class.getName (), this);
  tracker.open ();
} // start


public void stop (BundleContext aContext) {
  unregister (httpService);
  CoffeeMachineTimer.getTimer().stop ();
  controller = null;
  view = null;
  context = null;
} // stop


private void register (HttpService aHttpService) {
  try {
    if (aHttpService != null) {
      aHttpService.registerServlet ("/CoffeeMachine", view, null, null);
      category.debug ("Registered '/CoffeeMachine' hamlet");
      aHttpService.registerServlet ("/CoffeeMachineController",
        controller, null, null);
      category.debug ("Registered '/CoffeeMachineController' servlet");
      HttpContext httpContext = new CoffeeMachineContext (context);
      aHttpService.registerResources ("/Include", "/Resources", httpContext);
      category.debug ("Registered '/Include' resources");
    } // if
  } catch (Exception e) {
    category.error ("Unable to register", e);
  } // try
} // register


private void unregister (HttpService aHttpService) {
  try {
    if (aHttpService != null) {
      aHttpService.unregister ("/CoffeeMachine");
      category.debug ("Unregistered '/CoffeeMachine' hamlet");
      aHttpService.unregister ("/CoffeeMachineController");
      category.debug ("Unregistered '/CoffeeMachineController' servlet");
      aHttpService.unregister ("/Include");
      category.debug ("Unregistered '/Include' resources");
    } // if
  } catch (Exception e) {
    category.error ("Unable to unregister", e);
  } // try
} // unregister


/* ----- implementation of ServiceTrackerCustomizer ----- */


public Object addingService (ServiceReference aRef) {
  category.debug ("adding service");
  httpService = (HttpService) context.getService (aRef);
  register (httpService);
  return httpService;
} // addingService
```

```
  public void modifiedService (ServiceReference aRef, Object aObj) {
    category.debug ("modified service");
    httpService = (HttpService) context.getService (aRef);
    register (httpService);
  } // modifiedService


  public void removedService (ServiceReference aRef, Object aObj) {
    category.debug ("removed service");
  } // removedService


