CHAPTER 1

Using the HTTP Service

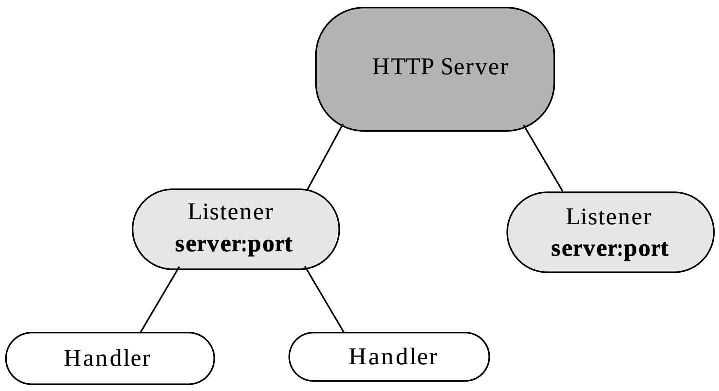
Overview of the JES HTTP Service

The Java Embedded Server^{TM} framework includes a core HTTP service that allows you to write services that serve resources to the Internet. For example, you might be building a video camera service that would allow your customers to view certain locations in their home from a website.

The main job of the core HTTP service is to serve servlets and **resources** to the Internet. Servlets are Java^{TM} classes based on the Java Servlet API, while resources can be HTML files, JSP files, or classes from bundles. The HTTP service maps resources to an URL **namespace**. A namespace is the part of the Internet's domain name system that the server controls, for example, all of the URL names that begin with http://myserver.com. This means that when a client requests an URL that falls within the server's namespace, the server delivers the corresponding resource using HTTP. The HTTP service can use either HTTP 1.1 or HTTP 1.0.

The HTTP service contains a Web server with **listeners** and **handlers**. Each listener corresponds to a port number for a Web server (for example, myserver.com:8080) and listens for HTTP requests. Handlers respond to the requests. The structure is that the HTTP server has a number of listeners and each listener has a number of handlers.

FIGURE 1-1 Conceptual Structure of the HTTP Server



The handlers are dynamically assigned to the listeners and the number of handlers varies according to the load the HTTP server is experiencing. You can configure the minimum and maximum number of handlers by setting system properties, as described in Appendix A.

To use the JES HTTP service, you need to write at least two components:

1. A **client service** that calls the HTTP service's API (contained in the com.sun.jes.service.http.HttpService interface, which extends org.osgi.service.http.HttpService).
2. An **HTTP client**, the object that actually requests a resource from the HTTP service, which is usually a Java servlet. You can write the Java servlet directly or generate it from a JSP file using the JES 2.0 JSP runtime tool (about which you can find more information in README-tcatjsp.txt in the JES 2.0 build).

In other words, you often need to write both a service and a servlet, or write a service and use the JSP runtime tool to create a servlet. In either case, you'll need access to the Java Servlet 2.1 API, available on the Internet at http://java.sun.com/products/servlet/2.1. In addition, the JES HTTP service supports both HTTP 1.0 and HTTP 1.1.

Registering Servlets and Resources

Because the JES HTTP service allows you to register Java servlets, it is aware of the servlet request-response model. Essentially, a servlet is an HTTP response to an HTTP request.

MIME stands for MultiPart Internet Mail Extensions and is the standard describing the different types of messages that can be sent across the Internet. This means that the HTTP service sends the servlet an request with a MIME body type, and the servlet sends back a response with a MIME body type. (For more information on MIME, you can look up the HTTP Request for Comments documents (RFCs) 1521 and 1522 on the Internet.)

But the servlet model is essentially that simple: the server sends a request, and the servlet sends back a response. A servlet that works with the core HTTP service uses HTTP as its protocol. This means that it typically extends HttpServlet (from the javax.servlet.http package) and uses an HttpServletRequest object as its request and an HttpServletResponse object as its response.

The JES HTTP service requires that you register both servlets and resources (which are the HTML files, JSP files, images, or Java classes that the servlet might use to deliver content to the client). Both servlets and resources must be registered with an `HttpContext` object, which maps a resource name to an URL. This means that when you submit an URL to the HTTP service, it locates the corresponding resource and responds.

What Registering Servlets Does

Servlets written according to the Java Servlet API allow your service to deliver dynamic content to the World Wide Web. Specifically, servlets do this by sending out.println statements that contain valid HTML or that call resources to the client Web browser. (You can also write JSP files to deliver dynamic content, rather than writing servlets directly. If you're interested, you should investigate the JES JSP runtime compiler with the Tomcat Web server; see the README-tcatjsp.txt file for more information.) Resources include images, HTML files, or any other type of object the servlet needs to deliver the content.

