

4

Designing and Developing Servlets to Handle Server-Side Exceptions

- 4.1 For each of the following cases, identify correctly constructed code for handling business logic exceptions, and match that code with correct statements about the code's behavior:
- Return an HTTP error using the `sendError` response method
 - Return an HTTP error using the `setStatus` method
- 4.2 Given a set of business logic exceptions, identify the following:
- The configuration that the deployment descriptor uses to handle each exception
 - How to use a `RequestDispatcher` to forward the request to an error page
 - Specify the handling declaratively in the deployment descriptor
- 4.3 Identify the method used for the following:
- Write a message to the WebApp log
 - Write a message and exception to the WebApp log

The previous chapter introduced the servlet container model and outlined some of the services that the container provides to the web components executing inside it. Another key service is the error handling mechanism that allows error pages to be configured declaratively through the deployment descriptor should an exception or error status code be returned.

This mechanism provides an excellent way to consistently handle adverse conditions in web applications, as there is nothing worse than presenting your users with a horrible Java stack trace that usually means very little to them. Besides, such instances do little to explain the problem that has occurred on the server and do nothing to instill confidence in the robustness of the application.

This chapter will introduce the declarative exception handling mechanism and will show how it can be used to provide descriptive error pages for a wide range of server-side errors. As in previous chapters, the objectives from this chapter will be demonstrated with examples, both in the text and in the accompanying web application.

Designing and Developing Servlets to Handle Server-Side Exceptions

4.1 For each of the following cases, identify correctly constructed code for handling business logic exceptions, and match that code with correct statements about the code's behavior:

Each time a client makes an HTTP request to a server for a resource such as an HTML page, a servlet, or a JSP page, the server responds to the request and returns a status line, some headers, and a blank line followed by the requested document:

```
HTTP/1.1 200 OK
Content-type: text/html

Hello World
```

The above example shows a very simple HTTP response as sent to a browser. Here we can see that the status line consists of the HTTP version (`HTTP/1.1`) and a status code (`200`) followed by a short textual description of the code (`OK`). Any necessary headers are then returned (`Content-type: text/html`), followed by two carriage returns and the requested document.

It should be noted that not all HTTP requests actually contain a document; in Chapter 1 we saw how the HTTP `HEAD` method only returns the response headers. Certain status codes can also be set to indicate to the browser that the server could not service the request for some reason and is therefore returning an error document or possibly nothing at all.

The requesting client (usually a browser) interprets the status code and can then decide how to process the response. In the vast majority of cases, HTTP requests are met with the default HTTP 200 OK status code, which means the request was successful.

Return an HTTP error using the setStatus method

So far, none of the servlets that we have seen explicitly set an HTTP status code as part of their response to the client. This is because we have been relying on the web server to set an appropriate one on our behalf, which it does if it detects that a status code has not explicitly been set.

All of the example servlets we have seen up until now have had the default 200 OK status code set by the server. As we have seen, relying on the server to set the status code is perfectly adequate for the vast majority of the time. However, there are occasions when explicitly returning one of the many available HTTP status codes can dramatically affect the way a client browser reacts to the response.

For example, by returning an appropriate status code and header we can inform a browser that a resource has permanently or temporarily moved. Most browsers in use today will automatically read the header and will request the resource from the new location.

The `HttpServletResponse` interface provides a method for setting the status code of an HTTP request:

```
public void setStatus(int statusCode)
```

This method can be used to set any non-error status codes, that is, the valid HTTP response codes that signify that the request generally succeeded and there was no server-side error. This method takes a single integer parameter that corresponds to the HTTP status code required:

```
response.setStatus(200);
```

While the method call given above is perfectly legal, it is far easier and less error prone to use the constants declared in the `HttpServletResponse` interface. Using these constants, the above method call can be replaced with the following:

```
response.setStatus(HttpServletResponse.SC_OK);
```

Some of the more common HTTP status codes and their `HttpServletResponse` constants are shown below, along with a short description:

Status Code	Constant	Meaning
200	SC_OK	The request succeeded normally.
204	SC_NO_CONTENT	The request succeeded but there was no new information to return.
301	SC_MOVED_PERMANENTLY	The resource has permanently moved to a new location and future references should use a new URI with their requests. The <code>Location</code> HTTP header may give the new URI and most browsers will automatically forward to the location specified.

