1. THE WEB TIER

Excerpted from: *Designing Enterprise Applications with the J2EE Platform* by Inderjeet Singh, Beth Stearns, Mark Johnson, Enterprise Team

THE PURPOSE OF THE WEB TIER

WEB-TIER TECHNOLOGIES
- Traditional Web-Tier Technologies
- Web-Tier Technologies in the J2EE Platform
- The Web Container
- Java Servlets
- JavaServer Pages (JSP) Technology
- Web-Tier Technology Guidelines

WEB-TIER APPLICATION STRUCTURE

WEB-TIER APPLICATION FRAMEWORK DESIGN
- Structuring the Web Tier
- Web-Tier MVC Controller Design
- Web-Tier MVC View Design
- Web-Tier MVC Model Design
- Web Application Frameworks
- Separating Business Logic from Presentation
- Web-Tier State
- Distributable Web Applications

SUMMARY

REFERENCES AND RESOURCES

2. INTEGRATING SERVLETS AND JSP:
THE MODEL VIEW CONTROLLER (MVC) ARCHITECTURE


UNDERSTANDING THE NEED FOR MVC
- MVC Frameworks
- Architecture or Approach?

IMPLEMENTING MVC WITH REQUESTDISPATCHER
- Defining Beans to Represent the Data
- Writing Servlets to Handle Requests
- Populating the Beans
- Storing the Results
- Forwarding Requests to JSP Pages
- Extracting Data from Beans
SUMMARIZING MVC CODE
   Request-Based Data Sharing
   Session-Based Data Sharing 444
   Application-Based Data Sharing 444

INTERPRETING RELATIVE URLS IN THE DESTINATION PAGE

APPLYING MVC: BANK ACCOUNT BALANCES

COMPARING THE THREE DATA-SHARING APPROACHES
   Request-Based Sharing
   Session-Based Sharing
   Application-Based Sharing

FORWARDING REQUESTS FROM JSP PAGES

INCLUDING PAGES

3. A MODEL 2 FRAMEWORK
Excerpted from: Advanced JavaServer Pages Advanced JavaServer Pages by David Geary

A MODEL 2 FRAMEWORK
   The Action Interface
   The Action Factory
   Action Routers
   The Action Servlet
   Retrofitting the Original Model 2 Example

REFINING THE DESIGN

ADDING USE CASES
   Step #1: Implement a Password Hint Action
   Step #2: Implement a Password Hint JSP Page
   Step #3: Add Mappings to the Properties File
   Step #4: Modify the Login Failed JSP Page

THE IMPORTANCE OF CUSTOM TAGS

JSP SCRIPTS

CONCLUSION

4. A CASE STUDY
Excerpted from: Advanced JavaServer Pages Advanced JavaServer Pages by David Geary

THE FRUITSTAND
   The Homepage
   Going Shopping
   The Storefront
   The Shopping Cart
   The Checkout
The Purchase

THE MODEL 2 FRAMEWORK
The Model
The Views—JSP Pages and Templates
The Controllers—Servlets and Actions

INTERNATIONALIZATION

AUTHENTICATION

HTML FORMS

SENSITIVE FORM RESUBMISSIONS

SSL

XML AND DOM

CONCLUSION

5. AN OVERVIEW OF J2EE 1.4 WEB SERVICES
Excerpted from: J2EE Web Services by Richard Monson-Haefel

THE J2EE PLATFORM

THE TECHNOLOGIES OF WEB SERVICES
  WS-I Basic Profile 1.0
  XML
  SOAP
  WSDL
  UDDI

THE J2EE WEB SERVICE APIS
  JAX-RPC
  SAAJ
  JAXR
  JAXP

WRAPPING UP

6. SERVICE ENDPOINT DESIGN
Excerpted from: Designing Web Services with the J2EE 1.4 Platform: JAX-RPC, SOAP, and XML Technologies (Java Series) by Inderjeet Singh, Sean Brydon, Greg Murray, Vijay Ramachandran, Thierry Violleau, Beth Stearns

EXAMPLE SCENARIOS

FLOW OF A WEB SERVICE CALL
KEY WEB SERVICES DESIGN DECISIONS

DESIGNING A SERVICE’S INTERACTION LAYER
  Designing the Interface
  Receiving Requests
  Delegating Web Service Requests to Processing Layer
  Formulating Responses

PROCESSING LAYER DESIGN

PUBLISHING A WEB SERVICE
  Publishing a Service in a Registry
  Understanding Registry Concepts
  Registry Implementation Scenarios

HANDLING XML DOCUMENTS IN A WEB SERVICE
  Exchanging XML Documents
  Separating Document Manipulation from Processing Logic
  Fragmenting XML Documents
  Using XML
  Using JAXM and SAAJ Technologies

DEPLOYING AND PACKAGING A SERVICE ENDPOINT
  Service Information in the Deployment Descriptor

PACKAGE STRUCTURE

CONCLUSION

7. BUSINESS TIER DESIGN CONSIDERATIONS AND BAD PRACTICES
   Excerpted from: Core J2EE Patterns: Best Practices and Design Strategies, Second Edition by Deepak Alur, Dan Malks, John Crupi

BUSINESS TIER DESIGN CONSIDERATIONS
  Using Session Beans
  Using Entity Beans
  Caching Enterprise Bean Remote References and Handles

BUSINESS AND INTEGRATION TIERS BAD PRACTICES
  Mapping the Object Model Directly to the Entity Bean Model
  Mapping the Relational Model Directly to the Entity Bean Model
  Mapping Each Use Case to a Session Bean
  Exposing All Enterprise Bean Attributes via Getter/Setter Methods
  Embedding Service Lookup in Clients
  Using Entity Beans as Read-Only Objects
  Using Entity Beans as Fine-Grained Objects
  Storing Entire Entity Bean-Dependent Object Graph
  Exposing EJB-related Exceptions to Non-EJB Clients
  Using Entity Bean Finder Methods to Return a Large Results Set
  Client Aggregates Data from Business Components
  Using Enterprise Beans for Long-Lived Transactions
  Stateless Session Bean Reconstructs Conversational State for Each Invocation
8. JAVA APPLICATION SERVER PERFORMANCE
Excerpted from: *Performance Analysis for Java Websites* by Stacy Joines, Ruth Willenborg, Ken Hygh

WEB CONTENT TYPES

WEB APPLICATION BASICS
- The Model-View-Controller Design Pattern
- Servlets
- JavaServer Pages (JSPs)
- Assorted Application Server Tuning Tips

BEYOND THE BASICS
- HTTP Sessions
- Enterprise JavaBeans (EJBs)
- Database Connection Pool Management
- Web Services

OTHER FEATURES
- Built-in HTTP Servers
- General Object Pool Management
- Multiple Instances: Clones

SUMMARY

9. DEVELOPING A PERFORMANCE TEST PLAN
Excerpted from: *Performance Analysis for Java Websites* by Stacy Joines, Ruth Willenborg, Ken Hygh

TEST GOALS
- Peak Load
- Throughput Estimates
- Response Time Measurements

DEFINING THE TEST SCOPE
- Building the Test

SCALABILITY TESTING

BUILDING THE PERFORMANCE TEAM

SUMMARY

10. SCALABLE AND HIGH-PERFORMANCE WEB APPLICATIONS
Excerpted from: *Building Scalable and High-Performance Java Web Applications Using J2EE Technology* by Greg Barish

THE EMERGENCE OF WEB APPLICATIONS
- Basic Definitions
- The Nature of the Web and Its Challenges

PERFORMANCE AND SCALABILITY
- Performance
- Scalability
THE INTERNET MEDIUM
  Wide Audience
  Interactive
  Dynamic
  Always On
  Integrated
  Lack of Complete Control

MEASURING PERFORMANCE AND SCALABILITY
  Measuring Performance
  Measuring Scalability
  Throughput and Price/Performance

SCALABILITY AND PERFORMANCE HINTS
  Think End-to-End
  Scalability Doesn’t Equal Performance
  Measure Scalability by Comparison

