8 Spring AOP Framework

In building enterprise applications, the tasks like logging, performance profiling, validation, error checking and handling, transaction management, and so on, are called *crosscutting concerns*, meaning that they are intermingled with the main business logic code dealt with a separate paradigm of *object oriented programming* (OOP). How to deal with such crosscutting concerns effectively and efficiently is an important task for building an enterprise application. *Aspect oriented programming* (AOP) is generally considered a standard practice for addressing such crosscutting concerns.

Spring has its own AOP framework, which are limited to the beans declared in its IoC container only. In addition, it supports integration with the AspectJ AOP framework (http://www.eclipse.org/aspectj). As an aspect oriented extension to the Java programming language, AspectJ is a very popular AOP framework for modularizing crosscutting concerns.

Throughout this chapter, we use a performance profiling framework (*perfBasic*) I described in my other text *Software Performance and Scalability: A Quantitative Approach* (Wiley, 2009) as a concrete crosscutting concern to demonstrate how to apply Spring AOP framework to solving real crosscutting concerns. You can apply the same AOP techniques illustrated in this chapter by substituting the perfBasic profiling concern with your crosscutting contexts.

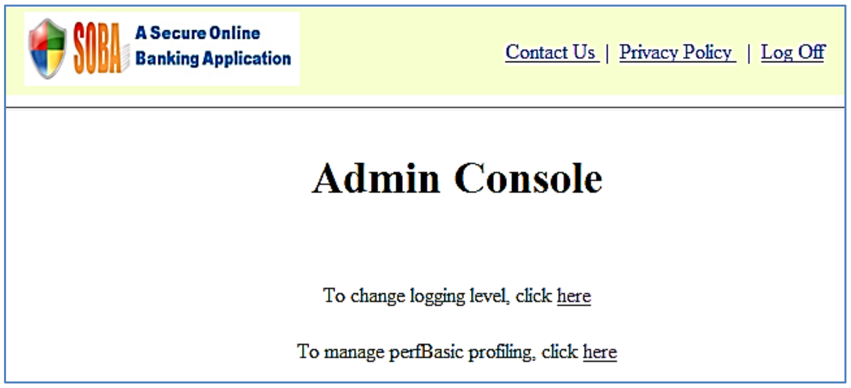
Let's begin with the admin console for SOBA next, which enables configuring log4j logging and *perfBasic* profiling for SOBA. Using SOBA admin console as a step stone, we illustrate how logging can be implemented with multiple approaches, from the most primitive one to the state-of-the-art one.

8.1 SOBA ADMIN CONSOLE – A RENDEZVOUS FOR CROSSCUTTING CONCERNS

Two crosscutting requirements need to be addressed for SOBA:

- Logging levels should be configurable dynamically.
- Performance profiling based on the *perfBasic* profiling framework should be configurable dynamically.

Figure 8.1 shows what the SOBA admin console looks like, which is what one would see after logging in with the user *appadmin/appadmin*. This admin user should have been created when the SOBA database was created initially by executing the *create_all.sql* script. If not, execute the *create_test_users.sql* script included in the *soba_db_scripts.zip* download file to get the admin user created.


A Secure Online Banking Application
Contact Us | Privacy Policy | Log Off

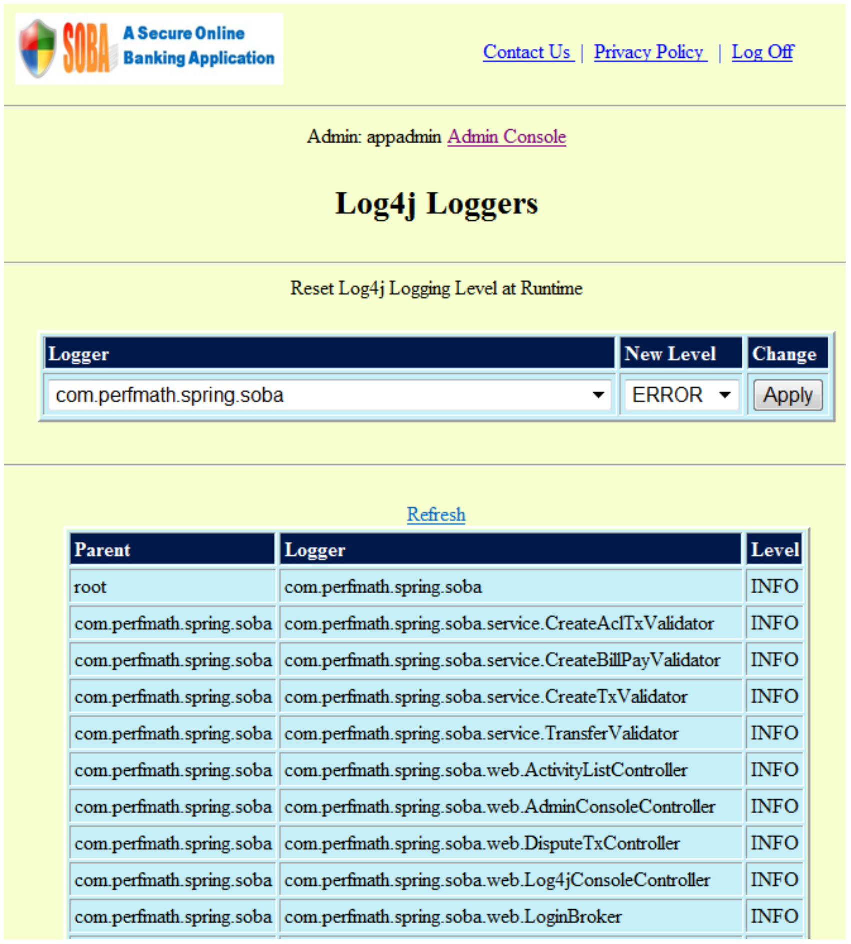
Admin Console

To change logging level, click here

To manage perfBasic profiling, click here

Figure 8.1 SOBA admin console.

A SOBA admin can change the logging level and manage perfBasic profiling, both at runtime on the admin console. Figures 8.2 and 8.3 show these two features.