Table continued on following page

Status Code	Constant	Meaning
302	SC_MOVED_TEMPORARILY	The resource has temporarily moved to a new location but future references should still use the original URI to access the resource. The Location HTTP header may give the new URI and most browsers will automatically forward to the location specified.
304	SC_NOT_MODIFIED	A conditional GET operation found that the resource is available and has not been modified.
401	SC_UNAUTHORIZED	The request requires HTTP authentication.
404	SC_NOT_FOUND	The requested resource is not available.
500	SC_INTERNAL_SERVER_ERROR	An error inside the HTTP server prevented it from fulfilling the request.

Note that it is not a requirement of the examination to remember all of the valid HTTP response status codes, although you should familiarize yourself with the more common codes such as 200 OK, 404 Not Found, and 500 Internal Server Error.

As previously mentioned, the Servlet API indicates that the `setStatus()` method should only be used to set **non-error** status codes. So, which of the many available HTTP status codes are classified as error codes? The simple answer is that any status code of the form 4xx indicates that the client is in error and any status code of the form 5xx indicates an error with the server.

As the Servlet API suggests, the `setStatus()` method is intended to be used with those status codes which indicate that the request was successful to some degree, for example, `SC_OK` (200) and `SC_MOVED_TEMPORARILY` (302). While it is quite possible to pass any status code to the `setStatus()` method, we will see shortly that there is a better alternative to this method, especially designed to indicate error conditions.

Another important point to note is that the `setStatus()` method should be called before any data is returned to the client, in other words before the response is flushed or closed. The reason for this limitation can be illustrated by recalling the HTTP response shown earlier:

```
HTTP/1.1 200 OK
Content-type: text/html

Hello World
```

The first line of the server response contains the status code (200 in this case), followed by any headers, and finally the body of the response. Therefore, if the response has already been committed (or flushed) it is too late to change the status: it will have already been sent before the response body that has just been committed.

HTTP headers do not cause a problem as they are buffered by the servlet and returned all at once. Therefore, the status may be set after a header has been set.

There is a slight caveat to this functionality, in that from version 2.2 of the Servlet Specification, servlet containers are allowed to, but not required to, buffer any output sent to the client for efficiency purposes. Therefore, providing that a buffer has not yet been filled and committed to the client, the `setStatus()` method can still be used to set the status code of the response.

The `ServletResponse` interface provides the following methods to interrogate and manipulate the buffer:

```
public boolean isCommitted()
public int getBufferSize()
public void setBufferSize(int size)
public void reset()
public void resetBuffer()
public void flushBuffer()
```

The `isCommitted()` method can be used to find out if any output has been returned to the client. If this method returns `true` then it is too late to set the status and any attempt to do so is ignored. The `reset()` method clears any data in the buffer as well as any headers and status codes, while the `resetBuffer()` method simply clears the buffer only. The `flushBuffer()` method forces any content in the buffer to be written to the client.

Let's illustrate this technique with an example (`ChangeStatusServlet.java`):

```
public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    PrintWriter writer = res.getWriter();

    // Let's set the status code manually
    res.setStatus(HttpServletResponse.SC_ACCEPTED);

    // ..and start to return some response
    writer.println("<html>");
    writer.println("<body>");
    writer.println("<p>Here is the start of the response...");
```

Providing the buffer has not been filled and automatically flushed, and the `flush()` method of the writer has not been called explicitly, we can change the status code:

```
if (!res.isCommitted()) {
    res.setStatus(HttpServletResponse.SC_OK);
} else {
```

It's too late to set the status code here so this method call will be ignored:

```
    res.setStatus(HttpServletResponse.SC_MOVED_TEMPORARILY);
}

// Continue returning the response and commit to the client
writer.println("Here is the end of the response");
writer.flush();
}
```

Set Status Example

Our next example demonstrates the outcome of setting various status codes and enables us to see how the client browser interprets the various codes. Note this example is part of the web application that accompanies this book (available on the accompanying CD).

