CHAPTER 2

Servlets

Servlets are defined as JSR 340, and the complete specification can be downloaded.

A servlet is a web component hosted in a servlet container and generates dynamic content. The web clients interact with a servlet using a request/response pattern. The servlet container is responsible for the life cycle of the servlet, receives requests and sends responses, and performs any other encoding/decoding required as part of that.

WebServlet

A servlet is defined using the `@WebServlet` annotation on a POJO, and must extend the `javax.servlet.http.HttpServlet` class.

Here is a sample servlet definition:

```
@WebServlet("/account")
public class AccountServlet extends javax.servlet.http.HttpServlet {
  //. . .
}
```

The fully qualified class name is the default servlet name, and may be overridden using the `name` attribute of the annotation. The servlet may be deployed at multiple URLs:

```
@WebServlet(urlPatterns={"/account", "/accountServlet"})
public class AccountServlet extends javax.servlet.http.HttpServlet {
  //. . .
}
```

The `@WebInitParam` can be used to specify an initialization parameter:

```
@WebServlet(urlPatterns="/account",
            initParams={
               @WebInitParam(name="type", value="checking")
                       }
           )
```

```
public class AccountServlet extends javax.servlet.http.HttpServlet {
  //. . .
}
```

The `HttpServlet` interface has one `do`*`XXX`* method to handle each of HTTP `GET`, `POST`, `PUT`, `DELETE`, `HEAD`, `OPTIONS`, and `TRACE` requests. Typically the developer is concerned with overriding the `doGet` and `doPost` methods. The following code shows a servlet handling the `GET` request:

```
@WebServlet("/account")
public class AccountServlet
  extends javax.servlet.http.HttpServlet {
  @Override
  protected void doGet(
        HttpServletRequest request,
        HttpServletResponse response) {
    //. . .
  }
}
```

In this code:

- The `HttpServletRequest` and `HttpServletResponse` capture the request/response with the web client.
- The request parameters; HTTP headers; different parts of the path such as host, port, and context; and much more information is available from `HttpServletRequest`.

The HTTP cookies can be sent and retrieved as well. The developer is responsible for populating the `HttpServletResponse`, and the container then transmits the captured HTTP headers and/or the message body to the client.

This code shows how an HTTP `GET` request received by a servlet displays a simple response to the client:

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response) {
    try (PrintWriter out = response.getWriter()) {
        out.println("<html><head>");
        out.println("<title>MyServlet</title>");
        out.println("</head><body>");
        out.println("<h1>My First Servlet</h1>");
        //. . .
        out.println("</body></html>");
    } finally {
        //. . .
    }
}
```

Request parameters may be passed in `GET` and `POST` requests. In a `GET` request, these parameters are passed in the query string as name/value pairs. Here is a sample URL to invoke the servlet explained earlier with request parameters:

```
. . ./account?tx=10
```

In a `POST` request, the request parameters can also be passed in the posted data that is encoded in the body of the request. In both `GET` and `POST` requests, these parameters can be obtained from `HttpServletRequest`:

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response) {
  String txValue = request.getParameter("tx");
  //. . .
}
```

Request parameters can differ for each request.

Initialization parameters, also known as *init params*, may be defined on a servlet to store startup and configuration information. As explained earlier, `@WebInitParam` is used to specify init params for a servlet:

```
String type = null;

@Override
public void init(ServletConfig config) throws ServletException {
  type = config.getInitParameter("type");
  //. . .
}
```

You can manipulate the default behavior of the servlet's life-cycle call methods by overriding the `init`, `service`, and `destroy` methods of the `javax.servlet.Servlet` interface. Typically, database connections are initialized in `init` and released in `destroy`.