} // Activator
```

At some point, the tracker object invokes `addingService()` to inform you about the availability of the HTTP service. Subsequently, `register()` is called to register with the HTTP service the Hamlet providing the view, the servlet providing the controller, and the resources (the coffee machine bitmap and the cascading style sheet used to format the output). The registration of the resources with `registerResources()` requires an HTTP context (`httpContext`). The `CoffeeMachineContext` class, shown in Listing 11, provides the HTTP context for the coffee machine resources. It implements the three methods of the `HttpContext` interface: `handleSecurity()`, `getMimeType()`, and `getResource()`. When a particular resource is requested (for example, the coffee machine bitmap with `<IMG SRC="Include/coffee_machine.jpg" />`), the `getResource()` method of the `CoffeeMachineContext` class returns a URL to the resource in the bundle (for example, `bundle://36/Resources/coffee_machine.jpg`).

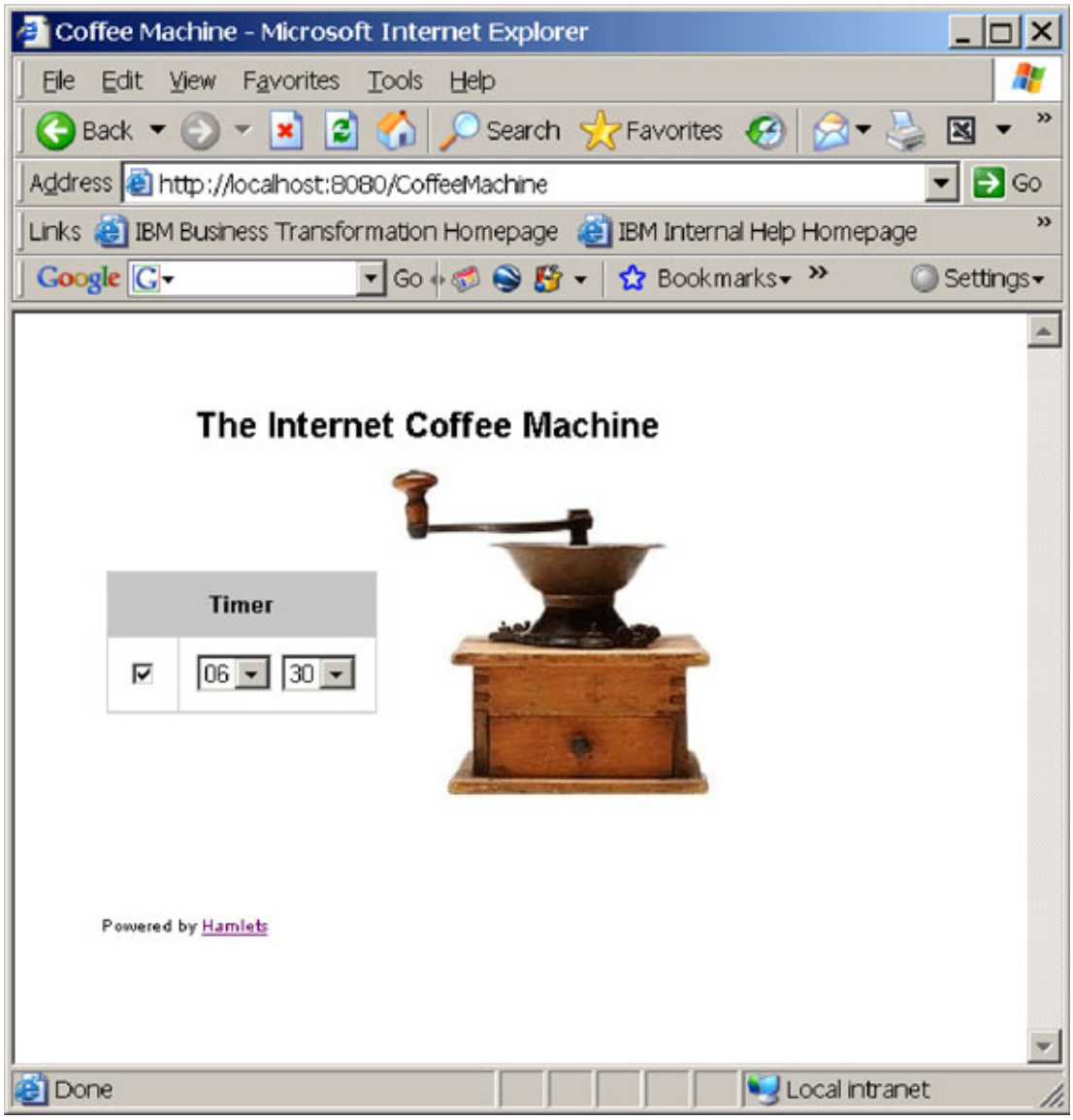
Listing 11. CoffeeMachine context

```
package com.ibm.zurich.CoffeeMachineBundle;


import java.net.*;
import javax.servlet.http.*;
import org.osgi.framework.*;
import org.osgi.service.http.*;


public class CoffeeMachineContext implements HttpContext {


  private  BundleContext  context;


  public CoffeeMachineContext (BundleContext context) {
    this.context = context;
  } // CoffeeMachineContext


  public boolean handleSecurity (HttpServletRequest req, HttpServletResponse res) {
    return true;
  } // handleSecurity


  public String getMimeType (String name) {
    return null;
  } // getMimeType
```

```
  public URL getResource (String path) {
    return context.getBundle().getResource (path);
  } // getResource