SUMMARY

11. SCALABILITY AND PERFORMANCE TECHNIQUES
Excerpted from: Building Scalable and High-Performance Java Web Applications Using J2EE Technology by Greg Barish

CACHING AND REPLICATION

PARALLELISM

REDUNDANCY

ASYNCHRONY

RESOURCE POOLING

SUMMARY

12. CAPACITY PLANNING FOR THE WEBLOGIC SERVER
Excerpted from: J2EE Applications and BEA WebLogic Server (2nd Edition) by Angela Yochem, David Carlson, Tad Stephens

WEBLOGIC JROCKIT
  Compilation Options
  Garbage Collection
  JVM Management and Monitoring

ANALYSIS OF CAPACITY PLANNING
  Factors Affecting Capacity

METHODOLOGY AND METRICS FOR CAPACITY PLANNING
  Setting Capacity Goals
  Server Hardware Capacity Planning
  A Baseline Capacity Profile
  LAN Infrastructure Capacity Planning
  External Connectivity Capacity Planning
CAPACITY PLANNING BEST PRACTICES
  Be Conservative with Your Capacity-Planning Estimates
  Load-Test Your Application
  Optimize Your Application
  Plan for Growth
CHAPTER 1

THE WEB TIER

Excerpted from: Designing Enterprise Applications with the J2EE Platform by Inderjeet Singh, Beth Stearns, Mark Johnson, Enterprise Team
A J2EE application’s Web tier makes the application’s business logic available on the World Wide Web. The Web tier handles all of a J2EE application’s communication with Web clients, invoking business logic and transmitting data in response to incoming requests.

This chapter describes several ways of using Web-tier technology effectively in a J2EE application design, including examples from the sample application. The chapter is specifically not about Web page design.

4.1 The Purpose of the Web Tier

A server in the Web tier processes HTTP requests. In a J2EE application, the Web tier usually manages the interaction between Web clients and the application’s business logic. The Web tier typically produces HTML or XML content, though the Web tier can generate and serve any content type. While business logic is often implemented as enterprise beans, it may also be implemented entirely within the Web tier.

The Web tier typically performs the following functions in a J2EE application:

- **Web-enables business logic**—The Web tier manages interaction between Web clients and application business logic.
- **Generates dynamic content**—Web-tier components generate content dynamically, in entirely arbitrary data formats, including HTML, images, sound, and video.
• **Presents data and collects input**—Web-tier components translate HTTP PUT and GET actions into a form that the business logic understands and present results as Web content.

• **Controls screen flow**—The logic that determines which “screen” (that is, which page) to display next usually resides in the Web tier, because screen flow tends to be specific to client capabilities.

• **Maintains state**—The Web tier has a simple, flexible mechanism for accumulating data for transactions and for interaction context over the lifetime of a user session.

• **Supports multiple and future client types**—Extensible MIME types describe Web content, so a Web client can support any current and future type of downloadable content.

• **May implement business logic**—While many enterprise applications implement business logic in enterprise beans, Web-only, low- to medium-volume applications with simple transactional behavior can implement business logic entirely within the Web tier.

### 4.2 Web-Tier Technologies

This section presents a quick review of Web technologies in the J2EE platform, first describing legacy technologies, and then the Web-tier component types that supersede them. Feel free to skip this section if you are already familiar with these technologies. If you need to refresh your understanding beyond what this section offers, see the J2EE Tutorial (a reference to the J2EE Tutorial is listed in “References and Resources” on page 127).

#### 4.2.1 Traditional Web-Tier Technologies

Understanding the history of dynamic Web content generation provides a context for understanding the benefits of Web technology in the J2EE platform. The earliest versions of the World Wide Web relied on basic HTTP servers to serve static HTML pages to HTML browsers. However, it quickly became clear that dynamic content, generated on demand, would make the Web a platform for delivering applications as well as content.
Several mechanisms were developed to allow Web servers to generate content on demand, all of which can be thought of as Web server functional extensions. In this context, a Web application is simply a complex Web server extension.

Web-tier technologies in the J2EE platform provide a superset of the functionality offered by the older technologies described here. Easy migration from or seamless integration with legacy Web applications is one of the strengths of Web-tier technologies in the J2EE platform.

The earliest standard server extension mechanism was the Common Gateway Interface (CGI), which defines a type of stand-alone executable program used by a server to produce dynamic content. While CGI remains a popular option for Web applications, it has some important limitations. CGI has performance limitations, because each HTTP request to a CGI program usually results in the creation of a heavyweight process in the host operating system. CGI is also a simple interface that offers no portable support for high-level system services, such as load balancing, scalability, high availability, security, state maintenance, and resource management, making scalable CGI solutions difficult to develop and maintain. CGI’s simplicity is a double-edged sword: It is easy to understand, but it does not offer many portable system services to the developer.

Some of CGI’s limitation can be overcome with a server extension API, which allows developers to create libraries that generate dynamic content. Examples of such APIs include NSAPI (for Netscape servers), Apache extension modules (for Apache), and ISAPI (for Microsoft Internet Information Server). While extension libraries alleviate the overhead of CGI process creation, server extension APIs are nonportable between server vendors, locking applications into a particular vendor’s API and product line. Worse, server extension libraries can compromise system stability, because an extension library crash can take down the entire server.

An improvement to server extension APIs is server-side scripting, in which a script running inside the server produces dynamic content. Fast CGI is a server-side scripting interface that replaces an operating system CGI program with a server-side CGI script. Server-side scripts that fail usually do not crash the server, because the script interpreter can easily intercede to recover from script failures. Although server-side scripts may be somewhat more portable than extension APIs, they are non-portable to the extent that they use server-specific features. Server-side scripts also do not provide uniform, portable access to high-level system services.
4.2.2 Web-Tier Technologies in the J2EE Platform

Web-tier technologies in the J2EE platform provide the benefits of server-side scripting, using compiled Java classes in a standardized, secure, and vendor-neutral environment. This section briefly describes and provides best practices for Web-tier technologies in the J2EE platform.

A Web application is a collection of Web-tier components, content, and configuration information, which operates as a single functional unit. The runtime support environment for a Web application is called a Web container. A Web application archive (.war) file contains all of the class files and resources for the Web application, along with an XML deployment descriptor file that configures the application. See Chapter 7 in particular for more on packaging and deploying Web applications.

The platform specification defines a contract between the Web container and each Web component, defining the component’s lifecycle, the behavior the component must implement, and the services that the server must provide to the component.

The platform specification also defines two types of Web component technologies: Java Servlets (“servlets”) and JavaServer Pages™ (JSP™) technology.

A servlet is a Java class that extends a J2EE server, producing dynamic content in response to requests from the server. The server passes service requests to the servlet through the standard interface `javax.servlet`, which every servlet must implement.

A JSP page is an HTML page with special markup that provides customizable behavior for generating dynamic content at runtime. A JSP page is usually translated into a servlet when it is deployed. JSP technology provides a document-centric, rather than programmatic, way to specify dynamic content generation.

4.2.3 The Web Container

A J2EE Web application runs inside a J2EE server’s Web container. The container manages each component’s lifecycle, dispatches service requests to application components, and provides standard interfaces to context data such as session state and information about the current request.

The Web container provides a consistent interface to the components it hosts, so Web components are portable across application servers. And, because packaging and deployment of J2EE Web applications are standardized, a Web application
can be deployed into any J2EE server without recompiling the code or rebuilding the application archive.

The next few sections describe Web-tier components in the J2EE platform and explain the benefits their features provide.

### 4.2.4 Java Servlets

A Java Servlet is a Java class that extends a J2EE-compatible Web server. Each servlet class produces dynamic content in response to service requests to one or more URLs.

Servlets offer some important benefits over earlier dynamic content generation technologies. Servlets are compiled Java classes, so they are generally faster than CGI programs or server-side scripts. Servlets are safer than extension libraries, because the Java Virtual Machine (JVM) can recover from a servlet that exits unexpectedly. Servlets are portable both at the source-code level (because of the Java Servlet specification) and at the binary level (because of the innate portability of Java bytecode). Servlets also provide a richer set of standard services than any other widely adopted server extension technology.