To make servlets available to the JES HTTP service, your bundle must register them. In general, your bundle registers a servlet when the HTTP service is registered, listening for an event to detect this. The JES HTTP service in turn uses a context object (created with HttpContext) to get information about the servlet, particularly the servlet's URL. The HTTP service then starts the servlet by calling its `service` method, causing the servlet to respond to the HTTP request. Registering a servlet gives the servlet access to part of the server's URI namespace, so that you can access the servlet on the server.

You can define an implementation of HttpContext yourself, or you can pass a null value instead of an HttpContext object, causing the JES HTTP service to use a default value for the context object. In either case, the context object defines such things as the MIME type the servlet uses for its response to the client, the URL at which the servlet is registered, and whether the HTTP service should service the request.

What Registering Resources Does

Your servlet typically uses resources such as images of HTML files. Registering resources makes them visible in the server's URI namespace, so that a bundle has access to them by a certain URL. Resources can be packaged into a bundle's JAR file, available on a filesystem before the bundle is installed, or created by the bundle.

Registering resources in an HTTP client service is similar to registering servlets, except that resources are typically registered as bundles. When you register a resource, you still create an HttpContext object (created from org.osgi.service.http. HttpContext or a class that implements it) to give the JES HTTP service information about the resource. You then use the registerResources method from HttpService to map a resource name to an alias name, so that the resource can be retrieved by its alias.

Because the resource is within a bundle, it gets a special bundle URL that looks like bundle:*//id/path*, for example, bundle:*//1224/images/foo.gif*. The *id* is the unique bundle ID assigned when the bundle is registered and available with the getBundleId method. The *path* is the path to the resource within the bundle or on the filesystem.

Handling Service Dependencies

When you register resources and servlets, you should do so in response to events. Events mark a change in a service's lifecycle. For example, an event fires when a service is registered or unregistered. Your service depends on the JES HTTP service, so you should register servlets and resources when the HTTP service is registered and unregister them when the HTTP service is unregistered.

The HTTP service may already be registered when you start your bundle activator class, or it may become registered while your bundle activator is running. Your bundle activator needs to handle both situations. Specifically, you need to register both within the start method, after checking that the HTTP service is already running, and also within the serviceChanged method in response to a REGISTERED event.

You then unregister your servlets and resources in response to an UNREGISTERING event (CODE EXAMPLE 1-1 makes this more clear). When you register and unregister in response to events, the JES framework uses **symbol resolution** to resolve service dependencies.

The servlet and resource registrations last as long as the JES HTTP service is registered with the framework. You should always unregister servlets explicitly with the unregister method, which calls the servlet's destroy method and stops the servlet. If you don't use unregister, the servlet's alias would be unregistered, but the servlet itself and any threads accessing it still exist.

Because other services may in turn depend on your service, never call the BundleActivator.stop method in your bundle activator class. As CODE EXAMPLE 1-1 shows, you should declare it but not implement it. You should only use BundleActivator.stop in the context of Bundle.stop, so that your bundle is stopped correctly.

The httpRegister Method

The main work of CODE EXAMPLE 1-1 is done by the httpRegister method, which is defined within the TestBundle class. httpRegister does the work of registering the resources and servlets when a REGISTERED event occurs.

The first thing that httpRegister does is create an HttpContext object. You must always create an HttpContext object, as the JES HTTP service uses the HttpContext object to get information about the servlet's registration. To create an HttpContext object, you must implement its methods, handleSecurity, getMimeType, and getResource, either in the BundleActivator class or in another class.

The implementation of these methods in CODE EXAMPLE 1-1 is very simple. The handleSecurity method returns true so that the JES HTTP service will service the request; the getMimeType method returns null to allow the JES HTTP service to determine the MIME type the servlet returns; and the getResource method returns an URL. For a more complete implementation of these methods, especially handleSecurity, see "Using Basic Authentication" later in this chapter.

The httpRegister method then registers resources and servlets using the registerResources and registerServlet methods. These methods map a resource to an URL. For example, the method call

```
hs.registerServlet( "/block", block, null, null );
```

registers the servlet object named block to the alias /block, so that is available from the URL http://*yourServer*:*yourPortNumber*/block. The complete rules of alias mapping that you use in the registerServlet and registerResources methods are summarized in TABLE 1-1.

TABLE 1-1 Specifying Aliases in JES 2.0

Alias	Definition
Default host name	Defined in **com.sun.jes.service.http.hostname**. Otherwise, accepts connections from any host. A JES extension.
Default port number	Defined in **com.sun.jes.service.http.port**. Otherwise, the default port number is **8080** for **http** and **443** for **https**. On UNIX systems, you need root privilege to bind to a port number below **1024**. A JES extension.
/*a*	Specifies the alias */a* after the default host name and port number.
/*a/b*	Specifies the alias */a/b* after the default host name and port number.
http://*host*:*port*/*alias*	Specifies a host name, port number, and servlet alias. A JES extension.
http://*host*/*alias*	Uses the default port number from **com.sun.jes.service.http.port**. Otherwise, uses **8080** as the default port number for **http** and **443** as the default port number for **https**. A JES extension.
http://*:*port*/*alias*	Registers the servlet, using any local host, the specified port number, and the specified alias. A JES extension.