In the following extract, from the `SetStatusCodeServlet`, we can see how a request parameter is used to pass the required status code for the response to the `setStatus()` method. Note, this example will demonstrate how browsers handle different status codes:

```
public class SetStatusCodeServlet extends HttpServlet{

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

        res.setContentType("text/html");

        // get the statusCode request parameter
        int statusCode = Integer.parseInt(request.getParameter("statusCode"));

        // set the status code
        res.setStatus(statusCode);
    }
}
```

In the next code snippet, the `SetStatusCodeServlet` also checks to see if the `statusCode()` parameter is either one of `HttpServletRequest.SC_MOVED_PERMANENTLY` or `HttpServletRequest.SC_MOVED_TEMPORARILY`. If so, it sets the `Location` header to another URL, which is used to indicate the new URL of the resource:

```
if (statusCode == HttpServletResponse.SC_MOVED_PERMANENTLY ||
    statusCode == HttpServletResponse.SC_MOVED_TEMPORARILY) {

    // Note the Location header must be an absolute URL so we need to
    // build a full URL
    StringBuffer sb = new StringBuffer("http://");
    sb.append(request.getServerName());
    sb.append(":"+request.getServerPort());
    sb.append(request.getContextPath());
    sb.append(request.getParameter("Location"));

    res.setHeader("Location", sb.toString());
}

PrintWriter writer = res.getWriter();
writer.println("<html>");
writer.println("<body>");
writer.println("<h1>Returning with HTTP Status Code " + statusCode
               + "</h1>");

writer.println("</body>");
writer.println("</html>");
writer.flush();
}
```

Run this example and see how the browser responds to the different status codes.

Luckily most servlets in use today do not have to explicitly set the status code of the response as modern containers will automatically call the `setStatus()` method on their behalf with a value of `SC_OK` (200). Should you wish to explicitly set a non-error status code, consider the most appropriate code available so that the browser may handle it correctly and display an informative page.

Return an HTTP error using the `sendError` method

As we discussed above, the `setStatus()` method should only be used to set non-error status codes. The reason for this is the existence of a couple of specially designed methods on the `HttpServletResponse` interface:

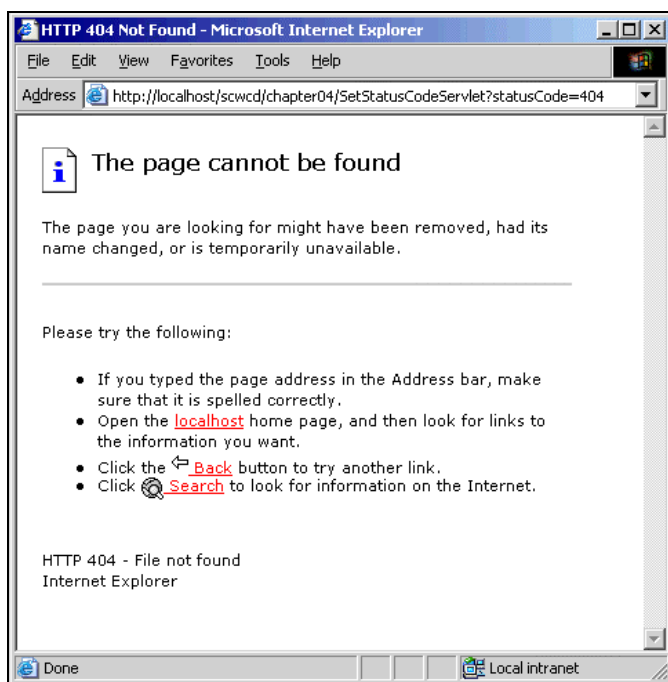
```
public void sendError(int statusCode)  
public void sendError(int statusCode, String message)
```

As the names suggest, these methods are designed to send responses back to the client to indicate some form of server-side error. As we can see, both methods take an HTTP status code in exactly the same way as the `setStatus()` method. The overloaded version of the method takes a string, which can be used to change the default error message accompanying each status code.

A major advantage of the `sendError()` method over the `setStatus()` method is that we can configure the container to return a custom error page that informs our users politely of the error while keeping the error page consistent with the look and feel of our application. This configuration is in the web application's deployment descriptor. We will see more of this shortly.

Let's have a look at the differences between using the `setStatus()` and `sendError()` methods to inform the user that a resource could not be found on the server via a 404, `HttpServletResponse.SC_NOT_FOUND` status code.