You can also define a servlet using the `servlet` and `servlet-mapping` elements in the deployment descriptor of the web application, *web.xml*. You can define the `AccountServlet` using *web.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1"
  xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
  http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
  <servlet>
    <servlet-name>AccountServlet</servlet-name>
    <servlet-class>org.sample.AccountServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>AccountServlet</servlet-name>
    <url-pattern>/account</url-pattern>
```

```
  </servlet-mapping>
</web-app>
```

The annotations cover most of the common cases, so *web.xml* is not required in those cases. But some cases, such as ordering of servlets, can only be done using *web.xml*.

If the `metadata-complete` element in *web.xml* is true, then the annotations in the class are not processed:

```
<web-app version="3.1"
  xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
  http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
  metadata-complete="true">
  //. . .
</web-app>
```

The values defined in the deployment descriptor override the values defined using annotations.

A servlet is packaged in a web application in a *.war* file. Multiple servlets may be packaged together, and they all share a *servlet context*. The `ServletContext` provides detail about the execution environment of the servlets and is used to communicate with the container—for example, by reading a resource packaged in the web application, writing to a logfile, or dispatching a request.

The `ServletContext` can be obtained from `HttpServletRequest`:

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response) {
  ServletContext context = request.getServletContext();
  //. . .
}
```

A servlet can send an HTTP cookie, named `JSESSIONID`, to the client for session tracking. This cookie may be marked as `HttpOnly`, which ensures that the cookie is not exposed to client-side scripting code, and thus helps mitigate certains kinds of cross-site scripting attacks:

```
SessionCookieConfig config = request.getServletContext().
                             getSessionCookieConfig();
config.setHttpOnly(true);
```

Alternatively, URL rewriting may be used by the servlet as a basis for session tracking. The `ServletContext.getSessionCookieConfig` method returns `SessionCookieConfig`, which can be used to configure different properties of the cookie.

The `HttpSession` interface can be used to view and manipulate information about a session such as the session identifier and creation time, and to bind objects to the session. A new session object may be created:

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response) {
  HttpSession session = request.getSession(true);
  //. . .
}
```

The `session.setAttribute` and `session.getAttribute` methods are used to bind objects to the session.

A servlet may forward a request to another servlet if further processing is required. You can achieve this by dispatching the request to a different resource using `RequestDispatcher`, which can be obtained from `HttpServletRequest.getRequestDispatcher` or `ServletContext.getRequestDispatcher`. The former can accept a relative path, whereas the latter can accept a path relative to the current context only:

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response) {
  request.getRequestDispatcher("bank").forward(request, response);
  //. . .
}
```

In this code, `bank` is another servlet deployed in the same context.

The `ServletContext.getContext` method can be used to obtain `ServletContext` for foreign contexts. It can then be used to obtain a `RequestDispatcher`, which can dispatch requests in that context.

You can redirect a servlet response to another resource by calling the `HttpServletResponse.sendRedirect` method. This sends a temporary redirect response to the client, and the client issues a new request to the specified URL. Note that in this case, the original request object is not available to the redirected URL. The redirect may also be marginally slower because it entails two requests from the client, whereas `forward` is performed within the container:

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response) {
  //. . .
  response.sendRedirect("http://example.com/SomeOtherServlet");
}
```

Here the response is redirected to the *http://example.com/SomeOtherServlet* URL. Note that this URL could be on a different host/port and may be relative or absolute to the container.

In addition to declaring servlets using `@WebServlet` and *web.xml*, you can define them programmatically using `ServletContext.addServlet` methods. You can do this from the `ServletContainerInitializer.onStartup` or `ServletContextListener.contextInitialized` method. You can read more about this in "Event Listeners" on page 17.

The `ServletContainerInitializer.onStartup` method is invoked when the application is starting up for the given `ServletContext`. The `addServlet` method returns `ServletRegistration.Dynamic`, which can then be used to create URL mappings, set security roles, set initialization parameters, and manage other configuration items:

```
public class MyInitializer implements ServletContainerInitializer {
  @Override
  public void onStartup (Set<Class<?>> clazz, ServletContext context) {
    ServletRegistration.Dynamic reg =
        context.addServlet("MyServlet", "org.example.MyServlet");
    reg.addMapping("/myServlet");
  }
}
```

Servlet Filters

A servlet filter may be used to update the request and response payload and header information from and to the servlet. It is important to realize that filters do not create the response—they only modify or adapt the requests and responses. Authentication, logging, data compression, and encryption are some typical use cases for filters. The filters are packaged along with a servlet and act upon the dynamic or static content.

You can associate filters with a servlet or with a group of servlets and static content by specifying a URL pattern. You define a filter using the `@WebFilter` annotation:

```
@WebFilter("/*")
public class LoggingFilter implements javax.servlet.Filter {
  public void doFilter(HttpServletRequest request,
                       HttpServletResponse response) {
    //. . .
  }
}
```

In the code shown, the `LoggingFilter` is applied to all the servlets and static content pages in the web application.

The `@WebInitParam` may be used to specify initialization parameters here as well.

A filter and the target servlet always execute in the same invocation thread. Multiple filters may be arranged in a filter chain.

You can also define a filter using `<filter>` and `<filter-mapping>` elements in the deployment descriptor:

```
<filter>
  <filter-name>LoggingFilter</filter-name>
  <filter-class>org.sample.LoggingFilter</filter-class>
</filter>
    . . .
<filter-mapping>
  <filter-name>LoggingFilter</filter-name>
```

```
    <url-pattern>/*</url-pattern>
  </filter-mapping>
```

In addition to declaring filters using `@WebFilter` and *web.xml*, you can define them programmatically using `ServletContext.addFilter` methods. You can do this from the `ServletContainerInitializer.onStartup` method or the `ServletContextListener.contextInitialized` method. The `addFilter` method returns `ServletRegistration.Dynamic`, which can then be used to add mapping for URL patterns, set initialization parameters, and handle other configuration items:

```
public class MyInitializer implements ServletContainerInitializer {
  public void onStartup (Set<Class<?>> clazz, ServletContext context) {
    FilterRegistration.Dynamic reg =
        context.addFilter("LoggingFilter",
                  "org.example.LoggingFilter");
    reg.addMappingForUrlPatterns(null, false, "/");
  }
}
```

Event Listeners

Event listeners provide life-cycle callback events for `ServletContext`, `HttpSession`, and `ServletRequest` objects. These listeners are classes that implement an interface that supports event notifications for state changes in these objects. Each class is annotated with `@WebListener`, declared in *web.xml*, or registered via one of the `ServletContext.addListener` methods. A typical example of these listeners is where an additional servlet is registered programmatically without an explicit need for the programmer to do so, or a database connection is initialized and restored back at the application level.

There may be multiple listener classes listening to each event type, and they may be specified in the order in which the container invokes the listener beans for each event type. The listeners are notified in the reverse order during application shutdown.

Servlet context listeners listen to the events from resources in that context:

```
@WebListener
public class MyContextListener implements ServletContextListener {

  @Override
  public void contextInitialized(ServletContextEvent sce) {
    ServletContext context = sce.getServletContext();
    //. . .
  }

  @Override
  public void contextDestroyed(ServletContextEvent sce) {
    //. . .
  }
}
```

The `ServletContextAttributeListener` is used to listen for attribute changes in the context:

```
public class MyServletContextAttributeListener
             implements ServletContextAttributeListener {

  @Override
  public void attributeAdded(ServletContextAttributeEvent event) {
    //. . . event.getName();
    //. . . event.getValue();
  }

  @Override
  public void attributeRemoved(ServletContextAttributeEvent event) {
    //. . .
  }

  @Override
  public void attributeReplaced(ServletContextAttributeEvent event) {
    //. . .
  }

}
```

The `HttpSessionListener` listens to events from resources in that session:

```
@WebListener
public class MySessionListener implements HttpSessionListener {

  @Override
  public void sessionCreated(HttpSessionEvent hse) {
    HttpSession session = hse.getSession();
    //. . .
  }

  @Override
  public void sessionDestroyed(HttpSessionEvent hse) {
    //. . .
  }
}
```

The `HttpSessionActivationListener` is used to listen for events when the session is passivated or activated:

```
public class MyHttpSessionActivationListener
             implements HttpSessionActivationListener {

  @Override
  public void sessionWillPassivate(HttpSessionEvent hse) {
    // ... hse.getSession();
  }

  @Override
```

```
  public void sessionDidActivate(HttpSessionEvent hse) {
    // ...
  }
}
```

The `HttpSessionAttributeListener` is used to listen for attribute changes in the session:

```
public class MyHttpSessionAttributeListener
             implements HttpSessionAttributeListener {

  @Override
  public void attributeAdded(HttpSessionBindingEvent event) {
    HttpSession session = event.getSession();
    //. . . event.getName();
    //. . . event.getValue();
  }