▼ To Register Servlets and Resources

1. **Write a bundle activator class that implements** BundleActivator **and** ServiceListener**.**
2. **Implement the** BundleActivator.start **method.**
3. **Within the** start **method, register a service listener for the JES HTTP service, get a reference to the service, get the service itself, and register your servlets and resources, if the HTTP service has been started before the bundle activator class.**
4. **Implement the** serviceChanged **method, listening for** REGISTERED **events and registering servlets and resources if the HTTP service is registered after the bundle activator class is started.**
5. **Also in** serviceChanged**, listen for an** UNREGISTERING **event and unregister your resources if the JES HTTP service is unregistered.**
6. **Declare the** BundleActivator.stop **method, but do not call it within your bundle activator class. You should only call this method within** Bundle.stop**.**

▼ Classes and Methods

TABLE 1-2 Classes and Methods for Registering Servlets and Resources

Class, Interface, or Package	**Methods**
org.osgi.framework.BundleActivator	start, stop
org.osgi.service.http.HttpContext	getMimeType, getResource, handleSecurity
org.osgi.service.http.HttpService	registerServlet, registerResources, unregister
org.osgi.framework.BundleContext	getServiceReference, getService, addServiceListener
org.osgi.framework.ServiceListener	serviceChanged
org.osgi.framework.ServiceEvent	getType
java.net.URL	openConnection
java.net.URLConnection	getInputStream

CODE EXAMPLE 1-1 TestBundle.java (OSGi-compliant)

```
package test;

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

Implement BundleActivator and ServiceListener

```
public class TestBundle implements BundleActivator, ServiceListener {

    private HttpService service;
    private ServiceReference ref;
    private BundleContext bc ;
    private byte[] buffer = new byte[2048];
    private ByteArrayOutputStream baos = new ByteArrayOutputStream(2048);


    public void start( BundleContext bc ) throws BundleException {
```

Get the bundle context from the framework

```
        this.bc = bc;
```

Add a service listener for the HTTP service

```
        try {
            bc.addServiceListener(this, "(objectClass=org.osgi.service.http.HttpService)");
        }  catch (InvalidSyntaxException e) {
            throw new BundleException("Could not register HttpService " + "event listener", e);
        }
```

Get a reference to the HTTP service and the service itself

```
        ref = bc.getServiceReference("org.osgi.service.http.HttpService");
            if (ref != null) {
                service = (HttpService)bc.getService(ref);
```

Register here if the HTTP service is already registered

Call httpRegister(), defined below

```
                httpRegister();
            }
        }
```

Leave the stop method empty

```
    public void stop(BundleContext bc) throws BundleException {  }
```

```
public void serviceChanged(ServiceEvent event) {
```

Always make these lines synchronized

```
    synchronized (this) {
        int etype = event.getType();
        if (etype == ServiceEvent.REGISTERED) {

            if (service == null) {
                ref = bc.getServiceReference( "org.osgi.service.http.HttpService" );
```

Check to see if an HTTP service is already registered

```
                if (ref != null) {
                    service = (HttpService)bc.getService(ref);
```

If so, get the original HTTP service registered

```
                    httpRegister();
                }
            }
```

Then, register servlets and resources

```
        } else if (etype == ServiceEvent.UNREGISTERING) {
            ServiceReference sr = event.getServiceReference();
```

When a service is unregistered, get its service reference

If it's the same as the one originally registered, unregister servlets and resources

```
            if (sr.equals(ref)) {
                service.unregister("/alias");

                bc.ungetService(ref);
                service = null;
            }
        }
    }
}
```

Define httpRegister

```
private void httpRegister() {
```

Implement the 3 methods required by HttpContext

```
    HttpContext hc = new HttpContext () {
        public boolean handleSecurity(HttpServletRequest req, HttpServletResponse res) {
            return true;
        }

        public String getMimeType(String name) {
            return null;
        }
        public URL getResource(String path) {
           URL u = getClass().getResource(path);
           System.out.println("url = " + u);
           return u;
        }
     };
```

Register resources and create an URL

```
     try {
        service.registerResources("/alias", "/resources", hc);

        URL url = new URL("http://laguna:8080/alias/foo.txt");

        System.out.println("Opening URL connection ...");
        URLConnection uc = url.openConnection();

        InputStream is = uc.getInputStream();
        System.out.println( new String(getBytes(is)) );
        is.close();