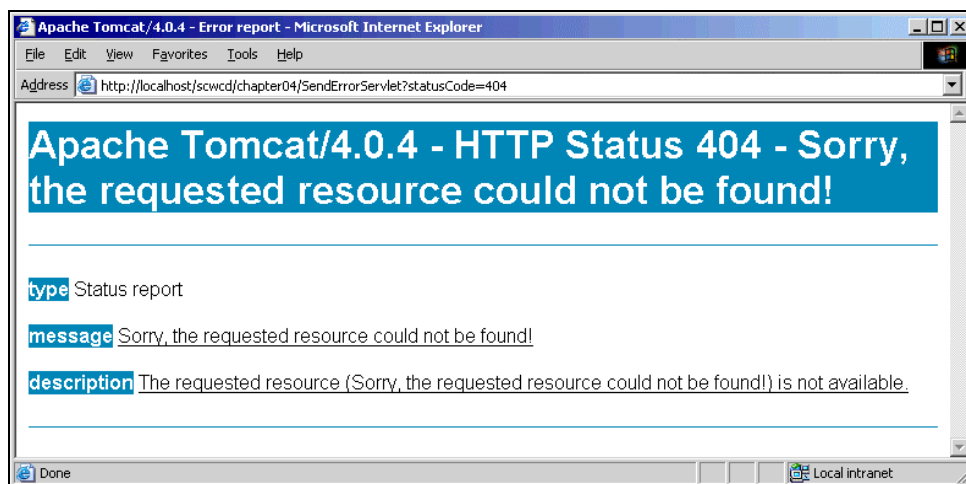
Firstly, using the `SetStatusCodeServlet` along with the `setStatus()` method, the browser displays the following:



As we can see, the browser displays its default error page informing the user that the requested resource could not be found. This is not the best way of informing users of the error.

Moving on to the second example; the `SendErrorServlet` uses the `sendError()` method to inform the user of the same 404 error:

```
res.sendError(statusCode,
    "Sorry, the requested resource could not be found!");
```



As we can see, the servlet container, Apache Tomcat in this example, recognizes that an error has occurred and generates its own server-specific error page informing the user of the error. Notice how the status code and the custom error message are clearly visible to the user. While this page is an improvement on the browser's default error page that we saw earlier, it is perhaps still not good enough for displaying errors to the users of our application.

Fortunately, the servlet container provides a mechanism that detects error conditions and, depending on the type of the error, returns a specific error page to the user. This gives the developer the opportunity to ensure that the look and feel of the application is consistent throughout the application; including error pages that help to increase user confidence in the application. This will be covered in detail in the next objective.

4.2 Given a set of business logic exceptions, identify the following:

As we mentioned briefly in the previous section, the servlet container gives web component developers a chance to trap error conditions on the server and to return custom error pages in the response, instead of relying on the default error page of a web server or browser.

The configuration that the deployment descriptor uses to handle each exception

The servlet container makes a distinction between two different error conditions on the server, those resources that return HTTP error status codes (such as 404 and 500) and those resources that throw Java exceptions. Both of these error conditions are catered for separately through declarative configuration in the deployment descriptor of the web application.

The deployment descriptor provides the following elements that are used to map HTTP status codes and Java exceptions to error pages:

```
<!ELEMENT error-page ((error-code | exception-type), location)>
<!ELEMENT error-code (#PCDATA)>
<!ELEMENT exception-type (#PCDATA)>
<!ELEMENT location (#PCDATA)>
```

The `<error-page>` element is used to map either an HTTP status code or a fully-qualified Java exception type to the location of a resource that will be returned to the user should the error occur. The value of the `<location>` element may point to a static resource (HTML page) or a dynamic resource (servlet or JSP page) but it must begin with a "/" since it is viewed as being relative to the application context.

Note that the `<error-page>` element exists beneath the `<web-app>` element but the web application DTD insists that the `<error-page>` element is located after any servlet context parameters, listener classes, servlet declarations, servlet mappings, and session configurations. Consult the web application DTD or take a look inside the `web.xml` deployment descriptor from the example web application that accompanies this book.