  @Override
  public void attributeRemoved(HttpSessionBindingEvent event) {
    //. . .
  }

  @Override
  public void attributeReplaced(HttpSessionBindingEvent event) {
    //. . .
  }
}
```

The `HttpSessionBindingListener` is used to listen to events when an object is bound to or unbound from a session:

```
public class MyHttpSessionBindingListener
             implements HttpSessionBindingListener {

  @Override
  public void valueBound(HttpSessionBindingEvent event) {
    HttpSession session = event.getSession();
    //. . . event.getName();
    //. . . event.getValue();
  }

  @Override
  public void valueUnbound(HttpSessionBindingEvent event) {
    //. . .
  }
}
```

The `ServletRequestListener` listens to the events from resources in that request:

```
@WebListener
public class MyRequestListener implements ServletRequestListener {
  @Override
  public void requestDestroyed(ServletRequestEvent sre) {
```

```
    ServletRequest request = sre.getServletRequest();
    //. . .
  }

  @Override
  public void requestInitialized(ServletRequestEvent sre) {
    //. . .
  }
}
```

The `ServletRequestAttributeListener` is used to listen for attribute changes in the request.

There is also `AsyncListener`, which is used to manage async events such as completed, timed out, or an error.

In addition to declaring listeners using `@WebListener` and *web.xml*, you can define them programmatically using `ServletContext.addListener` methods. You can do this from the `ServletContainerInitializer.onStartup` or `ServletContextListener.contextInitialized` method.

The `ServletContainerInitializer.onStartup` method is invoked when the application is starting up for the given `ServletContext`:

```
public class MyInitializer implements ServletContainerInitializer {
  public void onStartup(Set<Class<?>> clazz, ServletContext context) {
    context.addListener("org.example.MyContextListener");
  }
}
```

Asynchronous Support

Server resources are valuable and should be used conservatively. Consider a servlet that has to wait for a JDBC connection to be available from the pool, receiving a JMS message or reading a resource from the filesystem. Waiting for a "long-running" process to return completely blocks the thread—waiting, sitting, and doing nothing—which is not an optimal usage of your server resources. This is where the server can be asynchronously processed such that the control (or thread) is returned to the container to perform other tasks while waiting for the long-running process to complete. The request processing continues in the same thread after the response from the long-running process is returned, or may be dispatched to a new resource from within the long-running process. A typical use case for a long-running process is a chat application.

The asynchronous behavior needs to be explicitly enabled on a servlet. You achieve this by adding the `asyncSupported` attribute on `@WebServlet`:

```
@WebServlet(urlPatterns="/async", asyncSupported=true)
public class MyAsyncServlet extends HttpServlet {
```

```
    //. . .
  }
```

You can also enable the asynchronous behavior by setting the `<async-supported>` element to true in *web.xml* or calling `ServletRegistration.setAsyncSupported(true)` during programmatic registration.

You can then start the asynchronous processing in a separate thread using the `startAsync` method on the request. This method returns `AsyncContext`, which represents the execution context of the asynchronous request. Then you can complete the asynchronous request by calling `AsyncContext.complete` (explicit) or dispatching to another resource (implicit). The container completes the invocation of the asynchronous request in the latter case.

Let's say the long-running process is implemented:

```
class MyAsyncService implements Runnable {
  AsyncContext ac;

  public MyAsyncService(AsyncContext ac) {
    this.ac = ac;
  }

  @Override
  public void run() {
    //. . .
    ac.complete();
  }
}
```

This service may be invoked from the `doGet` method:

```
@Override
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response) {
  AsyncContext ac = request.startAsync();
  ac.addListener(new AsyncListener() {
    public void onComplete(AsyncEvent event)
        throws IOException {
      //. . .
    }

    public void onTimeout(AsyncEvent event)
        throws IOException {
      //. . .
    }
    //. . .
  });