     } catch (Exception e) {
        e.printStackTrace();
     }
  }

   private byte[] getBytes(InputStream is) throws IOException {
      int n;
      baos.reset();
      while ((n = is.read(buffer, 0, buffer.length)) != -1) {
         baos.write(buffer, 0, n);
      }
      return baos.toByteArray();
   }
}
```

How To Write a Servlet

The Java Servlet API consists of two packages, javax.servlet and javax.servlet.http. The two most important parts of the Java Servlet API (in the context of the JES HTTP service) are the javax.servlet.Servlet interface and the javax.servlet.http.HttpServlet class. A servlet that you use with the JES HTTP service should always extend HttpServlet, which implements Servlet through its superclass, GenericServlet.

The Servlet interface declares the three most common servlet methods—init, service, and destroy. (Remember that the registerServlet method calls service and the unregister method calls destroy.) When you extend HttpServlet to write a servlet that receives requests and returns responses by way of HTTP, you should always override at least one of its doXXX methods.

HttpServlet has a default implementation of service that dispatches requests to doGet, doPost, doPut, and other methods, according to the HTTP command that the servlet has received. An HTTP command is exchanged between the client and the server at the beginning of the request. An HTTP GET command would look something like this:

```
GET /index.html HTTP/1.1
```

Because many requests are GET requests, a servlet you write for the JES HTTP service will typically extend HttpServlet, implement a doGet method, and return HTML to the client Web browser by a number of out.println statements.

CODE EXAMPLE 1-2, which shows the beginning of SnoopServlet.java, is a simple example of how to write an HttpServlet.

First, notice that SnoopServlet extends HttpServlet. This is standard practice for servlets that return HTML to a Web browser by HTTP. SnoopServlet also happens to implement org.osgi.service.http.HttpContext, because it defines methods that the JES HTTP service can call to get information about a servlet’s registration.

After defining a default user name and password (admin for each) and a constructor, SnoopServlet starts with a doGet method. doGet is passed two objects: HttpServletRequest, which contains the information the user sent as a request, and HttpServletResponse, which allows the servlet to make a response.

CODE EXAMPLE 1-2 The Beginning of SnoopServlet.java (JES-compliant)

These lines are specific to the JES and OSGI APIs

```
package snoopbasic;

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


import org.osgi.framework.*;
import org.osgi.service.http.*;

import com.sun.jes.service.http.auth.basic.*;


public class SnoopServlet extends HttpServlet implements HttpContext {

    private final String USER = "admin";
    private final String PASS = "admin";

    SnoopActivator bc;

    public SnoopServlet ( SnoopActivator bundleContext ) {
        bc = bundleContext;
    }

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

        PrintWriterout;


        res.setContentType("text/html");
        out = res.getWriter ();


        out.println("<html>");
        out.println("<head><title>Snoop Servlet</title></head>");
        out.println("<body>");