In addition to producing content, servlets have several features that support application structure. A developer can create classes that respond to events in a servlet’s lifecycle by implementing listener interfaces. The sample application uses listener interfaces to initialize servlet data structures. A servlet can also be extended by one or more servlet filters, which are reusable classes that wrap calls to a servlet’s `service` method, transforming the request or the response. Servlet filters can be organized into filter chains that perform successive transformations on servlet requests or responses.

Distributed servlets are more scalable than non-distributed servlets. The Web container can provide an application with load balancing and failover by migrating user sessions among cluster nodes. Distributed servlets are marked `distributable` in the Web application deployment descriptor. They must follow a set of restrictions beyond those required of non-distributed servlets. The additional restrictions ensure that servlet code operates properly across session migrations.

For an introduction to or review of servlets, see the section entitled “Java Servlet Technology” in *The J2EE Tutorial*. 
4.2.5 JavaServer Pages (JSP) Technology

Most Web applications produce primarily dynamic HTML pages that, when served, change only in data values and not in basic structure. For example, all of the catalog pages in an online store may have identical structure and differ only in the items they display. JSP technology exists for producing such content.

A JSP page is a document containing fixed template text, plus special markup for including other text or executing embedded logic. The fixed template text is always served to the requester just as it appears in the page, like traditional HTML. The special markup can take one of three forms: directives, scripting elements, or custom tags (also known as “custom actions”). Directives are instructions that control the behavior of the JSP page compiler and therefore are evaluated at page compilation time. Scripting elements are blocks of Java code embedded in the JSP page between the delimiters `<%` and `%>`. Custom tags (discussed later in this section) are programmer-defined markup tags that generate dynamic content when the page is served. The JavaServer Pages specification defines a set of standard tags that are available in all platform implementations. Custom tags and scripting elements generate dynamic content that is included in a response when a page is being served.

JSP pages can specify dynamic content of any textual type, but they are primarily used for creating structured content such as HTML, XML, XHTML, and so on. JSP pages are easier to write than servlets, because they look like structured documents. JSP pages are a more natural development technology for page designers, who specialize in authoring structured documents. Although a JSP page looks to its author like a document, most J2EE implementations translate a JSP page into a servlet class when it is deployed. JSP pages are also compatible with a wide array of authoring tools that simplify page creation.

JSP pages differ from servlets in their programming model. A JSP page is primarily a document that specifies dynamic content, rather than a program that produces content. JSP page technology provides a “document-centric” alternative to “programmatic” servlets for creating dynamic, structured data.

4.2.5.1 XML JSP Page Syntax

The JSP specification defines an alternate XML syntax for JSP pages. Pages in standard JSP syntax cannot be well-formed XML because the markup does not conform to XML’s requirements. Pages using the alternate JSP XML syntax can be validated against an XML Schema Definition Language (XSDL) schema to check for many potential errors that would otherwise appear only at runtime. XML syntax can also
facilitate integration with development tools. For integrity, a single JSP file may not contain a mix of standard JSP syntax and XML syntax.

Writing JSP pages in XML syntax is different from using JSP pages to generate XML content. The XML JSP page syntax is a way to specify a JSP page using well-formed XML. JSP pages written in either standard or XML syntax are useful for generating dynamic XML content.

4.2.5.2 Custom Tags

JSP technology allows developers to define custom tags, which are markup tags that are replaced by dynamic content when the page is served. The dynamic content is created by a tag handler class, which a programmer creates and packages in a tag library archive file. A programmer defines the syntax for a tag and implements the tag’s behavior in the handler class. Page authors can then import and use tags in tag libraries just as they use other markup tags.

Custom tags provide several benefits to a J2EE application design.

• Custom tags are reusable, as scripting elements generally are not.
• Libraries of custom tags provide high-level services for JSP pages that are portable across JSP containers.
• Custom tags ease maintenance, because they reduce repeated code. Changing a tag’s handler class changes the tag’s behavior everywhere it is used.
• Custom tags help developers focus on their core skills. Page authors can work exclusively with custom tags and standard markup, instead of with a jumble of tags and cryptic scripting elements. Meanwhile, programmers can focus on developing custom tag logic.
• Custom tags can provide non-programmers, such as page authors, with an intuitive syntax for invoking business logic.
• Custom tags can decouple business logic and data presentation. This separation eases maintenance, clarifies the intent of each component, and allows programmers and page authors to work relatively independently of one another.

4.2.5.3 Standard Tag Libraries

Standard tag libraries are sets of custom tags that provide a basic set of domain-neutral functionality for JSP pages. Standard tags typically perform such functions
as Web resource inclusion, request forwarding, conditional logic, collection iteration, XSLT transformations, internationalization, state access, and HTML forms. Some companies have produced tag libraries that are intimately integrated with their tools and J2EE product lines. Other organizations have produced tag libraries for general use in J2EE applications. Apache Taglibs, for example, is an open-source project that contains dozens of custom tags.

The Java Standard Tag Library (JSTL) is now a part of the Java Community Process (JSR-52, A Standard Tag Library for JavaServer Pages). Once standardized, JSTL will provide a rich layer of portable functionality to JSP pages. It will be available in all compliant JSP containers. See the Apache Jakarta taglib page listed in “References and Resources” on page 127 for more on JSTL.

Standard tag libraries often provide much of the basic functionality that JSP pages need. Mature libraries have been tested and optimized by a community of developers. Adopting a high-quality standard tag library can save application development time.

4.2.6 Web-Tier Technology Guidelines

This section provides guidelines for effective use of servlets and JSP pages.

4.2.6.1 Where to Use Servlets

Servlets are most effectively used for implementing logic and generating binary content.

4.2.6.1.1 Use Servlets to Implement Services

Servlets are usually not visual components, except for some that generate binary content. Instead, think of a servlet as an information service provided by an application. A servlet can perform whatever service it provides—templating, security, personalization, application control—and then select a presentation component (often a JSP page) to which it forwards the request for display. The sample application implements its templating services as a servlet (see Section 4.4.3.1 on page 110). Just as a servlet can be thought of as a service, a servlet filter can be thought of as a customization or an extension of the services that a servlet provides.

4.2.6.1.2 Use Servlets as Controllers

Servlets are the preferred technology for implementing a Web-tier controller, which determines how to handle a request and chooses the next view to display. A control-
ler activates application operations and makes decisions, which are essentially procedural tasks that are best suited for program code in servlets.

JSP pages should not be used as controllers. Because JSP pages that are mostly logic are a mixture of markup tags and program code, they are difficult to read and maintain, especially for Web developers who are not programmers.

```java
public class extends HttpServlet {
    protected void doPost(HttpServletRequest req,
                           HttpServletResponse res) throws...
    {
        String creditCard = req.getParameter("creditCard");
        String jspPage = "/process" + creditCard + ".jsp";
        ServletContext sc = getServletContext();
        RequestDispatcher rd = getRequestDispatcher(jspPage);
        rd.forward(req, res);
    }
}
```

**Code Example 4.1**  A Servlet Properly Used as a Controller

Code Example 4.1 is an example of a servlet used properly as a controller. The same controller is implemented improperly as a JSP page in Code Example 4.5 on page 91. Comparing the two, the pure servlet implementation is cleaner and easier to maintain.

See also Section 4.2.6.9 on page 91.

### 4.2.6.1.3 Use Servlets to Generate Binary Content

Binary content should be generated by servlets. Servlets that output binary content must set the `Content-Type` HTTP header to the MIME type of the content being generated. A servlet writes its binary data to an `OutputStream` acquired from the `ServletRequest`, as shown in Code Example 4.2.

```java
public class JpgWriterServlet extends HttpServlet {
    public void service(HttpServletRequest req,
                         HttpServletResponse rsp) throws...
    {
        rsp.setHeader("Content-type", "image/jpg");
```
OutputStream os = rsp.getOutputStream();
// ... now write binary data to the OutputStream...

**Code Example 4.2**  A Servlet that Produces Binary Content

A servlet can write to either an OutputStream or a PrintWriter, but not both. JSP pages can’t create binary content.