Specify the handling declaratively in the deployment descriptor

The code snippet below shows how to provide error handling for specific HTTP error status codes (in this case, 404):

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

In this case, whenever a servlet or JSP page uses the `sendError()` method to set the status code to 404, the `/error/404.jsp` JSP page will be returned to the user automatically.

In a similar manner, the snippet given below will forward any `ServletException` or any subclass of `ServletException` thrown by a servlet to the JSP page at location `/error/error.jsp`:

```
<error-page>
  <exception-type>javax.servlet.ServletException</exception-type>
  <location>/error/error.jsp</location>
</error-page>
```

Due to the large number of possible errors that can occur in a complex web application, it is often a good idea to provide a default error page. In this way any errors or exceptions that are not explicitly caught and handled by a specific `<error-page>` declaration are still handled in a consistent manner and no horrible stack traces are visible to the user.

It is advisable to use a dynamic resource (such as a servlet or JSP page) to provide the error page. The servlet container makes a limited amount of information available through request attributes sent to the resource. They are as follows:

- ❑ `javax.servlet.error.status_code`
This is an `Integer` representing the status code of the error
- ❑ `javax.servlet.error.message`
This is a `String` representation of the error message
- ❑ `javax.servlet.error.exception_type`
This is a `Class` object indicating the class of exception thrown

How to use a `RequestDispatcher` to forward the request to an error page

We have seen how simple it is to use the declarative `<error-page>` mechanism provided by the servlet container to respond to exceptions and error status codes. Sometimes it may be necessary to forward a request to an error page manually, perhaps without throwing an exception or setting an error status code. The `javax.servlet.RequestDispatcher`, introduced in Chapter 1 provides us with this functionality.

A common use of a `RequestDispatcher` and error pages is to check a session attribute, perhaps to see if a user has successfully logged in or if their session has timed out since their last interaction with the application. While both of these scenarios could be dealt with by throwing a custom exception and letting the container direct the exception to an error page, it may be more desirable to forward a request to an error page manually. The advantage of this technique is that attributes may be stored in the request before it is dispatched to the error page, a facility not available using the declarative approach:

```

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

    HttpSession session = req.getSession();
    User user = (User)session.getAttribute("user");

    if (user != null) {
        if (!user.isLoggedOn()) {
            RequestDispatcher disp = req.getRequestDispatcher("/error.jsp");
            disp.forward(req, res);
        }
        // otherwise continue as normal
    }
}

```

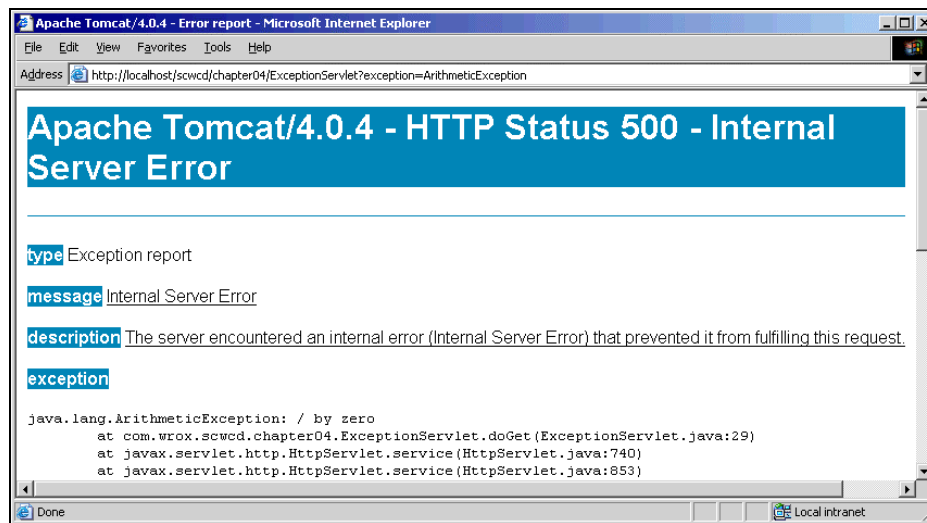
Here we can see how a servlet retrieves an object called user from the session, checks a condition upon it, and, if that condition is true, forwards the request to an error page (`/error.jsp`) using the `RequestDispatcher`.

Note that as an exception has not been thrown or status code set, the request attributes detailing the error will not be available to the error page.