A Secure Online Banking Application
Contact Us | Privacy Policy | Log Off
Admin: appadmin Admin Console

Log4j Loggers

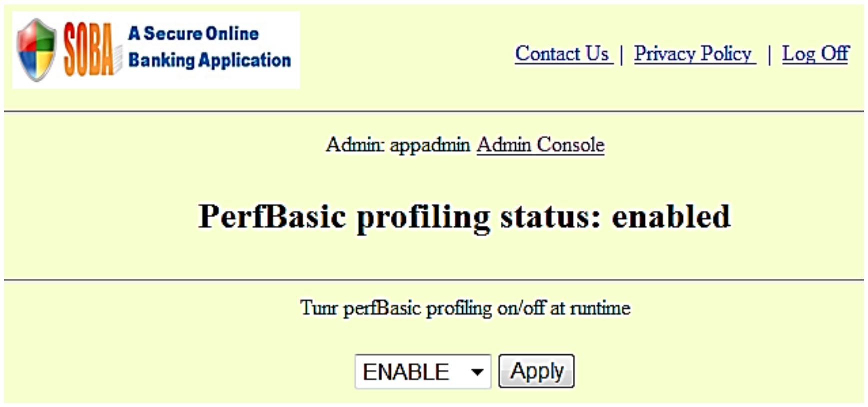
Reset Log4j Logging Level at Runtime

Logger	New Level	Change
com.perfmath.spring.soba	ERROR	Apply

Refresh

Parent	Logger	Level
root	com.perfmath.spring.soba	INFO
com.perfmath.spring.soba	com.perfmath.spring.soba.service.CreateAclTxValidator	INFO
com.perfmath.spring.soba	com.perfmath.spring.soba.service.CreateBillPayValidator	INFO
com.perfmath.spring.soba	com.perfmath.spring.soba.service.CreateTxValidator	INFO
com.perfmath.spring.soba	com.perfmath.spring.soba.service.TransferValidator	INFO
com.perfmath.spring.soba	com.perfmath.spring.soba.web.ActivityListController	INFO
com.perfmath.spring.soba	com.perfmath.spring.soba.web.AdminConsoleController	INFO
com.perfmath.spring.soba	com.perfmath.spring.soba.web.DisputeTxController	INFO
com.perfmath.spring.soba	com.perfmath.spring.soba.web.Log4jConsoleController	INFO
com.perfmath.spring.soba	com.perfmath.spring.soba.web.LoginBroker	INFO

Figure 8.2 Log4j loggers for SOBA.


A Secure Online Banking Application
Contact Us | Privacy Policy | Log Off
ENABLE
Apply

Admin: appadmin Admin Console

PerfBasic profiling status: enabled

Tunr perfBasic profiling on/off at runtime

Figure 8.3 PerfBasic admin console.

The logic behind the SOBA admin console works as follows:

- When you log in as *appadmin/appadmin*, after authentication, control is transferred to the `LoginBroker` controller, which redirects to `adminConsole.jsp` as listed in Listing 8.1. The `LoginBroker` controller is discussed when we discuss log4j in the next section.
- On the *admin console* page, control is transferred to `log4jConsole` controller or `perfBasicConsole` controller, depending on which URL is clicked. Listing 8.2 and Listing 8.3 show the `Log4jConcolseController` and `PerfBasicConsoleController` classes, respectively. As is seen, the `Log4jConcolseController` and `PerfBasicConsoleController` return to the `log4jConsole.jsp` and `perfBasicConsole.jsp`, which are shown in Listing 8.4 and Listing 8.5, respectively. You might want to take a quick look at these two jsp files to understand what Spring features they use and how they interact with the Spring controllers exactly.

Listing 8.1 adminConsole.jsp

```
<%@ include file="include.jsp" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>
<head>
<title>Admin Console</title>
</head>
<%@ include file = "banner.jsp" %>
<body>
<center>
<h1> Admin Console</h1>
<br> <br>
   To change logging level, click <a href="<c:url value="log4jConsole"/>">here</a> <p>
   To manage perfBasic profiling, click <a href="<c:url
                   value="perfBasicConsole"/>">here</a>
```

```
  <br> <br>
  </center>
</body>
</html>
```

Listing 8.2 Log4jConsoleController.java

```
package com.perfmath.spring.soba.web;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.SortedMap;
import java.util.TreeMap;
import java.util.Iterator;

import java.util.Map;
import org.apache.log4j.LogManager;
import org.apache.log4j.Logger;
import org.apache.log4j.Level;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.SessionAttributes;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;
import java.util.Enumeration;
import java.util.List;

import com.perfmath.spring.soba.util.MyLogger;

@Controller
@RequestMapping("/log4jConsole")
@SessionAttributes("log4j")
public class Log4jConsoleController extends AbstractController {

    @RequestMapping(value="/log4jConsole", method = RequestMethod.GET)
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        String loggerSelected = request.getParameter("loggerSelected");
        if (loggerSelected != null && loggerSelected.length() > 0) {
            String levelSelected = request.getParameter("levelSelected");
            setLogLevel(loggerSelected, levelSelected);
        }
```

```
            ModelAndView mav = new ModelAndView ();
            mav.setViewName("log4jConsole");
            String name = "";
            String effectiveLevel = "";
            String parent = "";
            Map<String, Object> model = new HashMap<String, Object> ();
            Enumeration loggers = LogManager.getCurrentLoggers();

            SortedMap sm = new TreeMap();

            while (loggers.hasMoreElements()) {
                Logger logger = (Logger) loggers.nextElement();
                name = logger.getName().toString();
                effectiveLevel = logger.getEffectiveLevel().toString();
                parent = logger.getParent().getName().toString();
                if (name.startsWith("com.perfmath")) {
                    sm.put (name, new MyLogger (name, effectiveLevel,parent));
                }
            }

            List<MyLogger> myLoggers = new ArrayList<MyLogger>();
            Iterator it = sm.keySet().iterator();
            while (it.hasNext() ) {
                Object key = it.next();
                myLoggers.add ((MyLogger) sm.get(key));
            }
            model.put ("myLoggers", myLoggers);
            mav.addObject ("model", model);
            return new ModelAndView("log4jConsole", model);
        }
        public void setLogLevel(String loggerSelected, String levelSelected) {
            Logger rootLogger = Logger.getRootLogger();
            Logger logger = rootLogger.getLoggerRepository().getLogger(loggerSelected);
            logger.setLevel((Level)Level.toLevel(levelSelected));
        }
    }
```

Listing 8.3 PerfBasicConsoleController.java

```
    package com.perfmath.spring.soba.web;

    import javax.servlet.http.HttpServletRequest;
    import javax.servlet.http.HttpServletResponse;
    import java.util.HashMap;
    import java.util.Map;
```

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.SessionAttributes;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;

import com.perfmath.spring.soba.util.PerfBasicUtil;

@Controller
@RequestMapping("/perfBasicConsole")
@SessionAttributes("perfBasic")
public class PerfBasicConsoleController extends AbstractController {

    @RequestMapping(value="/perfBasicConsole", method = RequestMethod.GET)
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        String profilingStatus = request.getParameter("profilingStatus");
        if (profilingStatus != null ) {
            resetProfiling(profilingStatus);
        }

        ModelAndView mav = new ModelAndView ();
        mav.setViewName("perfBasicConsole");

        Map<String, Object> model = new HashMap<String, Object> ();

        model.put ("profilingStatus", PerfBasicUtil.getProfilingStatus());
        mav.addObject ("model", model);
        return new ModelAndView("perfBasicConsole", model);
    }
    public void resetProfiling(String profilingStatus) {
        if (!PerfBasicUtil.getProfilingStatus().equals(profilingStatus)) {
            PerfBasicUtil.setProfilingStatus(profilingStatus);
            if (PerfBasicUtil.getProfilingStatus().equals("disabled")){
                PerfBasicUtil.flushApfWriter();
            }
        }
    }
}
```

Listing 8.4 log4jConsole.jsp