        out.println("<h1>Request information:</h1>");
        out.println("<br>");
        out.println("<br>");
        .
        .
        .
```

The doGet method does four things. First, it creates a PrintWriter object that will become the output stream for data sent back to the client. Next, it sets the response type to text/html, indicating that the servlet produces standard HTML as its output. Third, doGet uses the getWriter method of the HttpServletResponse object to send a response of type java.io.PrintWriter (remember that the response is encoded with MIME type text/html) back to the client.

Because the response object is created when the user's request is made, it points correctly to the client that made the request. Once all that is done, doGet uses out.println statements to send HTML tags that the client Web browser can interpret and display as a Web page. This pattern is typical of doGet methods.

To summarize, a servlet you write for the JES HTTP service usually does the following:

- Extends javax.servlet.http.HttpServlet
- Implements doGet or another doXXX method that receives requests from the servlet's default service method
- Sets the response type, usually to text/html
- Creates a PrintWriter object and sets it to the return value of the getWriter method to send the response
- Uses out.println statements to send HTML to the client

These points are all basic to HTTP servlets. In addition, SnoopServlet takes steps that are specific to the JES and OSGI APIs and that are explained in more detail in "Using Basic Authentication."

You may also want to check these excellent references for more information about writing Java servlets:

- *Java Servlet Programming*, Jason Hunter with William Crawford, O'Reilly & Associates, 1998
- *Java Enterprise in a Nutshell: A Desktop Quick Reference*, Servlets chapter, David Flanagan et al, O'Reilly & Associates, 1999
- Java Servlet Specification, Version 2.1, Sun Microsystems, November 1998, available at *http://java.sun.com/products/servlet*

Note that JES 2.0 supports the Java Servlet API 2.1 and does not include the concept of **sandboxed servlets** that have security restrictions on which files they can access.

Using Basic Authentication

HTTP 1.1 provides built-in support for **basic authentication**. Basic authentication, as describedin RFC 2617, is very simple and does not provide the highest level of security. However, all browsers support it, so it is used here as an example of how to implement the HttpContext.handleSecurity method.

Basic authentication is based on a challenge/response, username/password model. When you register servlets or resources, you can provide a special implementation of handleSecurity. When a client makes a request for some URL at which you have registered a servlet or resource, the server checks if the request has proper authentication information in its request headers. If not, the server provides a challenge to the client.

The challenge causes the Web browser to display a dialog box in which the user enters a name and password that are sent back to the server. If a user with the given name and password has been configured on the server, the HTTP service honors the request. With Java Embedded Server, basic authentication uses both the JES HttpService and the BasicSchemeHandler service.

Basic authentication has some inherent weaknesses. For instance, the user name and password are transmitted directly over the Internet in clear text. Anyone monitoring the network stream would have direct access to them. Basic authentication is simply used here as an example of how to implement handleSecurity.

The examples that follow show a servlet, SnoopServlet.java, that implements HttpContext and its three methods—getMimeType, getResource, and handleSecurity—and a bundle activator class, SnoopActivator.java, that uses an instance of SnoopServlet as its context object.

▼ To Use Basic Authentication

1. **Write a servlet that extends** HttpServlet **and implements** HttpContext**.**
2. **Specify a user name and password in the servlet.**
3. **Write a** doGet **method that checks the request headers for authorization information.**
4. **Write simple implementations of** HttpContext.getResource **and** HttpContext. getMimeType**.**
5. **Write an implementation of** handleSecurity **that uses a challenge/response model to request a user name and password.**
6. **Write a bundle activator class that implements** BundleActivator **and** ServiceListener**.**
7. **Get references to both the** HttpService **and** BasicSchemeHandler **service.**
8. **Use the references to obtain both the services.**
9. **Write a** serviceChanged **method that listens for events and calls a register method to register the servlet.**
10. **Write a register method that actually does the work of registering the servlet.**
11. **Declare a** stop **method, but leave its implementation empty.**

▼ Classes and Methods

Class, Interface, or Package	**Methods**
javax.servlet.http.HttpServlet	service, doGet
org.osgi.service.http.HttpContext	getMimeType, getResource, handleSecurity
org.osgi.framework.BundleActivator	start, stop
org.osgi.framework.ServiceListener	serviceChanged
org.osgi.framework.ServiceEvent	getServiceReference, getType
org.osgi.framework.BundleContext	addServiceListener
org.osgi.service.http.HttpService	registerServlet, registerResources, unregister
com.sun.jes.service.http.auth.basic. BasicSchemeHandler	getResponse, sendChallenge
com.sun.jes.service.http.auth.basic. BasicSchemeHandler.Response	getName, getPassword

CODE EXAMPLE 1-3 SnoopActivator.java (JES-compliant)

```
package snoopbasic;

import java.util.*;
import java.net.*;

import javax.servlet.*;
import javax.servlet.http.*;
import org.osgi.framework.*;
```

Import the http and http.auth. basic packages

```
import org.osgi.service.http.*;
import com.sun.jes.service.http.auth.basic.*;
```

Implement BundleActivator and ServiceListener

```
public class SnoopActivator implements BundleActivator, ServiceListener {

    SnoopServlet snoop = null;
    HttpService http;
    BasicSchemeHandler basic;
    ServiceReference httpref;
    ServiceReference httpauthref;
```

Define the alias the servlet will use

```
    final String SERVLET_ALIAS = "/snoopbasic";
    BundleContext bundleContext;
```

Implement a start method

```
public void start(BundleContext bc) throws BundleException {

      bundleContext = bc;
      snoop = new SnoopServlet(this);
```

Get a reference to the HTTP service

```
      httpref = bc.getServiceReference( "org.osgi.service.http.HttpService" );

      if ( httpref != null ) {
          http = (HttpService) bc.getService(httpref);
          httpRegister();
      }
```

Get a reference to the Basic Scheme Handler service

```
       httpauthref =
           bc.getServiceReference( "com.sun.jes.service.http.auth.basic.BasicSchemeHandler" );

       if ( httpauthref != null ) {
           basic = (BasicSchemeHandler) bc.getService(httpauthref);
       }
```

Add service listeners for both services

```
        try {
            bc.addServiceListener( this, "(objectClass=org.osgi.service.http.HttpService)" );
            bc.addServiceListener( this,
                "( objectClass=com.sun.jes.service.http.auth.basic.BasicSchemeHandler )" );
        }


        catch(InvalidSyntaxException ise) {
            ise.printStackTrace();
        }
    }


    public synchronized void serviceChanged(ServiceEvent event) {
```

Unregister servlets when the HTTP service is unregistered

```
        if ( event.getType() == ServiceEvent.UNREGISTERING ) {
            if ( event.getServiceReference().equals(httpref) ) {
                httpUnregister();
                httpref = null;
                http = null;
            }
            else if ( event.getServiceReference().equals(httpauthref) ) {
                httpauthref = null;
                basic = null;
            }
        }
        else if ( event.getType() == ServiceEvent.REGISTERED ) {
```

Register servlets when the HTTP service is registered