} // CoffeeMachineContext
```

The Web-based user interface of the coffee machine can now be accessed with a browser at the URL http://localhost:8080/CoffeeMachine, as Figure 8 shows.

Figure 8. CoffeeMachine bundle, providing a Web-based front-end for a coffee machine



The tracker object will call the `removedService()` method, followed by the `addingService()` method when the HTTP service is replaced. As was the case with the HelloServlet bundle, it is necessary to re-register the controller servlet, the

resources, and the Hamlet that provides the view. Re-registration is also required when `modifiedService()` is called -- when the HTTP is modified, in other words.

The `Activator`'s `stop()` method is invoked when the CoffeeMachine bundle is stopped. Here you unregister the Hamlet, servlet, and resources and clean up the model (M) with `CoffeeMachineTimer.getTimer().stop()`.

So far, not much has been different from the previous HelloServlet example. However, the CoffeeMachine bundle does not use `println()` statements in the `doGet()` method of a servlet to produce its output. Instead, it uses a Hamlet that enables the separation of HTML and Java code.

The `doGet()` method is invoked when the coffee machine is accessed through the network. You retrieve the current timer setting from the data model with `model.getTiming()` and create an instance (`handler`) of `CoffeeMachineHandler` (which is a private class of `CoffeeMachine` that extends `HamletHandler`). Next, the `serveDoc (req, res, template, handler)` method is called. It executes a compiled template (referenced by `template`) to produce the coffee machine's user interface (the HTML code) and calls the `handler`'s `getElementAttributes()` method to add the current timer setting (the `CHECKED` and `SELECTED` attributes) with `Helpers.getAttributes()`. The Java bytecode for the compiled template (CoffeeMachineTemplate.class) is generated at design-time from the XHTML template (CoffeeMachineTemplate.html) by the template compiler using Ant. You load this code during the creation of the CoffeeMachine hamlet with `Class.forName("CoffeeMachineTemplate.class")` in the `init()` method. For detailed information on how `getElementAttributes()` works, see Resources.

Listing 12. The CoffeeMachine hamlet provides the coffee machine's view

```
package com.ibm.zurich.CoffeeMachineBundle;


import javax.servlet.*;
import javax.servlet.http.*;
import com.ibm.hamlet.*;
import com.ibm.hamlet.helpers.*;
import org.apache.log4j.*;
import org.xml.sax.*;


public class CoffeeMachine extends Hamlet {


    // log4j
    private static Category category =
      Category.getInstance (CoffeeMachine.class.getName ());


    private Class template;


    private static class CoffeeMachineHandler extends HamletHandler {


      private CoffeeMachineTiming timing;
```

```
    public CoffeeMachineHandler (Hamlet hamlet, CoffeeMachineTiming aTiming) {
      super (hamlet);
      timing = aTiming;
    } // CoffeeMachineHandler


    public Attributes getElementAttributes (String id, String name, Attributes atts)
      throws Exception {

      if (id.equals ("Checkbox")) {
        if (timing.isSet ())
          atts = Helpers.getAttributes (atts, "CHECKED", "Checked");
      } else if (id.equals ("StartHour")) {
        String hour = atts.getValue ("Hour");
        if (hour.equals ("" + timing.getHour ()))
          atts = Helpers.getAttributes (atts, "SELECTED", "Selected");
      } else if (id.equals ("StartMinute")) {
        String minute = atts.getValue ("Minute");
        if (minute.equals ("" + timing.getMinute ()))
          atts = Helpers.getAttributes (atts, "SELECTED", "Selected");
      } // if
      return atts;

    } // getElementAttributes


  } // CoffeeMachineHandler



  public void init () throws ServletException {
    try {
      category.debug ("init");
      template = Class.forName ("CoffeeMachineTemplate");
    } catch (Exception e) {
      category.error ("", e);
      throw new ServletException (e);
    } // try
  } // init


  public void doGet (HttpServletRequest req, HttpServletResponse res)
    throws ServletException {

    try {
      category.debug ("doGet");
      CoffeeMachineTimer model = CoffeeMachineTimer.getTimer ();
      CoffeeMachineTiming timing = model.getTiming();
      HamletHandler handler = new CoffeeMachineHandler (this, timing);
      serveDoc (req, res, template, handler);
    } catch (Exception e) {
      category.error ("", e);
      throw new ServletException (e);
    } // try

  } // doGet


  public void destroy () {
    category.debug ("destroy");
  } // destroy


} // CoffeeMachine
```

Listing 13 shows the coffee machine's HTML code (CoffeeMachineTemplate.html) that the template compiler converts into Java bytecode (CoffeeMachineTemplate.class) at design-time. For detailed information on how template compilation works, see Resources.

Listing 13. CoffeeMachineTemplate.html

```
<!DOCTYPE CoffeeMachine [ <!ENTITY nbsp " "> ]>
<HTML>
  <HEAD>
    <TITLE>Coffee Machine</TITLE>
    <LINK REL="stylesheet" TYPE="text/css" HREF="Include/View.css" MEDIA="all" />
  </HEAD>
  <BODY>