```
<%@ include file="include.jsp"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="security"
    uri="http://www.springframework.org/security/tags"%>
```

```
<html bgcolor='blue'>
<head>
<script language="javascript" src="init.js"></script>
<script>
    function initAll(form) {
        // Initialize all form controls
        with (form) {
            //initSelect (loggerSelected, "");
        }
    }
</script>
<title>Log4J Console</title>
</head>
<%@ include file="banner.jsp"%>
<body bgcolor="F7FFCE" onload="initAll(document.setLogLevel)">
    <center>
        <security:authorize ifAnyGranted="ROLE_ADMIN">
            <table>
                <tr><td>Admin: <security:authentication property="name" /></td>
                <td><a href="<c:url value="loginBroker"/>">Admin Console</a></td>
                </tr>
            </table>
        </security:authorize>
        <h2>Log4j Loggers</h2>
        <hr>
        <!-- reset log level for a given class -->
        Reset Log4j Logging Level at Runtime <br>
        <form name="setLogLevel" method="POST"
            action="<c:url value="/log4jConsole" />">
            <table border="2" CELLPADDING="2" CELLSPACING="2"
                    BGCOLOR="#C6EFF7">
                <c:forEach var="column" items="Logger, New Level, Change">
                    <th align="left" bgcolor="#00184A"><FONT
                    COLOR="#FFFFFF">${column}
                    </FONT></th>
                </c:forEach>
                <tr>
                    <td><select name="loggerSelected">
                            <c:forEach items="${myLoggers}" varStatus="status"
                                var="myLogger">
                                <option
                                value="${myLogger.name}">${myLogger.name}</option>
                            </c:forEach>
                    </select></td>
                    <td><select name="levelSelected">
```

```
                                <option value="debug">DEBUG</option>
                                <option value="info">INFO</option>
                                <option value="warn">WARN</option>
                                <option value="error" selected>ERROR</option>
                                <option value="fatal">FATAL</option>
                                <option value="off">OFF</option>
                        </select></td>
                        <td colspan="2" align="center">
                        <input type="submit" value="Apply" /></td>
                    </tr>
                </table>
            </form>
            <!--  display loggers -->
            <hr>
            <br> <a href="<c:url value="/log4jConsole.htm"/>"> Refresh </a> <br>
            <table border="2" CELLPADDING="2" CELLSPACING="2" BGCOLOR="#C6EFF7">
                <c:forEach var="column" items="Parent, Logger, Level ">
                    <th align="left" bgcolor="#00184A"><FONT
                        COLOR="#FFFFFF">${column}
                    </FONT></th>
                </c:forEach>
                <c:forEach items="${myLoggers}" varStatus="status" var="myLogger">
                    <tr>
                        <td>${myLogger.parent}</td>
                        <td>${myLogger.name}</td>
                        <td>${myLogger.effectiveLevel}</td>
                    </tr>
                </c:forEach>
            </table>
        </center>
    </body>
```

Listing 8.5 perfBasicConsole.jsp

```
    <%@page import="com.perfmath.spring.soba.util.PerfBasicUtil"%>
    <%@ include file="include.jsp"%>
    <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
    <%@ taglib prefix="security"
        uri="http://www.springframework.org/security/tags"%>
    <html bgcolor='blue'>
    <head>
    <title>PerfBasic Console</title>
    </head>
    <%@ include file="banner.jsp"%>
    <body bgcolor="F7FFCE">
        <center>
            <security:authorize ifAnyGranted="ROLE_ADMIN">
```

```
                <table>
                    <tr>
                        <td>Admin: <security:authentication property="name" />
                        </td>
                        <td><a href="<c:url value="loginBroker"/>">Admin
                            Console</a></td>
                    </tr>
                </table>
            </security:authorize>
            <h2>PerfBasic profiling status: ${profilingStatus}</h2>
            <hr> Turn perfBasic profiling on/off at runtime <br>
            <form name="resetProfiling" method="POST"
                action="<c:url value="/perfBasicConsole" />">
                <select name="profilingStatus">
                    <option value="enabled">ENABLE</option>
                    <option value="disabled">DISABLE</option>
                </select> <input type="submit" value="Apply" />
                </td>
            </form>
        </center>
    </body>
    </html>
```

Next, we describe how log4j and perfBasic profiling work with SOBA for logging and performance profiling.

8.1.1 Logging with Log4j for SOBA

First, let us understand how log4j works.

Log4j is a specific implementation that implements the generic logging interface defined by a common framework such as the *Apache Commons Logging* or SLF4J (Simple Logging Façade for Java). Other similar Java logging implementations include *Java Logging API* (`java.util.logging`) that comes with Oracle JREs and logback, which is intended as a successor to log4j, and so on.

To use Log4j with Spring, a `log4j.xml` file needs to be in place. It is initially placed in the *war/WEB-INF/classes* directory with the Ant-based Eclipse SOAB project or *src/main/resources* directory with the Maven-based Eclipse project, which is eventually placed in the *WEB-INF/classes* directory of a SOBA deployment on Tomcat.

Listing 8.6 shows the log4j configuration file for SOBA. As is seen, log4j has three main elements:

- **Appender**. An *appender* defines where the log should go – *Console* or *File*. In Listing 8.6, we see three appenders: one *Console* appender and two *File* appenders (File and PERF). With each file appender, file name and path are explicitly specified.

- **Layout**. A layout specifies the logging format, which heavily depends on the symbol of "%" sign followed by a letter. Common patterns include:
    - %d for date/time
    - %t for thread name
    - %m for user-defined message
    - %r for milliseconds since program start
    - %C for class name
    - %M for method name
    - %n for carriage return, etc.
- **Logger**. A *logger* associates a package with a log level. A logger can also have an *appender-ref* attribute specified specifically so that the log can be directed to a separate file. See the logger for *com.perfmath.spring.soba* defined below with PERF specified as the value for the *appender-ref* attribute. Note also the root logger defined in Listing 8.6, from which all other loggers inherit.

Listing 8.6 log4j.xml configuration file

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration>
 <appender name="CONSOLE" class="org.apache.log4j.ConsoleAppender">
  <layout class="org.apache.log4j.PatternLayout">
   <param name="ConversionPattern"
        value="%d [%t] %-5p %c - %m%n"/>
  </layout>
 </appender>
<appender name="FILE" class="org.apache.log4j.RollingFileAppender">
  <param name="File" value="../logs/soba/soba.log"/>
  <param name="Append" value="true"/>
  <layout class="org.apache.log4j.PatternLayout">
   <param name="ConversionPattern" value="%d{ISO8601} %t %-5p %c{2} - %m%n"/>
  </layout>
 </appender>
 <appender name="PERF" class="org.apache.log4j.RollingFileAppender">
  <param name="File" value="../logs/soba/soba_perf_log4j.log"/>
  <param name="Append" value="true"/>
  <layout class="org.apache.log4j.PatternLayout">
   <param name="ConversionPattern"
        value="%d{ISO8601}|API|%t|%r|%m%C.%M%n"/>
  </layout>
  <filter class="org.apache.log4j.varia.LevelRangeFilter" >
   <param name="LevelMin" value="INFO"/>
   </filter>
 </appender>
 <logger name="org.apache">
  <level value="INFO"/>
```

```
</logger>
<logger name="org.springframework.web">
  <level value="INFO"/>
    </logger>
      <logger name="org.springframework.security">
  <level value="INFO"/>
    </logger>
<logger name="com.perfmath.spring.soba">
  <level value="INFO"/>
    <appender-ref ref="PERF"/>
</logger>
<root>
<priority value="INFO" />
<appender-ref ref="FILE"/>
</root>
</log4j:configuration>
```