For further explanation of the `RequestDispatcher` and its use refer to Chapter 1, *Objective 1.6*.

Error Page Servlet Example

Imagine a web application with a servlet that performs some form of numeric calculations based upon user input. Let's assume that this servlet was weak in validating user inputs and as a result an `ArithmeticException` was thrown. Without having any error pages configured for the web application, the user may be faced with the following stack trace returned from the server:



Most of you will agree that the above stack trace would hardly instill confidence in the application's users. A very simple default error page designed to handle any exception or status code could considerably improve this application and is surprisingly simple to build.

As mentioned earlier, the servlet container makes the status code, exception object, and message body of an error available as request attributes to a configured error page. All that an error page servlet would need to do is to retrieve this information and display it in a meaningful manner. The code snippet shown below does this:

```
public class ErrorPageServlet extends HttpServlet {

    // keys used to retrieve the error info from the request
    final String EXCEPTION = "javax.servlet.error.exception";
    final String MESSAGE = "javax.servlet.error.message";
    final String STATUS = "javax.servlet.error.status_code";

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

        // retrieve all the error information from the request
        Exception exception = (Exception) req.getAttribute(EXCEPTION);
        String message = (String) req.getAttribute(MESSAGE);
        Integer statusCode = (Integer) req.getAttribute(STATUS);

        PrintWriter out = res.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<h1>Oops! An Error Has Occurred</h1> " );
        out.println("<b>Status Code : </b>" + statusCode + "<br/>");
        out.println("<b>Exception Type :</b>" + exception.getClass());
        out.println("<br/><b>Message : </b>" + message+"<br/><hr/>");
        out.println("Please try again later");
        out.println("</body>");
        out.println("</html>");
        out.flush();
    }
}
```

This servlet simply retrieves the error information from the request and displays it to the user in a user-friendly manner. In a real-world application, this servlet would share the common stylesheet or design to maintain consistency.

Once the error page servlet is created it may be configured inside the deployment descriptor as follows:

```
<error-page>
  <exception-type>java.lang.ArithmeticException</exception-type>
  <location>/chapter04/ErrorPageServlet</location>
</error-page>
```

Here, the `<location>` element is configured to point to a URL that has a mapping for the `ErrorPageServlet` already set up. It should also be noted that the `<location>` element can accept any web application resource and our `ErrorPageServlet` could just as easily have been implemented using a JSP page providing the correct JSP directives were present. We will examine JSP directives in Chapter 8.

We can configure the `ExceptionHandlerServlet` to catch any exception or HTTP error code that we want because it only relies upon the request attributes sent to it. That is, it doesn't care what error occurred, it handles them all identically. In our case, only the `ArithmeticException` that we saw earlier and any subclasses will be caught.

Any further requests to our numeric calculating servlet that cause an `ArithmeticException` to be thrown will be directed to the `ExceptionHandlerServlet`. Here the error can be explained to the reader in a more user-friendly fashion and can offer advice on what caused the problem. Although the following error page is very basic, it does not confuse the user with a stack trace and is much more digestible.



It is always good practice to provide a set of error pages for your web applications and to configure their use through the deployment descriptor or by programmatically forwarding to the error pages using a `RequestDispatcher` object. Relying on the default error pages of the server is not a very good idea as they are implementation-specific and will not be consistent across different servers.

4.3 Identify the method used for the following:

So far we have seen how servlets can throw exceptions or return HTTP error codes to tell the user that all was not well with their request. We have also seen how the servlet container provides a declarative mechanism for dealing with such error conditions, thus informing the user of the problem in a clear and professional way.

While presenting any errors in a clear and consistent manner to a user is a key part of any web application, it is also important that the error is reported to the other relevant parties involved, such as the developers. A good application, either under development or in production, should take great care that errors are carefully logged so that the development team has a chance of knowing about and fixing any problems that arise.

Write a message to the WebApp log

A very handy logging method is provided by the `javax.servlet.ServletContext` interface that enables a message to be written to the log file of a web application:

```
public void log(String message)
```

This method simply prints a given message to the servlet's log file. In practice (at least with Tomcat) the outcome of the `log()` method may be found inside the servlet's log itself. Check your container's configuration for more information.