    <FORM NAME="ui" ACTION="CoffeeMachineController" METHOD="POST">

    <TABLE BORDER="0">
    <TR>

      <TD>
      <TABLE CLASS="report" CELLPADDING="10" CELLSPACING="1">

      <TR CLASS="odd">
        <TH CLASS="report" COLSPAN="2">Coffee Machine</TH>
      </TR>

      <TR CLASS="even">
        <TD CLASS="report"><INPUT ID="Checkbox" CLASS="Checkbox"
          TYPE="Checkbox" NAME="Checkbox" onClick="submit();" />
        </TD>
        <TD>
          <SELECT CLASS="report" size="1" NAME="Hour" onChange="submit();">
            <OPTION ID="StartHour" Hour="0">00</OPTION>
            <OPTION ID="StartHour" Hour="1">01</OPTION>
            <OPTION ID="StartHour" Hour="2">02</OPTION>
            <OPTION ID="StartHour" Hour="3">03</OPTION>
            <OPTION ID="StartHour" Hour="4">04</OPTION>
            <OPTION ID="StartHour" Hour="5">05</OPTION>
            <OPTION ID="StartHour" Hour="6">06</OPTION>
            <OPTION ID="StartHour" Hour="7">07</OPTION>
            <OPTION ID="StartHour" Hour="8">08</OPTION>
            <OPTION ID="StartHour" Hour="9">09</OPTION>
            <OPTION ID="StartHour" Hour="10">10</OPTION>
            <OPTION ID="StartHour" Hour="11">11</OPTION>
            <OPTION ID="StartHour" Hour="12">12</OPTION>
            <OPTION ID="StartHour" Hour="13">13</OPTION>
            <OPTION ID="StartHour" Hour="14">14</OPTION>
            <OPTION ID="StartHour" Hour="15">15</OPTION>
            <OPTION ID="StartHour" Hour="16">16</OPTION>
            <OPTION ID="StartHour" Hour="17">17</OPTION>
            <OPTION ID="StartHour" Hour="18">18</OPTION>
            <OPTION ID="StartHour" Hour="19">19</OPTION>
            <OPTION ID="StartHour" Hour="20">20</OPTION>
            <OPTION ID="StartHour" Hour="21">21</OPTION>
            <OPTION ID="StartHour" Hour="22">22</OPTION>
            <OPTION ID="StartHour" Hour="23">23</OPTION>
          </SELECT>
          <SELECT CLASS="report" size="1" NAME="Minute" onChange="submit();">
            <OPTION ID="StartMinute" Minute="0">00</OPTION>
            <OPTION ID="StartMinute" Minute="5">05</OPTION>
            <OPTION ID="StartMinute" Minute="10">10</OPTION>
            <OPTION ID="StartMinute" Minute="15">15</OPTION>
            <OPTION ID="StartMinute" Minute="20">20</OPTION>
            <OPTION ID="StartMinute" Minute="25">25</OPTION>
```

```
            <OPTION ID="StartMinute" Minute="30">30</OPTION>
            <OPTION ID="StartMinute" Minute="35">35</OPTION>
            <OPTION ID="StartMinute" Minute="40">40</OPTION>
            <OPTION ID="StartMinute" Minute="45">45</OPTION>
            <OPTION ID="StartMinute" Minute="50">50</OPTION>
            <OPTION ID="StartMinute" Minute="55">55</OPTION>
          </SELECT>
        </TD>
      </TR>

      </TABLE>
      </TD>

      <TD>
        <IMG SRC="Include/coffee_machine.jpg" />
      </TD>

    </TR>
    </TABLE>

    <BR/>
    <DIV CLASS="link">
      Powered by <A HREF="http://www.alphaworks.ibm.com/tech/hamlets">Hamlets</A>
    </DIV>