**4.2.6.2 Avoid Writing Servlets That Print Mostly Static Text**

Servlets composed mostly of println statements would be better implemented as JSP pages. JSP pages are for creating textual content that combines template data with dynamic data values. Servlets that print a great deal of text, and perform some logic between the print lines, are tedious to write and difficult to maintain. Every delimiter in the quoted strings written by the servlet must be properly escaped with a backslash, reducing readability. Updating the visual presentation of such a servlet requires modifying and recompiling a program, instead of updating a page of markup.

```java
public class PopulateServlet extends HttpServlet {
    public void doGet(HttpServletRequest req,
            HttpServletResponse res) throws ... {
        ...
        if (dbConnectionClosed) {
            PrintWriter out = response.getWriter();
            out.println("<html>");
            out.println("<body bgcolor=white>");
            out.println("<font size=\"+5\" color=\"red\">Can't con-
                    nect</font>");
            out.println("<br>Confirm your database is running");
            out.println("</body></html>");
        }
    }
}
```

**Code Example 4.3**  Bad Practice: a Servlet that Prints Static Content
In Code Example 4.3, a servlet is used inappropriately to generate static content. The code is difficult to read, requires careful delimiter escaping, and would probably need a programmer for nontrivial modifications.

A better design, shown in Code Example 4.4, demonstrates a servlet that detects an error and forwards the request to a JSP page, which reports the error. This maintains proper separation of function from presentation, allowing Web developers and programmers to focus on their core skills.

PopulateServlet.java:
public class PopulateServlet extends HttpServlet {
    protected void doGet(HttpServletRequest req,
                          HttpServletResponse res) throws ... {
        ...
        if (dbConnectionClosed) {
            ServletContext ctx = getServletContext();
            ctx.getRequestDispatcher("/db_failed.jsp").forward(req, res);
        }
    }
}

db_failed.jsp:
<html>
<body>
    <br><font color="red">Unable to Connect</font>
    <br>Confirm that your database is running
</body>
<html>

Code Example 4.4  Servlet Logic that Delegates Display to a JSP Page

4.2.6.3 Use RequestDispatcher Methods forward and include Correctly

A servlet uses two RequestDispatcher methods, forward and include, to create a response using other components. However, the two methods are intended for fundamentally different operations. Use RequestDispatcher.forward to delegate processing of an entire request to another component, and use RequestDispatcher.include to build a response containing results from multiple Web resources.
When using the request dispatcher in a servlet, keep in mind that RequestDispatcher.forward requires that the body of the servlet response be empty. Writing something to the response and then calling forward either causes a runtime exception or discards any data previously written.

4.2.6.4 Where to Use JavaServer Pages

JSP pages are typically used for creating structured or free-form textual data. They are most appropriate where data values change between requests, but data structure either doesn’t change, or changes very little.

4.2.6.4.1 Use JSP Pages for Data Presentation

JSP pages are most appropriately used for producing structured textual content. Enterprise application data view types typically include HTML, XHTML, and DHTML.

JSP pages are best used for content that is partially fixed, with some elements that are filled in dynamically at runtime. A JSP page contains fixed content called “template data” (not to be confused with the templating mechanism described in this chapter). Custom tags or scripting elements occur at various points in the template data, and are replaced at runtime with dynamic data, producing customized content.

JSP pages cannot create binary content. They are also usually not appropriate for creating content with highly variable structure or for controlling request routing. Servlets are better for those situations. (See Section 4.2.6.1 on page 82 for more on this topic.) JSP pages can reasonably activate business logic, if the implementation of that logic is in a custom tag instead of in a scriptlet; see Section 4.2.6.8 on page 89.

4.2.6.4.2 Use JSP Pages to Generate XML

JSP pages are an excellent technology for generating XML with fixed structure. They are particularly useful for generating XML messages in standard formats, where message tags are fixed and only attribute values or character data change each time the page is served. XML documents can also be created from templates, assembling several XML subdocuments into a composite whole.
4.2.6.4.3 Use JSP Pages to Generate Unstructured Textual Content

JSP pages are not limited to producing documents with structured markup. They can also create unstructured textual content such as ASCII text, fixed-width or delimited data, and even PostScript. For example, JSP pages would be an excellent choice for rendering personalized e-mail form letters to customers.

4.2.6.4.4 Use JSP Pages as Templates

JSP pages may also be appropriately used for assembling textual data from multiple sources, as described in Section 4.4.3.1 on page 110.

4.2.6.5 JSP Pages Character Encoding

JSP pages use class `javax.servlet.jsp.JSPWriter` to write content to the response stream. This class automatically ensures that any text output by the JSP page is properly encoded. But automatic encoding is also a limitation, because it means JSP pages cannot be used to produce binary content directly.

4.2.6.6 Avoid Heavy Use of Logic Tags

Standard tag libraries usually provide so-called “logic tags,” which are custom tags that loop, perform iterations, evaluate expressions, and make decisions. Avoid using standard tag libraries to perform a great deal of logic in JSP pages. Using custom tags for logic provides little benefit, and violates separation of logic and presentation. JSP pages that are simply procedural programs written in XML syntax are at least as difficult to maintain as other types of programs.

Instead of creating “procedural” JSP pages, implement logic in a custom tag, servlet, or helper class. One powerful technique is to define custom tags for application logic, implementing the tag handlers using enterprise beans. A thin layer of code (the tag handler class) links the custom tag with enterprise bean lookup and method invocation. This approach provides the view (JSP page or servlet) with direct access to model data (in the enterprise beans), maintaining separation of presentation from function.

4.2.6.7 Use JSP Include Directives and Tags Appropriately

The JSP include directive and the JSP include tag have similar syntax but different purposes.
An include directive includes literal text “as is” in the JSP page and is not intended for use with content that changes at runtime. The include occurs only when the servlet implementing the JSP page is being built and compiled. For example, the following include directive includes a file at page compilation time:

```jsp
<%@ include file="header.jsp" %>
```

The JSP include tag includes either static or dynamic content in the JSP page when the page is being served. When used to include static content, the include tag works just the same as the include directive. But when an include tag includes dynamic content, the tag generates and includes the dynamic content in the context of the current request. The include occurs each time the JSP page is served. For example, the following JSP include tag includes dynamic content, which depends on the current request:

```jsp
<jsp:include page="/servlets/currentUserInfoServlet"/>
```

In contrast, the JSP include directive is commonly used to modularize Web pages, to reuse content, and to keep Web page size manageable. For example, an include directive could include headers and footers on every page. Using a JSP include directive results in a larger servlet and, if abused, could lead to code bloat.

The JSP include tag is commonly used to insert dynamically generated content into a JSP page at the time it is served. An include tag is more flexible than an include directive, because it can select the content to include at runtime. But an include tag requires runtime processing, and is therefore slower than an include directive.

Each time a particular JSP page is requested, all of its included pages are evaluated and recompiled if necessary. A child page must redeclare the components it uses (for example, JavaBeans components and objects provided by the container). Pages share data by way of the `HttpSession` object.

The sample application uses primarily include tags, because most of its JSP pages produce dynamic content.

The include directive is problematic for internationalization, because the `contentType:charset` of the included page cannot be set independently of the including page. The include tag is the only choice for including content with page encoding in JSP version 1.1.

*Implementation note:* Some Web container implementations (including Tomcat) do not automatically track modifications to files included by an include
directive. If you change a file that is included, you also need to force a recompile by “touching” (change the modification date of) the “parent” files that include the modified file.

4.2.6.8 Using Custom Tags to Avoid Scriptlets

Consider using custom tags instead of scriptlets in JSP pages for the following reasons:

- **Scriptlet code is not reusable**—Scriptlet code appears in exactly one place: the JSP page that defines it. If the same logic is needed elsewhere, it must be either included (decreasing readability) or copied and pasted into the new context.

  Custom tags can be reused by reference.

- **Scriptlets encourage copy/paste coding**—Because scriptlet code appears in only one place, it is often copied to a new context. When the scriptlet needs to be modified, usually all of the copies need updating. Finding all copies of the scriptlet and updating them is an error-prone maintenance headache. With time, the copies tend to diverge, making it difficult to determine which scriptlets are copies of others, further frustrating maintenance.