  ScheduledThreadPoolExecutor executor = new ScheduledThreadPoolExecutor(10);
  executor.execute(new MyAsyncService(ac));
}
```

In this code, the request is put into asynchronous mode. `AsyncListener` is registered to listen for events when the request processing is complete, has timed out, or resulted in an error. The long-running service is invoked in a separate thread and calls `AsyncContext.complete`, signalling the completion of request processing.

A request may be dispatched from an asynchronous servlet to synchronous, but the other way around is illegal.

The asynchronous behavior is available in the servlet filter as well.

Nonblocking I/O

Servlet 3.0 allowed asynchronous request processing but only permitted traditional I/O, which restricted the scalability of your applications. In a typical application, `ServletInputStream` is read in a `while` loop:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {
  ServletInputStream input = request.getInputStream();
  byte[] b = new byte[1024];
  int len = -1;
  while ((len = input.read(b)) != -1) {
    //. . .
  }
}
```

If the incoming data is blocking or streamed slower than the server can read, then the server thread is waiting for that data. The same can happen if the data is written to `ServletOutputStream`. This restricts the scalability of the Web Container.

Nonblocking I/O allows developers to read data as it becomes available or write data when it's possible to do so. This increases not only the scalability of the Web Container but also the number of connections that can be handled simultaneously. Nonblocking I/O only works with async request processing in Servlets, Filters, and Upgrade Processing.

Servlet 3.1 achieves nonblocking I/O by introducing two new interfaces: `ReadListener` and `WriteListener`. These listeners have callback methods that are invoked when the content is available to be read or can be written without blocking.

The `doGet` method needs to be rewritten in this case:

```
AsyncContext context = request.startAsync();
ServletInputStream input = request.getInputStream();
input.setReadListener(new MyReadListener(input, context));
```

Invoking `set`*`XXX`*`Listener` methods indicates that nonblocking I/O is used instead of traditional.

`ReadListener` has three callback methods:

- The `onDataAvailable` callback method is called whenever data can be read without blocking.
- The `onAllDataRead` callback method is invoked whenever data for the current request is completely read.
- The `onError` callback is invoked if there is an error processing the request:

```
@Override
public void onDataAvailable() {
  try {
    StringBuilder sb = new StringBuilder();
    int len = -1;
    byte b[] = new byte[1024];
    while (input.isReady() && (len = input.read(b)) != -1) {
      String data = new String(b, 0, len);
    }
  } catch (IOException ex) {
    //. . .
  }
}

@Override
public void onAllDataRead() {
  context.complete();
}

@Override
public void onError(Throwable t) {
  t.printStackTrace();
  context.complete();
}
```

In this code, the `onDataAvailable` callback is invoked whenever data can be read without blocking. The `ServletInputStream.isReady` method is used to check if data can be read without blocking and then the data is read. `context.complete` is called in `onAllDataRead` and `onError` methods to signal the completion of data read. `ServletInputStream.isFinished` may be used to check the status of a nonblocking I/O read.

At most, one `ReadListener` can be registered on `ServletIntputStream`.

`WriteListener` has two callback methods:

- The `onWritePossible` callback method is called whenever data can be written without blocking.
- The `onError` callback is invoked if there is an error processing the response.

At most, one `WriteListener` can be registered on `ServletOutputStream`. `ServletOutputStream.canWrite` is a new method to check if data can be written without blocking.

Web Fragments

A web fragment is part or all of the *web.xml* file included in a library or framework JAR's *META-INF* directory. If this framework is bundled in the *WEB-INF/lib* directory, the container will pick up and configure the framework without requiring the developer to do it explicitly.

It can include almost all of the elements that can be specified in *web.xml*. However, the top-level element must be `web-fragment` and the corresponding *file* must be called *web-fragment.xml*. This allows logical partitioning of the web application:

```
<web-fragment>
  <filter>
    <filter-name>MyFilter</filter-name>
    <filter-class>org.example.MyFilter</filter-class>
    <init-param>
      <param-name>myInitParam</param-name>
      <param-value>...</param-value>
    </init-param>
  </filter>
  <filter-mapping>
    <filter-name>MyFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
</web-fragment>
```

The developer can specify the order in which the resources specified in *web.xml* and *web-fragment.xml* need to be loaded. The `<absolute-ordering>` element in *web.xml* is used to specify the exact order in which the resources should be loaded, and the `<ordering>` element within *web-fragment.xml* is used to specify relative ordering. The two orders are mutually exclusive, and absolute ordering overrides relative.