Next, let's use the `LoginBroker.java` file to demonstrate how log4j is used. As is seen in Listing 8.7, first, one needs to import the two *Apache Commons Logging* packages of `Log` and `LogFactory`. Then, define a log object to get the logger for this class and use `log.info (String message)` to log a message anywhere in an executable location of the class. Note the uses of `log.info (..)` statements at the beginning and end of the `getModelAndView` method. Note how SLF4J and `PerfBasicUtil.log ()` are used similarly. Perform a login with `appadmin/appadmin` and check the *soba_perf_log4j.log* file, which should have two lines similar to the following two lines I obtained on my system (note that each line has been broken into two below):

```
2012-07-18 00:29:08,932 |API| http-bio-8443-exec-
5|13769|+com.perfmath.spring.soba.web.LoginBroker.getModelAndView;
2012-07-18 00:29:08,932|API|http-bio-8443-exec-5|13769|-
com.perfmath.spring.soba.web.LoginBroker.getModelAndView;
```

One more thing with log4j you probably already know is there are six logging levels of `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, and `FATAL`. When you specify something like `log.info (..)` in your program, the message is logged only if the current log level is equal to or higher than "INFO". Otherwise, the message is not logged.

Listing 8.7 LoginBroker.java

```
package com.perfmath.spring.soba.web;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.slf4j.Logger;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.SessionAttributes;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;

import com.perfmath.spring.soba.util.*;
@Controller
@RequestMapping("/loginBroker")
@SessionAttributes("login")
public class LoginBroker extends AbstractController {
private Log log = LogFactory.getLog (this.getClass());
private Logger logger = org.slf4j.LoggerFactory.getLogger(this.getClass());
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
            HttpServletResponse response) throws Exception {

        ModelAndView mav = new ModelAndView ();
        mav= getModelAndView(request);
        return mav;
    }

    public ModelAndView getModelAndView(HttpServletRequest request) {
        PerfBasicUtil.log();
        log.info ("+");
        logger.info("Testing logging from SLF4J ... it works");
        String authority = SecurityContextHolder.getContext()
                .getAuthentication().getAuthorities().toString();
        String viewString = "";
        if (authority.contains("ADMIN")) {
            viewString = "adminConsole";
        } else if (authority.contains("REP")) {
            viewString = "repConsole";
        } else if (authority.contains("CUST") ) {
            viewString = "redirect:activityList";
        }
        log.info ("-");
        PerfBasicUtil.log();
        return new ModelAndView (viewString);
    }
}
```

You might have noticed the use of `PerfBasicUtil.log()` above and below `log.info (..)` as shown in Listing 8.7. We discuss what it is about in the next section.

8.1.2 Performance Profiling with perfBasic for SOBA

You might have noticed in Listing 8.6 log4j.xml that we have a PERF appender configured as follows:

```
<appender name="PERF" class="org.apache.log4j.RollingFileAppender">
 <param name="File" value="../logs/soba/soba_perf_log4j.log"/>
 <param name="Append" value="true"/>
 <layout class="org.apache.log4j.PatternLayout">
  <param name="ConversionPattern"
      value="%d{ISO8601}|API|%t|%r|%m%C.%M%n"/>
 </layout>
 <filter class="org.apache.log4j.varia.LevelRangeFilter" >
  <param name="LevelMin" value="INFO"/>
  </filter>
</appender>
```

Note that this appender specifies a logging format of **%d** for date, **%t** for thread name, **%r** for milliseconds since program startup, **%m** for user-defined message, and **%C.%M** for *className.methodName*. This format conforms to the perfBasic profiling logging format I once proposed in my other text *Software Performance and Scalability: A Quantitative Approach* (Wiley, 2009). However, there is a problem here: although all log data is written to a separate file, it cannot be independently turned on or off at runtime. Therefore, I needed to have it separated from log4j. That is why I have a `PerfBasicUtil.java` class and the associated `perfBasicConsoleController.java` and `perfBasicConsole.jsp` for using my perfBasic profiling framework with SOBA.

Listing 8.8 shows the `PerfBasicUtil.java` class, which illustrates how it helps implement the perfBasic profiling framework. Note the following:

- The attribute **profilingStatus**: This is the variable that controls how profiling can be turned on and off dynamically at runtime. The actual work is done in `PerfBasicConsoleController` shown in Listing 8.3 as follows, which specifies that if current `profilingStatus` and requested `profilingStatus` are not equal to each other, then reset it; also, the `ApfWriter` needs to be flushed to flush all performance log data after profiling is turned off.

```
public void resetProfiling(String profilingStatus) {
        if (!PerfBasicUtil.getProfilingStatus().equals(profilingStatus)) {
            PerfBasicUtil.setProfilingStatus(profilingStatus);
            if (PerfBasicUtil.getProfilingStatus().equals("disabled")){
                PerfBasicUtil.flushApfWriter();
            }
        }
    }
```

- The **log** method: This is the method that writes performance log data. Note how the class name and method name are extracted from the stack trace of the executing thread. This log method is what you see in Listing 8.7 **LoginBroker.java**.

Listing 8.8 PerfBasicUtil.java

```
    package com.perfmath.spring.soba.util;

    import java.io.FileWriter;
    import java.io.IOException;
    import java.io.PrintWriter;
    import java.net.*;

    public class PerfBasicUtil {

        private String apfFileName;
        private static String profilingStatus;
        private static PrintWriter apfWriter;

        public static String getProfilingStatus() {
            return profilingStatus;
        }

        public static void setProfilingStatus(String profilingStatus) {
            PerfBasicUtil.profilingStatus = profilingStatus;
        }

        public static void flushApfWriter() {

            if (apfWriter != null) {
                apfWriter.flush();
            }

        }

        public static void log() {
            StackTraceElement[] stacktrace = Thread.currentThread().getStackTrace();
            if (getProfilingStatus().equals("enabled")) {
                apfWriter.println("API" + "," + Thread.currentThread().getName()
                        + "," + System.currentTimeMillis() + "," + "unknownServer"
                        + "," + Thread.currentThread().getThreadGroup().getName()
                        + "," + "unknownClient" + "," + "unknownUser" + ","
                        + stacktrace[2].toString());
            }
        }
}
```

Now functional logging through *log4j* and performance logging through *perfBasic* are separated and can be controlled independently from each other – both dynamically. PerfBasic logging is not limited by logging levels any more. However, there is a common problem, which is where the concept of *crosscutting concern* comes in, that is, you have to insert those logging statements into every method you want to be logged manually. This is not efficient, for example, if you have 20k classes and on average five methods per class need to have those logging statements inserted, it would sum up to 200k statements (log4j and perfBasic together). For perfBasic, it's not a huge deal, because we only need to add at the beginning and end of a method and the logging statement is the same, namely, *PerfBasicUtil.log()*; which can be done by writing a simple program (in fact I did provide such a program when I first introduced perfBasic several years ago). Still, it would be more desirable if we could have other alternatives, which is what we would strive throughout the remainder of this chapter. Our first attempt would be with the method of dynamic proxy, as discussed in the next section.