```
            httpref = bundleContext.getServiceReference( "org.osgi.service.http.HttpService" );
                if ( httpref != null && http == null ) {
                    http = (HttpService) bundleContext.getService(httpref);
                    httpRegister();
                }
            }
```

Get the Basic Scheme Handler service

```
           httpauthref = bundleContext.getServiceReference(
               "com.sun.jes.service.http.auth.basic.BasicSchemeHandler");
            if ( httpauthref != null && basic == null ) {
                basic = (BasicSchemeHandler) bundleContext.getService(httpauthref);
            }
        }
```

Define the registration method

```
   private void httpRegister() {
```

Register the servlet using the servlet as the context object

```
       try {
           http.registerServlet( SERVLET_ALIAS, snoop, null, snoop );
       }
```

```
    catch(ServletException se) {
        se.printStackTrace();
    }
    catch(NamespaceException nse) {
        nse.printStackTrace();
    }
}
```

Define the unregister method

```
void httpUnregister() {
    try {
        http.unregister(SERVLET_ALIAS);
    }
    catch(IllegalArgumentException iae) { }
    }

    BasicSchemeHandler getBasicSchemeHandlerRef() {
        return basic;
}
```

Leave the implementation of stop blank

```
public void stop(BundleContext bc) throws BundleException {  }
}
```

CODE EXAMPLE 1-4 SnoopServlet.java

```
package snoopbasic;

import java.io.*;
import java.util.*;
import java.net.*;
```

Implement the servlet packages

```
import javax.servlet.*;
import javax.servlet.http.*;
```

Implement the http and http. auth.basic packages

```
import org.osgi.framework.*;
import org.osgi.service.http.*;
import com.sun.jes.service.http.auth.basic.*;
```

Extend HttpServlet and implement HttpContext

```
public class SnoopServlet extends HttpServlet implements HttpContext {
```

Define a user name and password

```
    private final String USER = "admin";
    private final String PASS = "admin";

    SnoopActivator bc;

    public SnoopServlet( SnoopActivator bundleContext ) {
      bc = bundleContext;
    }
```

Implement a doGet method

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

     PrintWriterout;

     res.setContentType("text/html");
     out = res.getWriter ();

     out.println("<html>");
     out.println("<head><title>Snoop Servlet</title></head>");
     out.println("<body>");

     out.println("<h1>Request information:</h1>");
     out.println("<br>");
     out.println("<br>");
```

Get info from the request headers sent by the client

```
    print(out, "Request method", req.getMethod());
    print(out, "Request URI", req.getRequestURI());
    print(out, "Request protocol", req.getProtocol());
    print(out, "Servlet path", req.getServletPath());
    print(out, "Path info", req.getPathInfo());


   print(out, "Path translated", req.getPathTranslated());
   print(out, "Query string", req.getQueryString());
   print(out, "Content length", req.getContentLength());
   print(out, "Content type", req.getContentType());
   print(out, "Server name", req.getServerName());


   print(out, "Server port", req.getServerPort());
   print(out, "Remote user", req.getRemoteUser());
   print(out, "Remote address", req.getRemoteAddr());
   print(out, "Remote host", req.getRemoteHost());
   print(out, "Authorization scheme", req.getAuthType());


   out.println("<br>");
   out.println("<br>");
   out.println("</body></html>");

   out.flush();
 }
```

Print the user's name

```
private void print (PrintWriter out, String name, String value) {
   out.print(" " + name + ": ");
   out.println(value == null ? "&lt;none&gt;" : value);
   out.println("<br>");
}


private void print (PrintWriter out, String name, int value) {
   out.print(" " + name + ": ");
   if (value == -1) {
      out.println("&lt;none&gt;");
   } else {
      out.println(value);
   }
}
```

Implement the three methods required by HttpContext

```
public URL getResource( String str ) {
   return null;
}

 public String getMimeType(String str) {
   return null;
}
```

Implement handleSecurity

```
    public boolean handleSecurity(HttpServletRequest req, HttpServletResponse res) {
```

handleSecurity is called for each request to the servlet

```
        BasicSchemeHandler basic = bc.getBasicSchemeHandlerRef();
        BasicSchemeHandler.Response response = basic.getResponse(req);
```

Display a dialog box for the user to log in

Return false to display the user name and dialog box

```
        if ( response == null ) {
            try {
                basic.sendChallenge(res, "dummy");
            }
            catch( IOException ioe ) {
                ioe.printStackTrace();
            }
            return false;
        }
```

Get the user name and password

```
        String user = response.getName();
        String password = response.getPassword();
```

If the user name and password don’t pass the check, display the dialog box

```
        if ( ! check(user, password) ) {
            try {
                basic.sendChallenge(res, "dummy");
            }
            catch( IOException ioe ) {
                ioe.printStackTrace();
            }
            return false;
        }

        return true;
    }
```

Define the check method

```
    boolean check(String user, String pass) {
        if ( USER.equals(user) && PASS.equals(pass) ) {
            return true;
        }
        return false;
    }

}
```

Using the HttpAdmin Service

Once you register your servlets and resources, you can get information about them by using the HttpAdmin service. HttpAdmin works with HttpService. The two services are always registered together, so HttpAdmin is always registered when HttpService is.