  Custom tags centralize code in one place. When a tag handler class changes, the tag’s behavior changes everywhere it is used.

- **Scriptlets mix logic with presentation**—Scriptlets are islands of program code in a sea of presentation code. Changing either requires some understanding of what the other is doing to avoid breaking the relationship between the two. Scriptlets can easily confuse the intent of a JSP page by expressing program logic within the presentation.

  Custom tags encapsulate program logic so that JSP pages can focus on presentation.

- **Scriptlets break developer role separation**—Because scriptlets mingle programming and Web content, Web page designers need to know either how to program or which parts of their pages to avoid modifying. Poorly implemented scriptlets can have subtle dependencies on the surrounding template data. Consider, for example, the following line of code:

  ```jsp
  <% out.println("<a "href="" + url + "><" + text); %> </a>
  ```
It would be very easy to change this line in a way that breaks the page, especially for someone who does not understand what the line is doing.

Custom tags help the separation of developer roles, because programmers create the tags, and page authors use them.

- **Scriptlets make JSP pages difficult to read and to maintain**—JSP pages with scriptlets mix structured tags with JSP page delimiters and Java language code. The Java language code in scriptlets often uses “implicit” objects, which are not declared anywhere except in the JavaServer Pages specification. Also, even consistent indentation does not help readability much for nontrivial pages.

  JSP pages with custom tags are composed of tags and character data, which is much easier to read. JSP pages that use XML syntax can be validated as well.

- **Scriptlet compile errors can be difficult to interpret**—Many JSP page compilers do a poor job of translating line numbers between the source page and the generated servlet. Even those that emit error messages often depend on invisible context, such as implicit objects or surrounding template data. With poor error reporting, a missed semicolon can cost hours of development time.

  Erroneous code in custom tags will not compile either, but all of the context for determining the problem is present in the custom tag code, and the line numbers do not need translation.

- **Scriptlet code is difficult to test**—Unit testing of scriptlet code is virtually impossible. Because scriptlets are embedded in JSP pages, the only way to execute them is to execute the page and test the results.

  A custom tag can be unit tested, and errors can be isolated to either the tag or the page in which it is used.

  Expressions sometimes also suffer from these problems, but they are somewhat less problematic than scriptlets because they tend to be small.

  Custom tags maintain separation between developer roles. They encourage reuse of logic and state within a single source file. They also improve source code readability, and improve both testability and error reporting.

  Some projects have Web page authors but few or no programmers. Page authors with limited programming skill can use scriptlets effectively if they use scriptlets for display logic only. Business logic should never be implemented in scriptlets.
4.2.6.9 Avoid Forwarding Requests from JSP Pages

When a JSP page calls RequestDispatcher.forward, either directly or with a custom tag, it is acting as a controller. Controllers are better implemented as servlets than as JSP pages, because controllers are logical components, not presentation components. Code Example 4.5 demonstrates a controller implemented as a JSP page.

```jsp
<% String creditCard = request.getParameter("creditCard");
   if (creditCard.equals("Visa")) {
       <jsp:forward page="/processVisa.jsp"/>
   } else if (creditCard.equals("American Express")) {
       <jsp:forward page="/processAmex.jsp"/>
   } %>
```

**Code Example 4.5**  Bad Practice: JSP Page Acting as a Controller

In this example, the scriptlets and tags conditionally forward the request to another JSP page based on a request attribute. If each of the forward tags were replaced with a single line of Java code that performed the forward, the result would be a JSP page containing nothing but a scriptlet. This code would obviously be better implemented as a servlet. Code Example 4.1 on page 83 shows the code for a servlet implementation of this functionality.

4.3 Web-Tier Application Structure

The J2EE platform is a layered set of system services that are consistently available to J2EE applications across implementations. It is the top layer of a “stack” of services that support an application, shown in Figure 4.1. The J2EE platform runs on top of the J2SE platform, which itself runs on top of the host operating system. In
the Web tier, a J2EE Web container provides services related to serving Web requests.

![Figure 4.1 Platform and Application Layers](image)

Just as the J2EE platform has layers, J2EE applications can benefit from architectural layering. The highest-level division between layers in an application’s Web tier is between functions that are specific to a particular application and those that occur in all Web applications.

All Web-tier applications share a common set of basic requirements that are not provided by the J2EE platform itself. A software layer called an application framework can meet these requirements and can be shared between applications. As shown in Figure 4.1, application-specific code is written in terms of the application framework layer.

A Web-tier application framework sits on top of the J2EE platform, providing common application functionality such as dispatching requests, invoking model methods, and selecting and assembling views. Framework classes and interfaces are structural; they are like the load-bearing elements of a building, forming the application’s underpinnings. Application developers extend, use, or implement framework classes and interfaces to perform application-specific functions. For example, a framework may offer an abstract class that a developer may extend to execute business logic in response to application events. A Web-tier application framework makes Web-tier technologies easier to use, helping application developers to concentrate on business logic.
The BluePrints recommended best practice is to choose an existing, proven Web-tier application framework for a J2EE application rather than designing and building a custom framework layer. A Web-tier application framework can provide the following benefits to your application:

- **Decouples presentation and logic into separate components**—Frameworks encourage separating presentation and logic because the separation is designed into the extension interfaces.

- **Separates developer roles**—Application frameworks generally provide different interfaces for different developers. Presentation component developers tend to focus on creating JSP pages using custom tags, while logic developers tend to write action classes, tag handlers, and model code. This separation allows both types of developers to work more independently.

- **Provides a central point of control**—Most frameworks provide a rich, customizable set of application-wide features, such as templating, localization, access control, and logging.

- **Facilitates unit testing and maintenance**—Because framework interfaces are consistent, automated testing harnesses are easy to build and execute.

- **Can be purchased instead of built**—Time not spent developing structural code is available for developing business logic.

- **Provides a rich set of features**—Adopting a framework can leverage the expertise of a group of Web-tier MVC design experts. The framework may include useful features that you do not have the experience to formulate or the time to develop.

- **Encourages the development and use of standardized components**—Over time, developers and organizations can accumulate and share a toolbox of preferred components. Most frameworks incorporate a set of custom tags for view construction.

- **Provides stability**—Frameworks are usually created and actively maintained by large organizations or groups, and are used and tested in a large installed base. Accordingly, framework code tends to be more stable than custom code.

- **Has community support**—Popular frameworks attract communities of enthusiastic users who report bugs, provide consulting and training services, publish
tutorials, and produce useful add-ons. Open frameworks are particularly strong in this regard.

- **May reduce training costs and time**—Developers already trained and experienced in using a framework get up to speed more quickly and are more productive.

- **May simplify internationalization**—Most frameworks support a flexible internationalization strategy.

- **May support input validation**—Many frameworks have consistent ways to specify input validation. Validation is commonly available on the client side, on the server side, or both.

- **May be compatible with tools**—Good tools can improve productivity and reliability. Some frameworks are integrated with rapid application development tool sets.

All of these benefits come at a cost, of course. You always have less control over a design you have acquired rather than created yourself. Some frameworks must be purchased, although these are usually bundled with a tool set. Other people’s code in your application means other people’s bugs in your application. Still, most development projects find that a Web-tier framework improves design and implementation quality.

### 4.4 Web-Tier Application Framework Design

Model-View-Controller (“MVC”) is the BluePrints recommended architectural design pattern for interactive applications. MVC, described in Chapter 11, organizes an interactive application into three separate modules: one for the application model with its data representation and business logic, the second for views that provide data presentation and user input, and the third for a controller to dispatch requests and control flow. Most Web-tier application frameworks use some variation of the MVC design pattern.

The MVC design pattern provides a host of design benefits. MVC separates design concerns (data persistence and behavior, presentation, and control), decreasing code duplication, centralizing control, and making the application more easily modifiable. MVC also helps developers with different skill sets to focus on their core skills and collaborate through clearly defined interfaces. For example, a J2EE application project may include developers of custom tags, views, applica-
tion logic, database functionality, and networking. An MVC design can centralize control of such application facilities as security, logging, and screen flow. New data sources are easy to add to an MVC application by creating code that adapts the new data source to the view API. Similarly, new client types are easy to add by adapting the new client type to operate as an MVC view. MVC clearly defines the responsibilities of participating classes, making bugs easier to track down and eliminate.