    </FORM>
  </BODY>
</HTML>
```

Now assume that the user wants to set the timer of the coffee machine. As you can see in Listing 14, the user interface does not need to offer a submit button, because all changes to the checkbox to activate or deactivate the timer and the drop-down lists for selecting a timer setting are immediately submitted to the CoffeeMachineController servlet (`<FORM NAME="ui" ACTION="CoffeeMachineController" METHOD="POST">`) with the help of some JavaScript code (`onClick="submit();"`) in the `<INPUT>` tag and the two `<SELECT>` tags.

Listing 14. CoffeeMachineController servlet providing the coffee machine's controller (C)

```
package com.ibm.zurich.CoffeeMachineBundle;


import javax.servlet.*;
import javax.servlet.http.*;
import org.apache.log4j.*;


public class CoffeeMachineController extends HttpServlet {


  // log4j
  private static Category category =
    Category.getInstance (CoffeeMachineController.class.getName ());


  public void init () {
    category.debug ("init");
  } // init
```

```
  public void doPost (HttpServletRequest req, HttpServletResponse res)
    throws ServletException {

    try {
      category.debug ("doPost");
      String set = req.getParameter ("Checkbox");
      String hour = req.getParameter ("Hour");
      String minute = req.getParameter ("Minute");
      CoffeeMachineTimer model = CoffeeMachineTimer.getTimer ();
      CoffeeMachineTiming timing = new CoffeeMachineTiming ();
      timing.set ("on".equals (set));
      timing.setHour (hour);
      timing.setMinute (minute);
      model.setTiming (timing);
      res.sendRedirect ("CoffeeMachine");
    } catch (Exception e) {
      category.error ("", e);
      throw new ServletException (e);
    } // try

   } // doPost


   public void destroy () {
     category.debug ("destroy");
   } // destroy


} // CoffeeMachineController
```

The `CoffeeMachineController`'s `doPost()` method is invoked whenever a timer setting change occurs. After retrieving the submitted parameter values with `getParameter()`, you create an instance (`timing`) of `CoffeeMachineTiming` and use its setter methods to set hour, minute, and timer status (on or off). Next, you call `setTiming()` on the coffee machine model (M) -- obtained with `CoffeeMachineTimer.getTimer()` -- to set the new timer setting. And finally, you redirect to the CoffeeMachine hamlet to refresh the view. Thus, the logic to process the input (the servlet) and the logic to generate the view (the Hamlet) are nicely separated (as described by the MVC design pattern).

The `CoffeeMachineTiming` class is straightforward and shown in Listing 15. It represents coffee machine timer settings.

Listing 15. CoffeeMachineTiming class representing timer settings

```
package com.ibm.zurich.CoffeeMachineBundle;


public class CoffeeMachineTiming {


  private  boolean  set;
  private  int      hour;
  private  int      minute;


  public CoffeeMachineTiming () {
    set = false;
    hour = 12;
    minute = 0;
  } // CoffeeMachineTiming
```

```
  public CoffeeMachineTiming (CoffeeMachineTiming aTimingDesc) {
    set = aTimingDesc.set;
    hour = aTimingDesc.hour;
    minute = aTimingDesc.minute;
  } // CoffeeMachineTiming


  public void set (boolean b) {
    set = b;
  } // set


  public boolean isSet () {
    return set;
  } // isSet


  public void setHour (String aHour) {
     hour = Integer.parseInt (aHour);
  } // setHour


  public int getHour () {
    return hour;
  } // getHour


  public void setMinute (String aMinute) {
    minute = Integer.parseInt (aMinute);
  } // setMinute


  public int getMinute () {
    return minute;
  } // getMinute


  public String toString () {
    StringBuffer buf = new StringBuffer ();
    buf.append ("Set: ");
    buf.append (set);
    buf.append (", ");
    buf.append ("Hour: ");
    buf.append (hour);
    buf.append (", ");
    buf.append ("Minute: ");
    buf.append (minute);
    return buf.toString ();
  } // toString


  public static void main (String args[]) {
    CoffeeMachineTiming timing = new CoffeeMachineTiming ();
    System.out.println (timing.toString ());
  } // main