Write a message and exception to the WebApp log

The `javax.servlet.ServletContext` interface provides another very useful and complementary logging method:

```
public void log(String message, java.lang.Throwable t)
```

This method will not only print a message to the servlet's log file but will also log a `Throwable` object in the case of an error condition. The location of the servlet's log file is implementation-dependent but to give you some idea, the Apache Tomcat container stores its log file under `%CATALINA_HOME%/logs`.

A point worth noting is that `GenericServlet` (the base class of all servlets) provides two identical log methods to those shown above. These methods simply exist for convenience and allow servlet developers the luxury of just calling `log()` instead of `getServletContext().log()` from inside their servlets. Calls to the `GenericServlet`'s logging methods are simply delegated to the methods provided by the `ServletContext` interface.

Although these methods can be a great help, their use comes at a price. Verbose logging can considerably slow the performance of an application, and care should be taken to log only application errors or other critical information. Verbose logging can be beneficial during development but care should be taken not to add too much, otherwise performance will suffer.

The Error Page Servlet Revisited

We can easily improve the `ErrorPageServlet` that we saw earlier and enable it to log any exceptions or errors it deals with to the web application log. The required changes are as shown:

```
// log the error
ServletContext sc = getServletContext();

StringBuffer sb = new StringBuffer();
sb.append("An error has occurred : Status = ");
sb.append(statusCode);
sb.append(", Exception = ");
sb.append(exception.getClass());
sb.append(", Message = ");
sb.append(message);
sc.log(sb.toString());
```

The next time the `ErrorPageServlet` deals with an exception or error code, the support team can be sure that the error is logged to the application log file as follows:

```

localhost_Jog.2002-09-25.txt - Notepad
File Edit Format Help
2002-09-25 17:42:27 ErrorPageservlet: init
2002-09-25 17:42:27 An error has occurred : Status = 500, Exception =
class java.lang.ArithmeticException, Message = / by zero
    
```

As we can see from the last entry in the log file, our `ArithmeticException` has been successfully logged.

Summary

In this section we will recap all the core examination objectives covered in this chapter. To ensure your knowledge of the server-side exception handling mechanism, with regard to servlets, is up to examination standard, make sure you are confident with all the points raised here and can expand on the points where necessary:

Objectiv Number	Objective	Recap
4.1	For each of the following cases, identify correctly constructed code for handling business logic exceptions, and match that code with correct statements about the code's behavior:	
	Return an HTTP error using the setStatus method	The <code>HttpServletResponse</code> provides: <code>void setStatus(int statusCode)</code>
	Return an HTTP error using the sendError method	The <code>HttpServletResponse</code> provides: <code>void sendError(int statusCode)</code> <code>void sendError(int statusCode, String msg)</code>

Objective Number	Objective	Recap
4.2	<p>Given a set of business logic exceptions, identify the following:</p> <p>The configuration that the deployment descriptor uses to handle each exception:</p>	<pre> <error-page> <error-code>404</error-code> <location>/error/404.jsp</location> </error-page> or: <error-page> <exception-type> javax.servlet.ServletException </exception-type> <location>/error/404.jsp</location> </error-page> </pre>
	<p>How to use a RequestDispatcher to forward the request to an error page</p>	<p>The ServletRequest interface provides:</p> <pre> RequestDispatcher getRequestDispatcher(String loc) </pre> <p>This method accepts absolute or relative location parameters but must be within the same servlet context</p> <p>The ServletContext interface provides:</p> <pre> RequestDispatcher getRequestDispatcher(String loc) RequestDispatcher getNamedDispatcher(String loc) </pre> <p>These methods accept only absolute location parameters, which must begin with a "/" and are interpreted as relative to the context root.</p> <p>The RequestDispatcher interface provides:</p> <pre> void forward(ServletRequest req, ServletResponse res) </pre>

Objective Number	Objective	Recap
4.3	Identify the method used for the following: Write a message to the WebApp log	The ServletContext interface provides: void log(String msg)
	Write a message and exception to the WebApp log	The ServletContext interface provides: void log(String msg, Throwable t)