This section describes how to use MVC to organize a J2EE Web application design using the sample application’s Web Application Framework design as an example. Many of the key classes described (the controller, the templating service, the abstract action class, and so on) are usable for any application, not just for an online shopping application.

A J2EE application’s Web tier serves HTTP requests. At the highest level, the Web tier does four basic things in a specific order: interprets client requests, dispatches those requests to business logic, selects the next view for display, and generates and delivers the next view. (See Figure 4.2.)

The Web-tier controller receives each incoming HTTP request and invokes the requested business logic operation in the application model. Based on the results of the operation and state of the model, the controller then selects the next view to display. Finally, the controller generates the selected view and transmits it to the client for presentation.

Figure 4.2 is deceptively simple. An enterprise application’s Web tier commonly has the following requirements:
• An application design must have a strategy for serving current and future client types.

• A Web-tier controller must be maintainable and extensible. Its tasks include mapping requests to application model operations, selecting and assembling views, and managing screen flow. Good structure can minimize code complexity.

• Application model API design and technology selection have important implications for an application’s complexity, scalability, and software quality.

• Choosing an appropriate technology for generating dynamic content improves development and maintenance efficiency.

The BluePrints best practice is to implement the Web tier of a J2EE enterprise application using an appropriate Web application framework. (See Section 4.4.5 on page 114.) The next several sections describe the general design of a J2EE application Web tier. If you choose to use a Web application framework, the following discussion will help you to understand what the framework does and how to use it. If you write your own Web-tier architectural code, the following design discussions will help you make educated decisions about how to use the technology.

4.4.1 Structuring the Web Tier

Overall structure is the most important consideration in a Web-tier design. Both the sample application and the various existing Web application frameworks implement some form of “Model 2” architecture, where a servlet manages client communication and business logic execution, and presentation resides mainly in JSP pages.

The literature on Web-tier technology in the J2EE platform frequently uses the terms “Model 1” and “Model 2” without explanation. This terminology stems from early drafts of the JSP specification, which described two basic usage patterns for JSP pages. While the terms have disappeared from the specification document, they remain in common use. Model 1 and Model 2 simply refer to the absence or presence (respectively) of a controller servlet that dispatches requests from the client tier and selects views.

A Model 1 architecture consists of a Web browser directly accessing Web-tier JSP pages. The JSP pages access Web-tier JavaBeans that represent the application model, and the next view to display (JSP page, servlet, HTML page, and so on) is determined either by hyperlinks selected in the source document or by
request parameters. A Model 1 application control is decentralized, because the current page being displayed determines the next page to display. In addition, each JSP page or servlet processes its own inputs (parameters from GET or POST). In some Model 1 architectures, choosing the next page to display occurs in scriptlet code, but this usage is considered poor form. (See the design guideline Section 4.2.6.8 on page 89.)

A Model 2 architecture introduces a controller servlet between the browser and the JSP pages or servlet content being delivered. The controller centralizes the logic for dispatching requests to the next view based on the request URL, input parameters, and application state. The controller also handles view selection, which decouples JSP pages and servlets from one another. Model 2 applications are easier to maintain and extend, because views do not refer to each other directly. The Model 2 controller servlet provides a single point of control for security and logging, and often encapsulates incoming data into a form usable by the back-end MVC model. For these reasons, the Model 2 architecture is recommended for most interactive applications.

An MVC application framework can greatly simplify implementing a Model 2 application. Application frameworks such as Apache Struts and JavaServer Faces™ (see Section 4.4.5 on page 114) include a configurable front controller servlet, and provide abstract classes that can be extended to handle request dispatches. Some frameworks include macro languages or other tools that simplify application construction.

The Model 1 architecture can provide a more lightweight design for small, static applications. Model 1 architecture is suitable for applications that have very simple page flow, have little need for centralized security control or logging, and change little over time. Model 1 applications can often be refactored to Model 2 when application requirements change.

4.4.1.0.1 When to Switch from Model 1 to Model 2

JSP pages in a Model 1 application that use scripting elements, custom tags, or JavaScript to forward requests should be refactored to Model 2.

A Model 1 architecture is best when the page navigation is simple and fixed, and when a simple directory structure can represent the structure of the pages in the application. Such applications usually embed the page flow information in the links between the pages. The presence of forward in a JSP page implies that logic embedded in the page is making a decision about the next page to display.

Over time, as the application grows and changes, page flow logic accumulates. The application becomes difficult to maintain because the page flow logic is
distributed across multiple pages. The best time to switch from Model 1 to Model 2 is before this maintenance problem arises. This is why it’s usually best to choose Model 2 from the outset, basing the application on an existing Web controller framework that best meets application requirements. Model 1 remains a viable option for simple, static applications.

4.4.2 Web-Tier MVC Controller Design
The Model 2 architecture uses servlets for processing requests and selecting views. The Front Controller architectural design pattern centralizes an application’s request processing and view selection in a single component. Each type of Web client sends requests to and receives responses from a single URL, simplifying client development. The Front Controller receives requests from the client and dispatches them to the application model. This single point of dispatch makes the Front Controller a logical place for such global facilities as security and logging. The Front Controller also selects and formats the next client view. The controller is also an application of the Mediator pattern, because it decouples view components from one another.

In the J2EE platform, a Front Controller is typically implemented as a servlet. The sample application’s Front Controller servlet handles all HTTP requests. The user views, discussed in the next section, are mostly JSP pages chosen by the Front Controller.

4.4.2.1 Web-Tier Controller Design
A Web-tier MVC controller maps incoming requests to operations on the application model, and selects views based on model and session state. Web-tier controllers have a lot of duties, so they require careful design to manage complexity. Because most enterprise applications grow over time, extensibility is an important requirement. This section describes some strategies for the internal structure of a controller in the Web tier, illustrated by example code adapted from the Web Application Framework, part of the BluePrints sample application.

4.4.2.1.1 Identifying the Operation to Perform
When a controller receives an HTTP request, it needs to be able to distinguish what application operation is being requested. How can the client, for example, request
that the server create a new user? There are several ways to indicate to the server which operation to perform. The more common methods include:

- Indicate the operation in a hidden form field, which a POST operation delivers to the controller; for example:

  ```html
  <FORM METHOD="POST" ACTION="http://myServer/myApp/myServlet">
    <INPUT TYPE="HIDDEN" NAME="OP" VALUE="createUser"/>
    <!-- other form contents... -->
  </FORM>
  ```

- Indicate the operation in a HTTP GET query string parameter; for example:

  ```
  http://myHost/myApp/servlets/myServlet?op=createUser
  ```

- Use a servlet mapping to map all URLs with a particular suffix or base URL to a specific servlet. A servlet mapping is a deployment descriptor definition that compares request paths to a pattern and dispatches matching requests to the corresponding servlet. For example, imagine that a Web application’s deployment descriptor defines the following servlet mapping:

  ```
  <servlet-mapping>
    <servlet-name>myServlet</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>
  ```

  Imagine also that the servlet’s context path is http://myServer/myApp/servlets. The servlet container would direct a request with URL http://myServer/myApp/createUser.do to myServlet, because the request URL matches the pattern *.do. Servlet myServlet can extract the requested operation’s name from the request URL. Chapter 11 of the Java Servlet 2.3 specification defines servlet mappings.

  Of the three options discussed here, the BluePrints recommendation is to use servlet mappings when they are available. Servlet mappings provide the most flexible way to control where to route URLs based on patterns in the URLs. Most Web application frameworks (see Section 4.4.5 on page 114) use servlet mappings to direct requests to the appropriate front controller for an application.

  The sample application uses a servlet mapping to handle request URLs. The servlet container maps all request URLs matching *.do to the main Web-tier con-
troller servlet, MainServlet.java. Another servlet mapping routes all URLs matching *.screen to the templating service, which assembles composite views.