8.2 SOLVING CROSSCUTTING CONCERNS WITH DYNAMIC PROXY

Proxy is a design pattern that belongs to the *structural pattern* category. With the proxy pattern, the caller calls the proxy and the proxy acts on behalf of the original object. Therefore, a proxy is basically a wrapper of an object. Figure 8.4 shows the concept of a proxy and a *proxied object*. A proxied object is also known as a *target object* or *advised object* as is explained later in context of AOP.

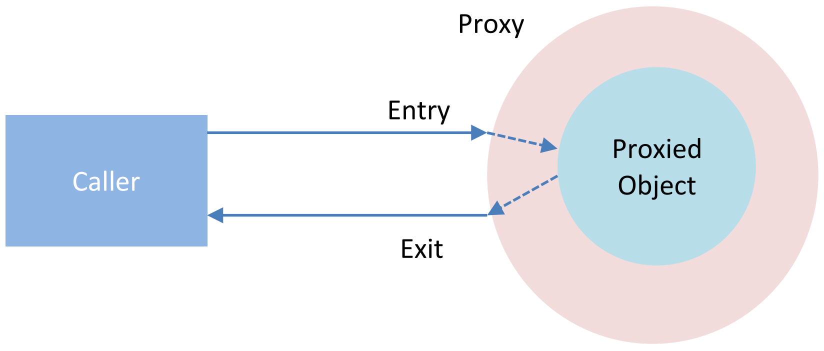


Figure 8.4 The concept of a proxy.

There are two types of proxies: static proxies and dynamic proxies. A static proxy works on a one-to-one basis that a proxy is needed for each interface and it is resolved at compile time. Obviously, this is worse than manually inserting those logging statements one by one in each method that requires logging.

A dynamic proxy is different that it is created at runtime dynamically and it works with any object. There are two types of dynamic proxies: those provided by the JDKs and CGLIB proxies (http://cglib.sourceforge.net – a powerful, high performance and quality Code Generation Library for extending Java classes and implementing interfaces at runtime). The difference between the two is that the JDK proxy requires at least one interface is implemented and only the methods declared in the interface are processed through the proxy, whereas the CGLIB proxy does not have this restriction. Therefore, if you look at the `LoginBroker.java` class shown in Listing 8.7, it's clear that a JDK-based dynamic proxy would not work and only a CGLIB dynamic proxy would work. It appears that all classes in the `services` and `dao` packages do follow the interface-implementation pattern, and therefore a JDK-based dynamic proxy should work, but that's not the case either, because those objects are *injected* by Spring under the hood and we do not create them externally. These constraints become clear after we demonstrate how to apply JDK-based dynamic proxies to classes outside the Spring framework next.

Let us now demonstrate how to apply JDK-based dynamic proxies to non-Spring-based classes. In order to demonstrate this, we have a few classes created in the `com.perfmath.spring.soba.aop.proxy` package. The first class is the `PerfBasicLoggingHandler.java` in this package. This class is shown in Listing 8.9. It is seen that:

- It does not import any Spring or Apache packages and it's purely standard Java without vendor specific packages.
- It implements the `InvocationHandler` from the `java.lang.reflect` package.
- It uses a `createProxy` method to create a proxy for a target object.
- It has an `invoke` method that would invoke the method of the target object. You can put your logging statements before and after the `method.invoke` (..) statement, but for perfBasic logging, we only need to insert a statement after the statement, because we only want to know how long it took for that `method.invoke` (..) method to be executed as an approximate time for the elapsed time of that method. With the `PerfBasicUtil.java` class, we could no longer use the *stacktrace* to extract the class and method of the profiled object, because the *stacktrace* has become more complicated due to the introduction of proxies. Therefore, we extract such information based on what's available in the *handler's invoke* method, as shown in Listing 8.10.

Listing 8.9 PerfBasicLoggingHandler.java

```
package com.perfmath.spring.soba.aop.proxy;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class PerfBasicLoggingHandler implements InvocationHandler {

   public static Object createProxy(Object target) {
      return Proxy.newProxyInstance(
            target.getClass().getClassLoader(),
            target.getClass().getInterfaces(),
            new PerfBasicLoggingHandler(target));
   }
   private Object target;

   public PerfBasicLoggingHandler(Object target) {
      this.target = target;
   }
```

```
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
     long startTime = System.currentTimeMillis();
      Object result = method.invoke(target, args);
      PerfBasicUtil.log(target.toString() + "@" + method.getName() +
            ":" + (System.currentTimeMillis() - startTime));
      return result;
    }
  }
```

Listing 8.10 A new method of PerfBasicUtil.log (String message) to be used with JDK-based dynamic proxy

```
  // for proxy-based calls
  public static void log(String message) {
        int index1 = message.indexOf("@");
        int index2 = message.lastIndexOf("@");
        if (index1 > 0) {
            message = message.substring (0, index1) + "." + message.substring (index2 +
1);
        }
        if (getProfilingStatus().equals("enabled")) {
            String data = "API" + "," + Thread.currentThread().getName()
                    + "," + System.currentTimeMillis() + "," + "unknownServer"
                    + "," + Thread.currentThread().getThreadGroup().getName()
                    + "," + "unknownClient" + "," + "unknownUser" + ","
                    + message;
            if (apfWriter != null) {
                apfWriter.println(data);
            } else {
                System.out.println (data);
            }
        }
     }
```

Now let's say we want to profile a `MySQLJDBCConnection` object that checks and measures how long it takes to create a new JDBC connection. According to what we explained about what a JDK-based dynamic proxy can and can't do, we have to make it work as follows:

- Creating an interface, which is defined in `IJDBCConnection.java` as shown in Listing 8.11. This is a simple interface, so we don't need to explain about it.
- Creating an implementation of the `IJDBCConnection` interface, which is the `MySQLJDBCConnection.java` class as shown in Listing 8.12. This is also a very simple Java class and we don't need to explain about it.
- Creating a `JDBCConnectionTest.java` class, which does the actual work of testing the JDBC connection with a given implementation, which is the

MySQLJDBCConnection.java class in this case, as is shown in Listing 8.13. This is how this JDBCConnectionTest.java class works :

- First, a mySQLJDBCConnection object is instantiated as usual.
- Then, the object is passed to the PerfBasicLoggingHandler's createProxy method to create a jdbcConnection object, which is the target object to be worked on.