The absolute ordering contains one or more `<name>` elements specifying the name of the resources and the order in which they need to be loaded. Specifying `<others/>` allows for the other resources not named in the ordering to be loaded:

```
<web-app>
  <name>MyApp</name>
  <absolute-ordering>
    <name>MyServlet</name>
    <name>MyFilter</name>
  </absolute-ordering>
</web-app>
```

In this code, the resources specified in *web.xml* are loaded first and followed by `MyServlet` and `MyFilter`.

Zero or one `<before>` and `<after>` elements in `<ordering>` are used to specify the resources that need to be loaded before and after the resource named in the `web-fragment` is loaded:

```
<web-fragment>
  <name>MyFilter</name>
  <ordering>
    <after>MyServlet</after>
  </ordering>
</web-fragment>
```

This code will require the container to load the resource `MyFilter` after the resource `MyServlet` (defined elsewhere) is loaded.

If *web.xml* has `metadata-complete` set to true, then the *web-fragment.xml* file is not processed. The *web.xml* file has the highest precedence when resolving conflicts between *web.xml* and *web-fragment.xml*.

If a *web-fragment.xml* file does not have an `<ordering>` element and *web.xml* does not have an `<absolute-ordering>` element, the resources are assumed to not have any ordering dependency.

Security

Servlets are typically accessed over the Internet, and thus having a security requirement is common. You can specify the servlet security model, including roles, access control, and authentication requirements, using annotations or in *web.xml*.

`@ServletSecurity` is used to specify security constraints on the servlet implementation class for all methods or a specific `do`*XXX* method. The container will enforce that the corresponding `do`*XXX* messages can be invoked by users in the specified roles:

```
@WebServlet("/account")
@ServletSecurity(
  value=@HttpConstraint(rolesAllowed = {"R1"}),
  httpMethodConstraints={
    @HttpMethodConstraint(value="GET",
                          rolesAllowed="R2"),
    @HttpMethodConstraint(value="POST",
                          rolesAllowed={"R3", "R4"})
  }
)
public class AccountServlet
             extends javax.servlet.http.HttpServlet {
  //. . .
}
```

In this code, `@HttpMethodConstraint` is used to specify that the `doGet` method can be invoked by users in the `R2` role, and the `doPost` method can be invoked by users in the `R3` and `R4` roles. The `@HttpConstraint` specifies that all other methods can be invoked by users in the role `R1`. The roles are mapped to security principals or groups in the container.

The security constraints can also be specified using the `<security-constraint>` element in *web.xml*. Within it, a `<web-resource-collection>` element is used to specify constraints on HTTP operations and web resources, `<auth-constraint>` is used to specify the roles permitted to access the resource, and `<user-data-constraint>` indicates how data between the client and server should be protected by the subelement `<transport-guarantee>`:

```
<security-constraint>
  <web-resource-collection>
    <url-pattern>/account/*</url-pattern>
    <http-method>GET</http-method>
  </web-resource-collection>

  <auth-constraint>
    <role-name>manager</role-name>
  </auth-constraint>

  <user-data-constraint>
    <transport-guarantee>INTEGRITY</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

This deployment descriptor requires that only the `GET` method at the `/account/*` URL is protected. This method can only be accessed by a user in the `manager` role with a requirement for content integrity. All HTTP methods other than `GET` are unprotected.

If HTTP methods are not enumerated within a `security-constraint`, the protections defined by the constraint apply to the complete set of HTTP (extension) methods:

```
<security-constraint>
  <web-resource-collection>
    <url-pattern>/account/*</url-pattern>
  </web-resource-collection>

  . . .
</security-constraint>
```

In this code, all HTTP methods at the `/account/*` URL are protected.

Servlet 3.1 defines *uncovered* HTTP protocol methods as the methods that are not listed in the `<security-constraint>` and if at least one `<http-method>` is listed in `<security-constraint>`:

```
<security-constraint>
  <web-resource-collection>
    <url-pattern>/account/*</url-pattern>
    <http-method>GET</http-method>
  </web-resource-collection>
  . . .
</security-constraint>
```

In this code fragment, only the HTTP `GET` method is protected and all other HTTP protocols methods such as `POST` and `PUT` are uncovered.

The `<http-method-omission>` element can be used to specify the list of HTTP methods not protected by the constraint:

```
<security-constraint>
  <web-resource-collection>
    <url-pattern>/account/*</url-pattern>
    <http-method-omission>GET</http-method-omission>
  </web-resource-collection>

  . . .
</security-constraint>
```

In this code, only the HTTP `GET` method is not protected and all other HTTP protocol methods are protected.

The `<deny-uncovered-http-methods>` element, a new element in Servlet 3.1, can be used to deny an HTTP method request for an uncovered HTTP method. The denied request is returned with a 403 (`SC_FORBIDDEN`) status code:

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
           http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
         version="3.1">
  <deny-uncovered-http-methods/>
  <web-resource-collection>
    <url-pattern>/account/*</url-pattern>
    <http-method>GET</http-method>
  </web-resource-collection>
  . . .
</web-app>
```

In this code, the `<deny-uncovered-http-methods>` element ensures that HTTP `GET` is called with the required security credentials, and all other HTTP methods are denied with a 403 status code.

`@RolesAllowed`, `@DenyAll`, `@PermitAll`, and `@TransportProtected` provide an alternative set of annotations to specify security roles on a particular resource or a method of the resource:

```
@RolesAllowed("R2")
protected void doGet(HttpServletRequest request, HttpServletResponse response) {
  //. . .
}
```

If an annotation is specified on both the class and the method level, the one specified on the method overrides the one specified on the class.

Servlet 3.1 introduces two new predefined roles:

- * maps to any defined role.
- ** maps to any authenticated user independent of the role.

This allows you to specify security constraints at a higher level than a particular role.

At most, one of `@RolesAllowed`, `@DenyAll`, or `@PermitAll` may be specified on a target. The `@TransportProtected` annotation may occur in combination with either the `@RolesAllowed` or `@PermitAll` annotations.

The servlets can be configured for HTTP Basic, HTTP Digest, HTTPS Client, and form-based authentication:

```
<form method="POST" action="j_security_check">
  <input type="text" name="j_username">
  <input type="password" name="j_password" autocomplete="off">
  <input type="button" value="submit">
</form>
```

This code shows how form-based authentication can be achieved. The login form must contain fields for entering a username and a password. These fields must be named `j_username` and `j_password`, respectively. The action of the form is always `j_security_check`.

Servlet 3.1 requires `autocomplete="off"` on the password form field, further strengthening the security of servlet-based forms.

The `HttpServletRequest` also provides programmatic security with the `login`, `logout`, and `authenticate` methods.

The `login` method validates the provided username and password in the password validation realm (specific to a container) configured for the `ServletContext`. This ensures that the `getUserPrincipal`, `getRemoteUser`, and `getAuthType` methods return valid values. The `login` method can be used as a replacement for form-based login.

The `authenticate` method uses the container login mechanism configured for the `ServletContext` to authenticate the user making this request.

Resource Packaging

You can access resources bundled in the *.war* file using the `ServletContext.getResource` and `.getResourceAsStream` methods. The resource path is specified as a string with a leading "/." This path is resolved relative to the root of the context or relative to the *META-INF/resources* directory of the JAR files bundled in the *WEB-INF/lib* directory:

```
myApplication.war
  WEB-INF
```

```
    lib
      library.jar
```

library.jar has the following structure:

```
library.jar
  MyClass1.class
  MyClass2.class
  stylesheets
    common.css
  images
    header.png
    footer.png
```

Normally, if *stylesheets* and *image* directories need to be accessed in the servlet, you need to manually extract them in the root of the web application. Servlet 3.0 allows the library to package the resources in the *META-INF/resources* directory:

```
library.jar
  MyClass1.class
  MyClass2.class
  META-INF
    resources
      stylesheets
        common.css
      images
        header.png
        footer.png
```

In this case, the resources need not be extracted in the root of the application and can be accessed directly instead. This allows resources from third-party JARs bundled in *META-INF/resources* to be accessed directly instead of manually extracted.

The application always looks for resources in the root before scanning through the JARs bundled in the *WEB-INF/lib* directory. The order in which it scans JAR files in the *WEB-INF/lib* directory is undefined.