} // CoffeeMachineTiming
```

The `CoffeeMachineTimer` class, illustrated in Listing 16, provides a mechanism to trigger the coffee brewing process. It implements the `run()` method of the `Runnable` interface and uses a thread to check if it is time to start making coffee. Currently, once the timer has counted down, the program just prints the message

"Brewing coffee..." However, you could easily replace the `println()` statement with functionality that interacts with a real coffee machine (I am looking forward to hearing from anyone who'd like to try this!). Only a single instance of `CoffeeMachineTimer` can be created (since we are using the singleton pattern) with the static method `getTimer()`. The `start()` and `stop()` methods are used to start and stop the timer, and `getTiming()` and `setTiming()` allow you to get the current timer setting and to set a new timer setting. Note that synchronization is required to avoid race conditions.

Listing 16. CoffeeMachineTimer class provides the coffee machine's model (M)

```
package com.ibm.zurich.CoffeeMachineBundle;


import java.util.*;
import org.apache.log4j.*;


class CoffeeMachineTimer implements Runnable {


  // log4j
  private static Category category =
    Category.getInstance (CoffeeMachineTimer.class.getName ());


  private static CoffeeMachineTimer timer = null;


  private  boolean              abort;
  private  Thread               t;
  private  CoffeeMachineTiming  timing;


  CoffeeMachineTimer () {
    t = null;
    abort = false;
    timing = new CoffeeMachineTiming ();
  } // CoffeeMachineTimer


  public synchronized void start () {
    if (t == null) {
      category.debug ("Starting timer...");
      t = new Thread (this);
      t.start ();
    } // if
  } // start


  public synchronized void stop () {
    if (t != null) {
      category.debug ("Stopping timer...");
      abort = true;
      try {
        t.join ();
      } catch (Exception e) {
        category.error ("", e);
      } // try
    } // if
  } // stop
```

```
  public synchronized CoffeeMachineTiming getTiming () {
    return new CoffeeMachineTiming (timing);
  } // getTiming


  public synchronized void setTiming (CoffeeMachineTiming aTiming) {
    timing = new CoffeeMachineTiming (aTiming);
    System.out.println (timing.toString ());
  } // setTiming


  public void run () {
    category.debug ("Running...");
    while (!abort) {
      try {

        CoffeeMachineTiming timing = getTiming ();
        if (timing.isSet ()) {

          long curTime = System.currentTimeMillis ();
          Calendar cal = new GregorianCalendar ();
          cal.setTimeInMillis (curTime);
          cal.set (Calendar.HOUR_OF_DAY, timing.getHour ());
          cal.set (Calendar.MINUTE, timing.getMinute ());
          cal.set (Calendar.SECOND, 0);
          long setTime = cal.getTimeInMillis ();

          Date d1 = new Date (curTime);
          Date d2 = new Date (setTime);
          System.out.println (d1.toString () + ", " + d2.toString ());

          if ((curTime / 1000L) == (setTime / 1000L))
            System.out.println ("Brewing coffee ... ");

        } // if

        Thread.sleep (1000);

      } catch (Exception e) {
        category.error ("", e);
      } // try
    } // while
    category.debug ("Finished.");
  } // run


  public static synchronized CoffeeMachineTimer getTimer () {
    if (timer == null)
      timer = new CoffeeMachineTimer ();
    return timer;
  } // getTimer


} // CoffeeMachineTimer
```

The bundle's manifest is shown in Listing 17. It names all necessary import packages.

Listing 17. Manifest for the CoffeeMachine bundle