This example shows all the logistics necessary for using JDK-based dynamic proxies to solve a crosscutting concern. Listing 8.14 shows the output of a test run, indicating that the proxy-based call to create a JDBC connection against a MySQL server on an Intel i7-2600 system took 170 milliseconds.

This dynamic proxy worked fine when executed standalone on Eclipse and produced the expected output as shown in Listing 8.14. However, when it was built with Maven and deployed with the rest of SOBA on Tomcat 7, SOBA could not be started up, due to the following error:

```
Jul 22, 2012 7:47:00 PM org.apache.catalina.core.StandardContext startInternal
SEVERE: Error listenerStart
Jul 22, 2012 7:47:00 PM org.apache.catalina.core.StandardContext startInternal
SEVERE: Context [/soba] startup failed due to previous errors
Jul 22, 2012 7:47:00 PM org.apache.catalina.loader.WebappClassLoader
    clearReferencesJdbc
SEVERE: The web application [/soba] registered the JDBC driver
[com.mysql.jdbc.Driver] but failed to unregister it when the web application was
stopped. To prevent a memory leak, the JDBC Driver has been forcibly unregistered.
Jul 22, 2012 7:47:00 PM org.apache.catalina.loader.WebappClassLoader
clearReferencesThreads
SEVERE: The web application [/soba] appears to have started a thread named
[MySQL Statement Cancellation Timer] but has failed to stop it. This is very likely to
create a memory leak.
```

The above error is reported here just to show that AOP programming is a lot more challenging than regular Java programming. It has very stringent rules to follow and associated error messages are less indicative. After hours of debugging, I finally resolved it by making the following changes:

- Changed the arguments for checkConnection show in Listing 8.11 into:

```
checkConnection (String connectionDescriptor)
```

- Changed the checkConnection implementation in Listing 8.12 into:

```
public class MySQLJDBCConnection implements IJDBCConnection {
    public Connection checkConnection(String connectionDescriptor) {
        Connection conn = null;
        long startTime = System.nanoTime ();
        String[] part = new String[4];
        StringTokenizer st = new StringTokenizer(connectionDescriptor, "+");
        int i = 0;
```

```
            while (st.hasMoreElements()) {
                part[i] = st.nextElement().toString();
                i++;
            }
        try {
                Class.forName(part[0]).newInstance();
                conn = DriverManager.getConnection(part[1], part[2], part[3]);
        }
        catch (Exception e) {
            System.out.print ("Cannot connect to database server: " + e.getMessage());
            }
        return conn;
      }
```

■ Changed the `checkConnection` call in Listing 8.13 into:

```
String connectionDescriptor = "com.mysql.jdbc.Driver+"
                    + "jdbc:mysql://localhost:3306/soba31+" + "soba31admin+"
                    + "soba31admin";
        conn = jdbcConnection.checkConnection(connectionDescriptor);
```

This trick helped resolve the above error magically (I figured that somehow *it* knew that a MySQL connection was being made with given *driver*, *url*, *username*, and *password*, so I won't let it know that by introducing a `StringTokenizer` to reconstruct all pieces of information from one String object of `connectionDescriptor`).

Listing 8.11 IJDBCConnection interface

```
package com.perfmath.spring.soba.aop.proxy;
import java.sql.Connection;
public interface IJDBCConnection {
public Connection checkConnection (String username,
String password, String url, String driver);
}
```

Listing 8.12 MySQLJDBCConnection.java class that implements the IJDBCConnection interface

```
package com.perfmath.spring.soba.aop.proxy;

import java.sql.Connection;
import java.sql.DriverManager;

public class MySQLJDBCConnection implements IJDBCConnection {
    public Connection checkConnection(String username, String password, String url,
            String driver) {
        Connection conn = null;
     try {
        Class.forName (driver).newInstance ();
        conn = DriverManager.getConnection (url, username, password);
     }
     catch (Exception e) {
        System.out.print ("Cannot connect to database server: " + e.getMessage());
        }
     return conn;
    }
}
```

Listing 8.13 JDBCConnectionTest.java

```
    package com.perfmath.spring.soba.aop.proxy;
    import java.sql.*;

    import com.perfmath.spring.soba.util.PerfBasicUtil;

    public class JDBCConnectionTest
    {
      public static void main (String[] args)
      {
        Connection conn = null;
        try {
           IJDBCConnection mySQLJDBCConnection = new MySQLJDBCConnection ();
           IJDBCConnection jdbcConnection =
                (IJDBCConnection) PerfBasicLoggingHandler.createProxy
               (mySQLJDBCConnection);
           conn = jdbcConnection.checkConnection("soba31admin", "soba31admin",
                  "jdbc:mysql://localhost:3306/soba31", "com.mysql.jdbc.Driver");
           if (conn != null) {
          System.out.println ("   --> calling by proxy completed with " +
               " database connection established");
           }
        } catch (Exception e) {
          System.err.println ("Cannot connect to database server: " + e.getMessage());
        } finally {
          if (conn != null) {
            try
            {
              conn.close ();
              System.out.println ("   --> Database connection terminated");
            }
            catch (Exception e) { /* ignore close errors */ }
          }
        }
        PerfBasicUtil.setProfilingStatus("disabled");
    }
  }
```

Listing 8.14 The Output of a JDBCConnectionTest Run

```
API,main,1342849773512,unknownServer,main,unknownClient,unknownUser,
com.perfmath.srping.soba.aop.proxy.MySQLJDBCConnection.checkConnection:172
  --> calling by proxy completed with  database connection established
  --> Database connection terminated
```

Dynamic proxy does not require inserting crosscutting statements in every class or method of an application. However, as you can see from Listing 8.13, it requires manually creating an object through the `createProxy` method of a crosscutting handler. What if an application consists of 20k or 100k classes? And what if an application has already been developed and in use for quite some time and it's not so easy to modify it? And as you can see that it's even hard to use with SOBA because all objects are *injected*. A better, preferred solution is to use Spring AOP-AspectJ support, as is discussed in the next section.

8.3 SOLVING CROSSCUTTING CONCERNS WITH SPRING AOP-ASPECTJ SUPPORT

Before getting deep with Spring AOP and its support for AspectJ, let's first review some of the basic AOP concepts next.

8.3.1 Basic AOP Concepts

The following concepts are essential for understanding AOP in general:

- **Joinpoint**: A joinpoint is basically an execution point in a program. You can consider it defining *where* an event occurs, for example, before/after a method execution event, or after an exception is thrown, etc., or where you could insert crosscutting statements if it was done in an intrusive way.
- **Pointcut**: A pointcut is a predicate for matching desired joinpoints. A pointcut defines on which package/class/method to apply an action or *advice* in AOP's term as explained next. You can consider a pointcut a match-maker between an AOP advice and one or more joinpoints. Spring uses the AspectJ *pointcut expression* by default.
- **Advice**: An advice advises what action to take at well-defined joinpoints matched by one or more pointcuts. Spring supports the following five types of advices:
    - *Before advice*: Advice to be executed *before* a method execution.
    - *Around advice*: Advice to be executed around a join point such as a method invocation.
    - *After returning advice* : Advice to be executed after a join point completes normally without throwing an exception.
    - *After throwing advice*: Advice to be executed after a method exits due to an exception thrown.