Error Mapping

An HTTP error code or an exception thrown by a serlvet can be mapped to a resource bundled with the application to customize the appearance of content when a servlet generates an error. This allows fine-grained mapping of errors from your web application to custom pages. These pages are defined via `<error-page>`:

```
<error-page>
  <error-code>404</error-code>
  <location>/error-404.jsp</location>
</error-page>
```

Adding the preceding fragment to *web.xml* will display the */error-404.jsp* page to a client attempting to access a nonexistent resource. You can easily implement this mapping for other HTTP status codes as well by adding other `<error-page>` elements.

The `<exception-type>` element is used to map an exception thrown by a servlet to a resource in the web application:

```
<error-page>
  <exception-type>org.example.MyException</exception-type>
  <location>/error.jsp</location>
</error-page>
```

Adding the preceding fragment to *web.xml* will display the */error.jsp* page to the client if the servlet throws the `org.example.MyException` exception. You can easily implement this mapping for other exceptions as well by adding other `<error-page>` elements.

The `<error-page>` declaration must be unique for each class name and HTTP status code.

Handling Multipart Requests

`@MultipartConfig` may be specified on a servlet, indicating that it expects a request of type `multipart/form-data`. The `HttpServletRequest.getParts` and `.getPart` methods then make the various parts of the multipart request available:

```
@WebServlet(urlPatterns = {"/FileUploadServlet"})
@MultipartConfig(location="/tmp")
public class FileUploadServlet extends HttpServlet {

  @Override
  protected void doPost(HttpServletRequest request,
                        HttpServletResponse response)
        throws ServletException, IOException {
    for (Part part : request.getParts()) {
      part.write("myFile");
    }
  }
}
```

In this code:

- `@MultipartConfig` is specified on the class, indicating that the `doPost` method will receive a request of type `multipart/form-data`.
- The `location` attribute is used to specify the directory location where the files are stored.
- The `getParts` method provides a `Collection` of parts for this multipart request.
- `part.write` is used to write this uploaded part to disk.

Servlet 3.1 adds a new method, `Part.getSubmittedFileName`, to get the filename specified by the client.

This servlet can be invoked from a JSP page:

```
<form action="FileUploadServlet"
      enctype="multipart/form-data"
      method="POST">
    <input type="file" name="myFile"><br>
    <input type="Submit" value="Upload File"><br>
</form>
```

In this code, the form is `POST`ed to `FileUploadServlet` with encoding `multipart/form-data`.

Upgrade Processing

Section 14.42 of HTTP 1.1 (RFC 2616) defines an upgrade mechanism that allows you to transition from HTTP 1.1 to some other, incompatible protocol. The capabilities and nature of the application-layer communication after the protocol change are entirely dependent upon the new protocol chosen. After an upgrade is negotiated between the client and the server, the subsequent requests use the newly chosen protocol for message exchanges. A typical example is how the WebSocket protocol is upgraded from HTTP, as described in the Opening Handshake section of RFC 6455.

The servlet container provides an HTTP upgrade mechanism. However, the servlet container itself does not have any knowledge about the upgraded protocol. The protocol processing is encapsulated in the `HttpUpgradeHandler`. Data reading or writing between the servlet container and the `HttpUpgradeHandler` is in byte streams.

The decision to upgrade is made in the `Servlet.service` method. Upgrading is achieved by adding a new method, `HttpServletRequest.upgrade`, and two new interfaces, `javax.servlet.http.HttpUpgradeHandler` and `javax.servlet.http.WebConnection`:

```
if (request.getHeader("Upgrade").equals("echo")) {
    response.setStatus(HttpServletResponse.SC_SWITCHING_PROTOCOLS);
    response.setHeader("Connection", "Upgrade");
    response.setHeader("Upgrade", "echo");
    request.upgrade(MyProtocolHandler.class);
    System.out.println("Request upgraded to MyProtocolHandler");
}
```

The request looks for the `Upgrade` header and makes a decision based upon its value. In this case, the connection is upgraded if the `Upgrade` header is equal to `echo`. The correct response status and headers are set. The `upgrade` method is called on `HttpServletRequest` by passing an instance of `HttpUpgradeHandler`.