Sample Questions

1. Which classes or interfaces provide the `setStatus(int statusCode)` and `sendError(int statusCode, String message)` methods that can be used to send HTTP status codes to the client browser? Select one correct answer:
 - a. `HttpServletResponse` and `ServletResponse` respectively
 - b. Both are provided by `HttpServletResponse`
 - c. `ServletException` and `HttpServletResponse` respectively
 - d. Both are provided by `ServletResponse`

2. If a servlet experiences an error condition during its execution, which method should it use to send an error back to the client browser? Select one correct answer:
 - a. `ServletResponse.setError(int statusCode)`
 - b. `HttpServletResponse.setStatus(int statusCode)`
 - c. `HttpServletResponse.sendError(int statusCode, String message)`
 - d. `ServletException.sendError(String message, Throwable t)`

3. You wish to configure your application to forward to a specific error page (`/error.jsp`) whenever an `ArithmeticException` is thrown by any servlet or JSP page. Which deployment descriptor extract will do this for you, if at any? Select one correct answer:
 - a.

```
<error>
  <exception-type>java.lang.ArithmeticException</exception-type>
  <error-page>/error.jsp</error-page>
</error>
```

b.

```
<error-page>
  <status-code>500</status-code>
  <page-uri>/error.jsp</page-uri>
</error-page>
```

c.

```
<error-page>
  <exception-type>ArithmeticException</exception-type>
  <location>/error.jsp</location>
</error-page>
```

d. None of the above

4. Consider the following servlet code:

```
public class LoginServlet extends HttpServlet {

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

        try {
            int status = loginUser(req);
            // continue as normal...
        } catch (LoginException le) {
            // X
        }
    }
}
```

Assume that the `loginUser()` method is provided elsewhere in the servlet and is capable of throwing a `LoginException` if the user does not have valid credentials. Which line of code can be substituted for `// X` so that the error is forwarded to an error handling JSP page called `error.jsp` where the exception can be handled gracefully? Select one correct answer:

- a. `req.getRequestDispatcher("../error.jsp").forward(req, res);`
- b. `getServletContext().getRequestDispatcher("../error.jsp").forward(req, res);`
- c. `getServletContext().getNamedDispatcher("/error.jsp").redirect(req, res);`
- d. `req.getNamedDispatcher("../error.jsp").forward(req, res);`

5. Which of the following names can be used by a JSP page or a servlet to extract attributes from a request that is forwarded in the event of an error? Select three correct answers:

- a. `statusCode`
- b. `javax.servlet.error.status_code`
- c. `javax.servlet.http.message`

- d. `javax.servlet.error.message`
 - e. `javax.servlet.exception`
 - f. `javax.servlet.error.exception`
- 6.** Inside a deployment descriptor, which child element of the `<web-app>` tag is used to configure error-handling functionality? Select one correct answer:
- a. `<error-page>`
 - b. `<error>`
 - c. `<exception-type>`
 - d. `<status-code>`
- 7.** Which methods allow exceptions and messages to be written to the servlet's log? Select two correct answers:
- a. `ServletContext.log(String message, Throwable t)`
 - b. `HttpSession.log(String message, Exception e)`
 - c. `GenericServlet.log(String message, Throwable t)`
 - d. `ServletException.log(String message)`
- 8.** What is the result if the following servlet's `doPost()` method is called? Assume the response has been committed as a result of calling the `writer.flush()` method. Select one correct answer:

```
public class ServletY extends HttpServlet {  
  
    public void doPost(HttpServletRequest req, HttpServletResponse res)  
        throws ServletException, IOException {  
  
        PrintWriter writer = res.getWriter();  
        writer.println("Hello World");  
        writer.flush();  
  
        res.sendError(HttpServletResponse.SC_FORBIDDEN, "Bang!");  
    }  
}
```

- a. A `java.lang.IllegalStateException` is thrown
- b. A `NullPointerException` is thrown
- c. The browser displays its default `SC_FORBIDDEN` error page

- 9.** Which phrases are true when applied to a servlet forwarding a request to an error page when an exception is thrown during its execution? Select two correct answers:
- a.** This can be done by configuring a mapping between the error and the error page in the deployment descriptor
 - b.** This cannot be done without catching the exception explicitly in the servlet
 - c.** This cannot be done if the deployment descriptor has a `<istributable>` element
 - d.** This can be done even if the type of the exception is not known at run time