4.4.2.1.2 Invoking Model Methods

Once the controller has determined which operation to perform, it must invoke the corresponding application model method with parameters derived from the request. A naive controller design might use a large if-then-else statement, as shown in Code Example 4.6.

```java
if (op.equals("createUser")) {
    model.createUser(request.getAttribute("user"),
                     request.getAttribute("pass"));
} else if (op.equals("changeUserInfo") {  
    // ... and so on...
}
```

**Code Example 4.6**  A Poorly Designed Controller

The if-then-else approach leads to a very large service method, which is difficult to read and still more difficult to maintain. A better approach is to use the Command pattern. The sample application defines an abstract class Action, which represents a single application model operation. A controller can look up concrete Action subclasses by name and delegate requests to them. Sample code for the abstract class Action and a concrete class CreateUserAction appears in Code Example 4.7.

```java
// Action.java:
public abstract class Action {
    protected Model model;
    public Action(Model model) { this.model = model; }
    public abstract String getName();
    public abstract Object perform(HttpServletRequest req);
};

// CreateUserAction.java:
public class CreateUserAction extends Action {
    public CreateUserAction(Model model) {
        super(model);
    }
```
public String getName() { return "createUser"; }
public Object perform(HttpServletRequest req) {
    return model.createUser(req.getAttribute("user"),
        req.getAttribute("pass"));
}

Code Example 4.7  An Abstract Action Class and a Concrete Subclass

Code Example 4.7 defines an abstract class Action, which has a name and a perform method that executes a model method corresponding to the name. For example, Action’s concrete subclass CreateUserAction has the name “createUser”. Its perform method invokes the model method createUser using parameters extracted from the HTTP request.

```java
public class ControllerServlet extends HttpServlet {
    private HashMap actions;
    public void init() throws ServletException {
        actions = new HashMap();
        CreateUserAction cua = new CreateUserAction(model);
        actions.put(cua.getName(), cua);
        //... create and add more actions
    }
    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws IOException, ServletException {
        // First identify operation "op" from URL.
        // method getOperation() is defined elsewhere.
        String op = getOperation(req.getRequestURL());
        // Then find and execute corresponding Action
        Action action = (Action)actions.get(op);
        Object result = null;
        try {
            result = action.perform(req);
        } catch (NullPointerException npx) {
            //... handle error condition: no such action
```
Code Example 4.8 Using a Map to Identify and Execute Actions

Code Example 4.8 shows a controller servlet that maintains a hash map of Action objects, each indexed by its name. When the servlet loads, the servlet container calls the method init, which fills the hash map with Action objects that invoke model operations. The hash map key is the name of the operation. Each time the servlet's service method receives a request, it identifies the name of the operation to perform, looks up the corresponding Action in the hash map, and executes it by invoking the Action's perform method. The Action returns a result object that the servlet uses, along with other data, to decide which view to display next. When this controller receives a request containing the name createUser, it finds an instance of CreateUserAction in the hash map. It then invokes the Action's perform method, which uses the model to create a user (as shown in Code Example 4.7).

The code samples shown in this section are greatly simplified for clarity. The Web Application Framework used by the sample application provides a full, working example of this sort of controller called MainServlet. The servlet container dispatches requests with a servlet mapping: it forwards all URLs matching *.do to the MainServlet. Code Example 4.8 demonstrates how to provide an extensible framework for dispatching client requests.

The sample application improves the extensibility of the servlet code in Code Example 4.8 even further by externalizing the map of requests to actions. The controller in the sample application initializes the actions hash map from an external XML file, which contains pairs of operation names and corresponding Action class names. The controller servlet initializes the action map with the request names and action classes referred to in the XML file. The XML file is deployed as a resource in the Web application archive. Adding a new Action is as simple as adding a concrete Action subclass to the application archive and defining a configuration file mapping that associates the request URL with the action class. An example of such a mapping appears in Code Example 4.9. With no code
modification, the sample application controller servlet can dispatch requests using actions that did not even exist when the controller was written.

Dispatching service requests to the application model is only half of the Web-tier controller’s job. It is also responsible for determining the next view to display.

### 4.4.2.1.3 Controlling Dynamic Screen Flow

The succession of views that a Web application user sees is called screen flow. A Web-tier controller controls screen flow by selecting the next view a user sees. In static Web sites, screens (usually Web pages) are statically linked to one another. By contrast, a controller dynamically chooses the “next” screen in response to both user actions and model operation results.

In this section, the term “view” means a Web resource with a URL from which Web content is available. A view might be a JSP page, a servlet, static content, or some combination of the three, assembled into a page. Typically, the “next” view to display depends on one or more of:

- The current view
- The results of any operation on the application model, returned by model method invocations
- Possibly other server-side state, kept in PageContext, ServletRequest, HttpSession, and ServletContext.

For example, the next view to display after a sign on view very likely depends on:

- The current view
- The user id and password contained in the request
- The success or failure of the sign on operation, and
- Possibly other server-side state. Examples of such state might include a maximum number of allowed users (application scope), or the URL the user was trying to access (request scope). See Section 4.4.7 on page 116 for a description of state and its scope.

The controller uses this data to determine which view to display next. A Web controller “displays a view” by forwarding the request to a JSP page, servlet, or
other component that renders the view in a format compatible with the client; for example, returning HTML to a browser.

The controller in the sample application uses two components to select and generate views: a screen flow manager, which selects the next view to display; and a templating service, which actually generates the view content. The controller uses the screen flow manager to select a view, and forwards the request to the templating service, which assembles and delivers a view to the client. Both the screen flow manager and the templating servlet are generic components that are usable in any Web application. The component-based design reduces component coupling, promoting code reuse and simplifying the controller design.

![Figure 4.3 Web-Tier Controller OID](image)

**Figure 4.3** Web-Tier Controller OID

Figure 4.3 is an object interaction diagram that shows the Web-tier controller interacting with other Web-tier classes. The diagram shows the following sequence of calls:

1. The controller receives a POST from the client.
2. The controller creates an *Action* corresponding to the requested operation (as described in the previous section).
3. The controller calls the *Action*’s `perform` method.
4. `perform` calls a model business method.
5. The controller calls the screen flow manager to select the next view to display.
6. The screen flow manager determines the next view and returns its name to the controller.

7. The controller forwards the request to the templating service, which assembles and delivers the selected view to the client.

Most request URLs map to a specific view URL. For example, the screen flow map can define that the view signoff.screen always follows request URL /signoff.do. Sometimes the next screen to display depends on model operation results, server-side state, or user activity. For example, the next view following the request URL /signin.do, which signs a user into the system, depends on whether the sign in operation succeeded.

In the sample application, an application assembler configures the screen flow manager with an XML-based file called a screen flow map. The screen flow map defines a next view name for each request URL. For dynamic view selection, a screen flow map can also map a request URL to a flow handler, which is a Java class that selects the next view programmatically. Flow handlers are typically written by component providers or application assemblers.

4.4.2.1.4 Example

The Web Application Framework screen flow map mappings.xml configures the screen flow manager. Sample application Code Example 4.9 shows a URL action mapping that uses a flow handler to determine the next view in code.

```xml
<url-mapping url="signoff.do" screen="signoff.screen">
  <action-class>
    com.sun.j2ee.blueprints.petstore.controller.web.actions.SignOffHTMLAction
  </action-class>
</url-mapping>
```

**Code Example 4.9**  Excerpt from the Sample Application Screen Flow Map

In Code Example 4.9, the url-mapping element defines a mapping for request URL /signoff.do. The action element declares an action of class SignoffHTMLAction, which performs the business logic for this URL (signing off a user). An application assembler or a component provider wrote the action class SignoffHTMLAction to sign a user off of the application. The screen attribute tells
the screen flow manager to display screen signoff.screen after the action completes.

Figure 4.4 shows the result of an HTTP POST to the URL /signoff.do.