HttpAdmin exposes two methods—getResourceRegistrations and getServletRegistrations, each of which return an array of HttpRegistration objects. An HttpRegistration object has five basic parts: the alias name, a bundle object, an HttpContext object, a resource name or servlet object, and an URL. You extract these parts with the methods the HttpRegistration interface contains:

```
public java.lang.String getAlias()
public javax.servlet.Servlet getServlet()
public java.lang.String getResourceName()
public java.net.URL getURL(java.lang.String defaultHost) throws java.net.MalformedURLException
public HttpContext getHttpContext()
public Bundle getBundle()
```

CODE EXAMPLE 1-5 shows how to use an HttpRegistration object. The example is the Java servlet that displays the Home Portal that is shipped with the Java Embedded Server. This example happens to be a servlet, but you can use the HttpRegistration object outside of a servlet as well. The servlet also implements basic authentication, which you learned about in the previous section.

Remember that you must have the HttpService and HttpAdmin services running in order to run this example. In this release, the two services always start together.

▼ To Use the HttpRegistration Object in a Servlet

1. **Write a class that extends** HttpServlet **and implements** HttpContext**.**
2. **Get references to the** BasicSchemeHandler **and** UserPasswordService **services, then get the services themselves.**
3. **Write a** doGet **method to return data to the client. Be sure to set the content type and get a** PrintWriter **object.**
4. **Get a reference to the** HttpAdmin **service, then get the service itself.**
5. **Get the array of servlet registrations.**
6. **Move through the array, taking action on each servlet. Remember that you can use any of the methods in** HttpRegistration**.**
7. **Write a** doPost **method, if your servlet is likely to receive** POST **requests.**
8. **Implement** HttpContext **and its** getResource**,** getMimeType**, and** handleSecurity **methods. If you like, you can use basic authentication, as described in the previous section.**

▼ Classes and Methods

TABLE 1-3 Classes and Methods for Using an HttpRegistration Object

Class, Interface, or Package	Methods
javax.servlet.http.HttpServlet	doGet, doPost
org.osgi.service.httpHttpContext	getResource, getMimeType, handleSecurity
org.osgi.framework.BundleContext	getService, getServiceReference
com.sun.jes.service.http.HttpAdmin	getResourceRegistrations, getServletRegistrations
com.sun.jes.service.http.HttpRegistration	getAlias, getBundle, getHttpContext, getResourceName, getServlet, getURL
javax.servlet.ServletResponse	setContentType, getWriter
javax.servlet.ServletConfig	getInitParameter
javax.servlet.Servlet	getServletConfig

CODE EXAMPLE 1-5 HomePortalServlet.java (JES-compliant)

	package com.sun.jes.impl.homeportal;
	
	import org.osgi.framework.*;
	import org.osgi.service.http.*;

Import both HttpRegistration and HttpAdmin	import com.sun.jes.service.http.HttpRegistration;
	import com.sun.jes.service.http.HttpAdmin;

Import both the authorization packages	import com.sun.jes.service.http.auth.basic.*;
	import com.sun.jes.service.http.auth.users.*;
	
	import java.io.*;
	import javax.servlet.*;
	import javax.servlet.http.*;
	import java.util.*;
	import java.net.URL;
	
	public class HomePortalServlet extends HttpServlet implements HttpContext {

Get the localized strings for displaying in different locales	private static final LocalizedStrings ls = (LocalizedStrings)
	     java.util.ResourceBundle.getBundle("com.sun.jes.impl.homeportal.LocalizedStrings");
	
	  private HttpServletRequest myRequest;
	  private HttpServletResponse myResponse;
	  private PrintWriter out;
	  private BundleContext bc;
	
	  private BasicSchemeHandler basic;
	
	  UserPasswordService ups;
	  private boolean isAdmin = false;
	
	  public HomePortalServlet (BundleContext bc, ServiceReference anHTTPBasicReference,
	     ServiceReference anHTTPUsersReference) {
	
	     super();
	     this.bc = bc;

When passed references, get the basic and user password authentication services	basic = (BasicSchemeHandler) bc.getService(anHTTPBasicReference);
	     ups = (UserPasswordService) bc.getService(anHTTPUsersReference);
	  }

Implement a doGet method

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    this.myRequest = request;
    this.myResponse = response;
    this.myResponse.setContentType("text/html");
    this.out = new java.io.PrintWriter( response.getWriter() );
```

These four methods are all defined later

```
    htmlBegin();
    this.displayServices();
    displayChangePassword();
    htmlEnd();
}
```

Start the HTML output