- *After (finally) advice*: Advice to be executed regardless of whether an exception is thrown or not.

■ **Aspect**: An aspect is a modularization of a crosscutting concern that applies to more than one class. Spring offers two approaches to implementing an aspect class: using the @AspectJ annotation style or using a schema-based approach that is independent of AspectJ. The @AspectJ style is preferred in general, because it gives more flexibilities to tap into the vast possibilities of solving crosscutting concerns than Spring's own schema-based approach. Later in this section, we present an example of using the @AspectJ style to implement an aspect class. But it's good to keep in mind the limitations with Spring's own AOP that:

- It is limited to the classes that implement an interface.
- It is limited to method execution joinpoints only.
- It is limited to Spring beans only.

■ **Target object**: A target object is the object to receive advices from one or more aspects. Because Spring AOP is implemented using runtime proxies, a target object is also called a *proxied* object or *advised* object, to be more pertinently.

■ **Introduction**: An introduction provides opportunities for adding additional attributes and methods on behalf of a type or introducing new interfaces to advised objects. In the AspectJ community, an introduction is known as an inter-type declaration.

■ **Weaving**: This is a mechanism for linking or weaving aspects with other application types or objects to create an advised object. Although weaving can be done at compile time, load time, or runtime, Spring AOP *weaves* at runtime.

Instead of giving each permutation a light touch, let us go through a concrete, easier to understand example next to help explain the APO concepts and the most likely approach you would use if you really need to use AOP a lot with your product, that is, how Spring supports AspectJ framework within its own AOP framework. You should refer to the Spring reference documentation and AspectJ texts to learn more if AOP is really a big deal to your project.

8.3.2 Spring and AspectJ Mixed AOP

My suggested approach to learning anything new is to focus on one good example, understand the concepts and the underlying mechanism really well, and then expand. The example presented next follows this guideline.

To illustrate how a Spring and AspectJ mixed AOP scenario might work, we focus on the *around advice*. This scenario consists of the following four classes:

1 A `JDBCConnection` Class: This `JDBCConnection.java` class is shown in Listing 8.15. It has two methods: a method named `getConnection` with a given connection descriptor and a method named `returnConnection` with a given open connection to close it.

2 A `ManageJDBCConnection` Class: This `ManageJDBCConnection.java` class is shown in Listing 8.16. It depends on the preceding `JDBCConnection` class to manage JDBC connections. It has three methods: `openConnection`, `closeConnection`, and `setJdbcConnection`. The first two methods are for normal operations with JDBC

connections, while the last method auto-wires it to the dependent class of `JDBCConnection`.

3 A `PerfBasicLoggingAspect` Class: This `PerfBasicLoggingAspect.java` class is shown in Listing 8.17. This is the *aspect* to be applied to an advised object. It defines two *pointcuts* and one *around advice* as follows:

- o The first pointcut specifies the `getConnection (String)` method from all classes in the `com.perfmath.spring` package with an argument of `connDescriptor` of *String* type, which matches that method of the `JDBCConnection` class.
- o The second pointcut specifies *all* method executions of the Spring beans whose names are prefixed with `'jdbcConnection'`. This becomes clear after we look at the bean configuration file for this example next.

Listing 8.18 shows the Spring bean configuration file `springaopaspectj.xml` for this example. In addition to the relevant schemas declared, there are only two tags: One is to specify @AspectJ autoproxy with `<aop:aspectj-autoproxy>` and the other is `<context:component-scan>`. The first tag enables Spring to create proxies for advised objects automatically (remember that it's always the proxies that work behind the scene), whereas the second tag instructs Spring to scan beans from the base package of `com.perfmath.spring.soba.aop`, which is where the advised objects reside. By default, the JDK-based dynamic proxies are created. If you rewrite the first tag like `<aop:aspectj-autoproxy proxy-target-class ="true"/>`, the CGLIB dynamic proxies will be created instead.

Listing 8.19 shows the `AOPDriver.java` program that is used to drive the test. It first creates an application context from the `springaopaspectj.xml` Spring bean configuration file. It then gets the Spring bean named `manageJDBCConnection` from the `context` created in the preceding step. This is made possible with the `@Component` annotation in Listing 8.16 `ManageJDBCConnection.java` and the tag `<context:component-scan>` in Listing 8.18 `springaopaspectj.xml`. It then calls the `openConnection` and `closeConnection` methods to drive the test. Listing 8.20 shows the output with the JDK-based dynamic proxies, while Listing 8.21 shows the output with the CGLIB-based dynamic proxies, both were obtained on an Intel i7-2600 based system. It is seen that there is not much difference in performance between these two different types of dynamic proxies (125 ms versus 126 ms). However, they were significantly faster than the non-AOP based approach, which took 172 ms as shown in Listing 8.14. If you want to learn more about Spring AOP and AspectJ, refer to the *Recommended Reading* section at the end of this chapter.

This wraps up our AOP chapter.

Listing 8.15 JDBCConnection.java

```
package com.perfmath.spring.soba.aop.aspectj;

import java.sql.Connection;
import java.sql.DriverManager;
import java.util.StringTokenizer;
import org.springframework.stereotype.Component;

@Component("jdbcConnection")
public class JDBCConnection {

    public Connection getConnection(String connectionDescriptor) {
        long startTime = System.nanoTime ();
        String[] part = new String[4];
        StringTokenizer st = new StringTokenizer(connectionDescriptor, "+");
        int i = 0;
        while (st.hasMoreElements()) {
            part[i] = st.nextElement().toString();
            i++;
        }
        System.out.println("   --> Parsing took  " +
                (System.nanoTime () - startTime)/1000 +
                " microseconds inside JDBCConnection");
        Connection conn = null;
        try {
            Class.forName(part[0]).newInstance();
            conn = DriverManager.getConnection(part[1], part[2], part[3]);
            System.out.println("   --> Connect to database server " + part[1]
                    + " successful");
        } catch (Exception e) {
            System.out.println("Cannot connect to database server: "
                    + e.getMessage());
        }
        System.out.println("   --> getConnection took  " +
           (System.nanoTime () - startTime)/1000000 +
            " ms inside JDBCConnection");
        return conn;
    }

    public void returnConnection(Connection conn) {
        if (conn != null) {
            try {
                conn.close();
                System.out.println("  --> Database connection terminated");
            } catch (Exception e) {
            }
        }
    }
}
```