```
Manifest-Version: 1.0
Bundle-Name: Coffee Machine Bundle
Bundle-SymbolicName: coffeemachinebundle
Bundle-Version: 1.0.0
Bundle-Description: This bundles provides a network-enabled coffee machine.
Bundle-Vendor: Rene Pawlitzek
Bundle-Activator: com.ibm.zurich.CoffeeMachineBundle.Activator
Bundle-Category: example
Import-Package: org.osgi.framework, org.osgi.util.tracker, org.osgi.service.http,
javax.servlet, javax.servlet.http, com.ibm.hamlet,
com.ibm.hamlet.helpers, org.apache.log4j, org.xml.sax, org.xml.sax.helpers
```

And finally, Listing 18 illustrates the Ant script to build the CoffeeMachine bundle. It contains a task `template` to invoke the template compiler that converts all *Template.html files (CoffeeMachineTemplate.html, in this case) into Java classes (CoffeeMachineTemplate.class). Hamlets generate user interfaces by executing these Java classes. Note that the *Template.html files do not need to be included in the bundle. They are only required for template compilation. The application's resources (all gif, jpg, and css files) are part of the bundle and located in the Resources subdirectory.

Listing 18. Ant script to build the CoffeeMachine bundle

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

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


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


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


  <target name="compile">
    <javac destdir="./classes" srcdir="." />
  </target>


  <target name="template">
    <taskdef name="hamletc" classname="com.ibm.hamlet.ant.CompileTask" />
    <hamletc destdir="./classes">
      <fileset dir="./com/ibm/zurich/CoffeeMachineBundle">
        <include name="*Template.html" />
      </fileset>
    </hamletc>
  </target>


  <target name="copy">
    <copy todir="./classes">
      <fileset dir="./com/ibm/zurich/CoffeeMachineBundle">
        <include name="*.html"/>
      </fileset>
    </copy>
    <copy todir="./classes/Resources">
```

```
      <fileset dir="./com/ibm/zurich/CoffeeMachineBundle/Include">
        <include name="*.gif"/>
        <include name="*.jpg"/>
        <include name="*.css"/>
      </fileset>
    </copy>
  </target>


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


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


</project>
```

This completes the network-enabled coffee machine example.

Conclusion

The OSGi framework is a Java technology-based service platform with remote management capabilities. It has been successfully used in a number of areas: home automation, the automotive industry, consumer electronics, and even desktop applications. Hamlets are an ideal addition to the OSGi framework because they enable the construction of Web-based applications for which the presentation layer and logic layer (providing the dynamic content) are completely separated. After a short introduction, this article explained step-by-step how to use Hamlets to create a network-enabled coffee machine whose timer can be remotely set using a browser.

Acknowledgments

I thank Chris Giblin and Rainer Hauser for their valuable comments and suggestions.

Resources

- "Introducing Hamlets," René Pawlitzek (developerWorks, March 2005): This article presents the fundamentals of Hamlet programming and explains how to separate content and presentation with just a smidgen of code.
- "Programming Hamlets," René Pawlitzek (developerWorks, May 2005, updated March 2007): This tutorial illustrates various aspects of Hamlet V1.3 programming as it provides a number of practical Hamlet examples. Includes detailed information on how `getElementAttributes()` works.
- "Implementing Hamlets," René Pawlitzek (developerWorks, February 2006): This article describes the Hamlet V1.1 implementation and introduces HamletHandlers, an additional way to provide dynamic content.
- "Compiling Hamlets," René Pawlitzek (developerWorks, June 2006): This article describes a small addition to the Hamlet framework: a template compiler for accelerating Hamlets.
- Hamlets home page: Check it out on alphaWorks.
- Hamlets on SourceForge: Now that this project is open source, find out how you can contribute.
- Log4j: Download this logging package from the Apache Project.
- Hamlets and OSGi: Learn more about both at Wikipedia.
- The OSGi Alliance: Find out more about this organization, formerly known as the Open Services Gateway initiative.
- OSGi Service Platform Specification: Get the details from OSGi.
- "About the OSGi Service Platform," Revision 4.1, November 11, 2005: Read the technical whitepaper to learn more. Document is a PDF.
- "OSGi introductory tutorial," Peter Kriens, www.aQute.biz, 2006: A good place to start with OSGi. Document is in Microsoft PowerPoint format.
- "OSGi tutorial: A step-by-step introduction to OSGi programming based on the open source Knopflerfish OSGi framework," Sven Haiges, October 2004: More information on OSGi, and Knopflerfish as well. Document is a PDF.
- Knopflerfish: The open source OSGi implementation used in this article's screen shots.
- Apache Felix and Eclipse Equinox: Two more open source OSGi implementations.

About the author

René Pawlitzek

René Pawlitzek is a citizen of Liechtenstein and holds an engineering degree in computer science from the Swiss Federal Institute of Technology (ETH Zürich). René works as a research and development engineer focusing on security information management solutions for the Advanced Operating Environment group (formerly Global Security Analysis Lab (GSAL)) at the IBM Zürich Research Laboratory in Switzerland. Before coming to IBM, he worked in California for Hewlett-Packard, WindRiver Systems, and Borland International.