```
void htmlBegin() {

    this.out.println("<HTML><HEAD><TITLE>");
    this.out.println(ls.welcomeHomePortal());
    this.out.println( "</TITLE></HEAD>" );
    this.out.println( "<BODY BACKGROUND='/images/homeportal/backrnd_tile.gif'>" );
    this.out.println( "<img src='/images/homeportal/dotcom_title.gif'><BR>" );

}
private void displayServices() {

try {
```

Get HttpAdmin before you can get an HttpRegistration object

```
    HttpAdmin httpService = (HttpAdmin)this.bc.getService(
        bc.getServiceReference( "com.sun.jes.service.http.HttpAdmin") );

    HttpRegistration[] regs = httpService.getServletRegistrations();
```

Move through the servlets in the array

```
    for ( int i=0; i<regs.length; i++ ) {

        Servlet servlet = regs[i].getServlet();

        ServletConfig servletConfig = servlet.getServletConfig();
```

Check for init parameters that make the servlet a home portal

```
        String presentationStr = servletConfig.getInitParameter(
                "com.sun.jes.service.homeportal.displayName" );

        String presentationImageAlias = servletConfig.getInitParameter(
                "com.sun.jes.service.homeportal.displayImageURL" );
```

If the servlet has a display name, get its URL

```
          if ( presentationStr != null ) {
              URL aServletURL = regs[i].getURL("*");
```

If the servlet has an image, display it

```
        if ( presentationImageAlias != null ) { // present the icon too
                this.out.println( "<img src='" + presentationImageAlias + "'>" );
        }
```

Use a null target to ensure the same browser

```

            this.printLink( aServletURL.getFile(), presentationStr,null );
            this.out.print("<BR><BR>");
        }
```

If the servlet has an image, but no text ...

```
        else if ( presentationImageAlias != null ) {
```

Display the image linked to the servlet's URL

```
            URL aServletURL = regs[i].getURL("*");
            this.out.println("<a href='");
            this.out.println(aServletURL.getFile());
            this.out.print( "'>" );
            this.out.println( "<img src='" + presentationImageAlias +"'>" );
            this.out.print( "</A>" );
        }
    }
```

Check if the user is an administrator

```
        if(isAdmin == true) {
```

Then, display a link to the JES Management Panel

```
            this.out.print("<BR><BR>");
            this.out.print( "<img src='/images/homeportal/sel.gif' WIDTH='20' HEIGHT='22'
                ALIGN='BOTTOM' BORDER='0'>" );
            this.printLink("/admin", "  JES Management Panel", null);
            this.out.print("<BR><BR>");
        }
```

printLink is defined below

```


    } catch ( Exception e ) {
        e.printStackTrace();
    }

}
```

Display the Change Password Web page

```
void displayChangePassword() {

    this.out.print("<BR><BR>");
    this.out.print( "<img src='/images/homeportal/sel.gif' WIDTH='20' HEIGHT='22'
        ALIGN='BOTTOM' BORDER='0'>" );
    this.printLink("/chgp", "  Change Password", null);
    this.out.print("<BR><BR>");

}
```

Complete the HTML output

```
void htmlEnd() {
    this.out.println("</BODY></HTML>");
    this.out.close();
}
```

Implement a doPost method to handle POST requests

```
public void doPost ( HttpServletRequest req, HttpServletResponse res )
    throws IOException, ServletException {
        this.doGet ( req, res );
}
```

Define printLink here

```
protected void printLink( String location, String label, String target ) {
    this.out.print( "<A HREF='" );
    this.out.print( location );

    if ( target != null ) {
        this.out.print( "' TARGET='"+target );
    }

    this.out.print( "'>" );
    this.out.print( "<font face='Arial, Helvetica, sans-serif' color='#ffffff' size='5'>");
    this.out.print( label );
    this.out.print("</font>");
    this.out.print( "</A>" );
}
```

Implement the 3 HttpContext methods

```
public URL getResource( String str ) {
    return null;
}

public String getMimeType(String str) {
    return null;
}
```

Implement basic authentication

```
public boolean handleSecurity( HttpServletRequest req, HttpServletResponse res ) {

    BasicSchemeHandler.Response response = basic.getResponse(req);
    if ( response != null ) {
```

Get the user name and password from the request headers

```
    String user = response.getName();
    String password = response.getPassword();
```

check is defined below

```
    if ( check(user, password) ) {
        if (ups.isAdmin(user))
          isAdmin = true;
          return true;
        }
      }


   try {
     basic.sendChallenge(res, "homeportal");
   }
   catch( IOException ioe ) {
     ioe.printStackTrace();
   }
```

Returning false causes the browser to display the login dialog box

```
   return false;
  }
```

check is defined here

```
  boolean check(String user, String pass) {

    boolean result = false;

    try {
      result = ups.checkPassword(user, pass);
    }
    catch( IllegalArgumentException iae ) {
      iae.printStackTrace();
       result = false;
    }

    return result;
  }
}
```