Listing 8.16 ManageJDBCConnection.java

```
package com.perfmath.spring.soba.aop.aspectj;
```

```
import java.sql.Connection;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component("manageJDBCConnection")
public class ManageJDBCConnection {
    private JDBCConnection jdbcConnection;

    public Connection openConnection(String connectionDescriptor) {
        Connection conn = jdbcConnection.getConnection(connectionDescriptor);
        return conn;
    }

    public void closeConnection(Connection conn) {
        jdbcConnection.returnConnection(conn);
    }

    @Autowired
    public void setJdbcConnection(JDBCConnection jdbcConnection) {
        this.jdbcConnection = jdbcConnection;
    }
}
```

Listing 8.17 PerfBasicLoggingAspect.java

```
package com.perfmath.spring.soba.aop;

import java.util.Arrays;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;

@Component
@Aspect
public class PerfBasicLoggingAspect {
    @Pointcut("execution(* com.perfmath.spring..getConnection*(String))
&& args(connDescriptor)")
    public void getConnectionExecution(String connDescriptor) {
    }
    @Pointcut("bean(jdbcConnection*)")
    public void inJDBCConnection() {
    }

    @Around("getConnectionExecution(connDescriptor) && inJDBCConnection ()")
    public Object logAround(ProceedingJoinPoint joinPoint, String connDescriptor)
            throws Throwable {
        long startTime = System.currentTimeMillis();
        String signature = joinPoint.getTarget().getClass().getName() + "."
                + joinPoint.getSignature().getName();
        try {
            System.out.println("perfBasic logging: startTime=" + startTime
                    + " for " + signature);
            Object result = joinPoint.proceed();
            long endTime = System.currentTimeMillis();
            String logData = "perfBasic logging: endTime=" + endTime + " for "
                    + signature + ":elapsed time=" + (endTime - startTime)
                    + " ms";
            System.out.println(logData);
            return result;
        } catch (IllegalArgumentException e) {
            System.out.println("Illegal argument "
                    + Arrays.toString(joinPoint.getArgs()) + " in "
                    + joinPoint.getSignature().getName() + "()");
            throw e;
        }
    }
}
```

Listing 8.18 springaopaspectj.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-3.1.xsd
        http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
        http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.1.xsd">

    <!-- “false” for JDK-generated proxies (default) -->
    <aop:aspectj-autoproxy proxy-target-class="false" />

    <!-- “true” for CGLIB generated proxies -->
```

```
        <!-- aop:aspectj-autoproxy proxy-target-class="true" / -->

        <context:component-scan base-package="com.perfmath.spring.soba.aop.aspectj"
/>

    </beans>
```

Listing 8.19 AOPDriver.java

```
    package com.perfmath.spring.soba.aop.aspectj;

    import org.springframework.context.ApplicationContext;
    import org.springframework.context.support.ClassPathXmlApplicationContext;
    import java.sql.Connection;

    import com.perfmath.spring.soba.aop.ManageJDBCConnection;

    public class AOPDriver {
        public static void main(String[] args) {
            {
                ApplicationContext context = new ClassPathXmlApplicationContext(
                        "springaopaspectj.xml");

                ManageJDBCConnection jdbcConnectionTest =
                    (ManageJDBCConnection) context
                        .getBean("manageJDBCConnection");
                String connectionDescriptor = "com.mysql.jdbc.Driver+"
                        + "jdbc:mysql://localhost:3306/soba31+" + "soba31admin+"
                        + "soba31admin";
                Connection conn = jdbcConnectionTest
                        .openConnection(connectionDescriptor);
                jdbcConnectionTest.closeConnection(conn);
            }
        }
    }
```

Listing 8.20 Output from an AOP Test Run with JDK-Based Dynamic Proxies

```
perfBasic logging: startTime=1342828778820 for
com.perfmath.spring.soba.aop.aspectj.JDBCConnection.getConnection

  --> Parsing took  30 microseconds inside JDBCConnection

  --> Connect to database server jdbc:mysql://localhost:3306/soba31 successful

  --> getConnection took  108 ms inside JDBCConnection
```

```
perfBasic logging: endTime=1342828778945 for com.perfmath.spring.soba.aop.
aspectj.JDBCConnection.getConnection:elapsed time=125 ms

 --> Database connection terminated
```

Listing 8.21 Output from an AOP Test Run with CGLIB-Based Dynamic Proxies

```
perfBasic logging: startTime=1342829216823 for com.perfmath.spring.soba.aop.
aspectj.JDBCConnection.getConnection

  --> Parsing took  30 microseconds inside JDBCConnection

  --> Connect to database server jdbc:mysql://localhost:3306/soba31 successful

  --> getConnection took  111 ms inside JDBCConnection

perfBasic logging: endTime=1342829216949 for com.perfmath.spring.soba.aop.
aspectj.JDBCConnection.getConnection:elapsed time=126 ms

 --> Database connection terminated
```

8.4 SUMMARY

In this chapter, we introduced the concept of crosscutting concerns such as logging, validation, security, transaction management, exception handling, etc., that have to be dealt with in developing enterprise applications. Examples of SOBA functional logging and performance logging based on the perfBasic profiling framework I proposed several years ago were used to help corroborate those crosscutting concerns.

Then, we focused on two approaches to solving crosscutting concerns: The proxy-based approach and the AOP-based approach. From an application development point of view, the AOP-based approach is more flexible and practical, and out of many AOP-based implementations, Spring-AspectJ mixed implementation is arguably the best implementation strategy for Spring-based enterprise Java applications. We presented a concrete example to help explain the AOP concepts such as a joinpoint, pointcut, advice, aspect, and so on. The example also illustrated the procedure to develop and apply an aspect for solving a crosscutting concern.

Apparently, Spring AOP is a broad subject that is hard to explain all in one chapter of a book. If you really need to use AOP with your product, you need to study further, as explained in the Recommend Reading section next.

RECOMMENDED READING

The Spring Reference Documentation 3.1 covers the following AOP topics in chapters 8 and 9 (the topics we covered are highlighted):

- Chapter 8 Aspect Oriented Programming:
    - **8.1 Introduction**
    - **8.2 @AspectJ support**
    - 8.3 Schema-based AOP support

- **8.4 Choosing which AOP style to use**
- 8.5 Mixing aspect types
- **8.6 Proxying mechanisms**
- 8.7 Programmatic creation of @AspectJ Proxies
- 8.8 Using AspectJ with Spring applications
- 8.9 Further resources

■ Chapter 9 Spring AOP APIs

If you are interested in exploring more Spring AOP features, you can continue to try them out with SOBA. In fact, there is a self-review exercise next that I hope you can try it. AOP is a very powerful concept for solving large scale crosscutting concerns, but programming it is not easy at all. If you have never worked on it, you will know it after you try the AOP programing exercise 8.3 as described next.

SELF-REVIEW EXERCISES

8.1 What are the major differences conceptually between JDK-based dynamic proxies and CGLIB-based proxies?

8.2 What are the major limitations with pure Spring AOP implementations?

8.3 Modify the *aspect* listed in section 8.3 so that it would work for all methods of all classes to be proxied.