![Sequence diagram](image)

**Figure 4.4** OID of POST to Flow Handler Defined in Code Example 4.9

The servlet container deployment descriptor has a servlet mapping from pattern *.do to the controller, so when a client POSTs a request to /verifysignin.do, the following actions occur:

1. The servlet container routes the request to the controller.
2. The controller creates an instance class SignoffHTMLAction and passes the request to it.
3. The controller calls the SignoffHTMLAction’s perform method.
4. The SignoffHTMLAction object calls the model method that signs the user out of the application.
5. The controller asks the screen flow manager for the next view.
6. The controller forwards the request to the URL signoff.screen.
7. The templating servlet generates and delivers the requested view (a templated JSP page) to the client, so the user receives a view indicating that signoff succeeded.

The last piece of the puzzle not yet explained is how to map a view name in the design just described to an actual Web component (JSP page, servlet, and so on). Views and templating are discussed in Section 4.4.3 on page 110.

4.4.2.2 Serving Multiple Client Types

Web applications may support only one type of client with a single protocol, or multiple clients with different protocols, security policies, presentation logic, and workflows. Web clients may include several versions of a few different browsers, MIDP clients, so-called “rich” clients with stand-alone APIs, and Web service interfaces. Long-lived applications may need to be able to handle new types of Web clients.

Each type of client needs its own controller, which specializes in the protocols for that client type. A particular type of client may also need different presentation components for form factor or other reasons.

Following are some options for how to service requests from clients that use different application-level protocols. (Web-tier clients use HTTP for transport.) Each of the following alternatives expands upon Figure 4.2 by adding flexibility and increasing complexity.

![Figure 4.5 Using a Front Controller to Handle Browser Interaction](image)

Applications with a single client type can implement a single front controller. For example, a browser-only application is shown in Figure 4.5. Its single Front Controller servlet receives HTTP requests from the browser, translates the contents of these requests into operations on the application model, and serves views of result data as HTML (or XML). Additional controllers can support new client types, as shown in Figure 4.6.
The multiple-controller approach in Figure 4.6 provides extensibility for any future Web client types, including those that do not yet exist. In fact, because servlets aren’t limited to HTTP, this architecture can support even non-Web clients. Each controller can implement the workflow, presentation logic, and security constraints unique to its client type. Notice also that the code implementing the application model is shared by all of the controllers. This separation of model and controller ensures identical application behavior across client types and eases maintenance and testing.

Some application functionality, particularly security, can be easier to manage from a single point. Introducing a protocol router, as shown in Figure 4.7, can provide a single point of control for all Web clients, each of which still retain their own controllers.
The protocol router in Figure 4.7 is either a servlet or servlet filter that determines the client type and dispatches the request to the appropriate controller. The router typically uses the HTTP header `User-Agent` to determine what sort of client is requesting service. The protocol router can implement application-wide functionality such as security or logging. The client-specific controllers can implement behavior specific to each client’s particular protocol.

The Front Controllers in Figure 4.7 may or may not be servlets. If the Front Controllers are servlets, the protocol router dispatches requests to them using `RequestDispatcher.forward`. If the protocol router is a servlet, the Front Controllers can be a layer of simple objects to which the router delegates request processing.

Note that the controller alternatives shown in the last few figures can be implemented incrementally. Each of the approaches can be built on the preceding one. The BluePrints recommendation is to adopt and adapt the alternative that most closely matches current application requirements, and add new functionality as necessary.

Templating is an application of the Composite View design pattern, discussed in Chapter 11, which builds a page from a number of independent view components.
4.4.3 Web-Tier MVC View Design

MVC views display data produced by the MVC model. View components (also known as “presentation components”) in the Web tier are usually JSP pages and servlets, along with such static resources as HTML pages, PDF files, graphics, and so on. JSP pages are best used for generating text-based content, often HTML or XML. Servlets are most appropriate for generating binary content or content with variable structure. (For an in-depth explanation of appropriate technology usage, see Section 4.2.6.1 on page 82 and Section 4.2.6.4 on page 86.)

HTML browsers are very lightweight clients, so the Web tier generates and often styles dynamic content for browsers. Heavyweight clients can implement relatively more view functionality in the Client tier, and less in the Web tier. Such clients include stand-alone rich clients, application clients, and clients that use special content formats such as MacroMedia Flash or Adobe Portable Document Format (PDF).

Web-tier components are not limited to serving HTML-over-HTTP Web browsers. The Web tier may also serve MIDP clients using proprietary protocols, rich clients using XML, or Web service peers requesting services with Electronic Business XML (ebXML) or Simple Object Access Protocol (SOAP) messages. Each of these examples uses a different application-level protocol, while using HTTP for transport. A properly designed Web tier unifies access to application functionality for any client type. The Web tier also provides virtual session management for some client types.

See Chapter 3 for more on J2EE client technologies.

4.4.3.1 Templating

One typical application requirement is that application views have a common layout. A template is a presentation component that composes separate subviews into a page with a specific layout. Each subview, such as a banner, a navigation bar, or document body content, is a separate component. Views that share a template have the same layout, because the template controls the layout.
For example, Figure 4.8 shows the layout of a single page created by a template. Across the top of the page is a banner, on the left is a navigation menu, a footer appears at the bottom, and the body content occupies the remaining space.

![Diagram showing the layout of a page with banner, navigation menu, body, and footer]

**Figure 4.8**  A Template Composes Other Views into a Consistent Layout

Using templates in an application design centralizes control of the overall layout of pages in the application, easing maintenance. Changing the layout in the template file changes the page layout for the entire application. More importantly, the individual subviews (like the “Navigation Menu” in Figure 4.8) are used by reference in the template instead of by copy-and-paste. Therefore, changing a subview means changing a single source file instead of changing all the files in which that subview occurs.

Template implementation is most easily explained by example. In the sample application, a JSP page called a template file specifies the page layout. The template file is a standard JSP page that uses custom tags to include subviews into each area of the page. The template references the individual subviews by name.

Code Example 4.10 is an example from the sample application that produces the layout shown in Figure 4.8. This file, called `template.jsp`, is a JSP page that produces HTML. The file specifies the page layout as standard HTML tags, and
includes the content of other JSP pages using the custom tag `insert`, shown underlined in the code example.

```html
<%@ taglib uri="/WEB-INF/tlds/taglib.tld" prefix="template" %>
<html>
<head>
    <title><template:insert parameter="title" /></title>
</head>
<body bgcolor="#FFFFFF">
<table width="100%" border="0" cellpadding="5" cellspacing="0">
<tr>
    <td colspan="2">
        <template:insert parameter="banner" />
    </td>
</tr>
<tr>
    <td width="20%" valign="top">
        <template:insert parameter="sidebar" />
    </td>
    <!--- ... and so on ... --->
</table>
</body>
</html>
```

**Code Example 4.10** The Template JSP Page for the Layout Shown in Figure 4.8

The JSP page includes the page named by the `insert` tag's parameter attribute at the point where the tag occurs. A separate screen definitions file for the application provides values for these parameters for each screen name.

The templating service is a single servlet that processes all screens. A servlet mapping routes all requests with URLs matching `*.screen` to a TemplateServlet, which assembles and serves the requested screen. Code Example 4.11 shows the definition of the screen called `main.screen`. The screen definitions file defines `template.jsp` as its template file and defines a series of screens. Each screen has a name and a list of values for the parameters in the template file. The templating service replaces each `insert` tag in the template file with the contents of the subview named by the tag's parameter attribute. For example, the templating service replaces all instances of `<template:insert parameter="banner"/>` with the contents of "/banner.jsp". The result is a fully-rendered screen.
Code Example 4.11 Screen Definition of Sample Application’s “Main” View

The templating service described here is part of the sample application’s Web Application Framework. The templating service is reusable as a component in other applications. Its design is based on the Composite View design pattern, which assembles a view from reusable subviews. For more information on the Composite View design pattern, please see Chapter 11.

4.4.4 Web-Tier MVC Model Design

An MVC application model both represents business data and implements business logic. Many J2EE applications implement their application models as enterprise beans, which offer scalability, concurrency, load balancing, automatic resource management, and other benefits. Simpler J2EE applications may implement the model as a collection of Web-tier JavaBeans components used directly by JSP pages or servlets. JavaBeans components provide quick access to data, while enterprise beans provide access to shared business logic and data.

Notice that the “application model” in Figure 4.5 on page 107 is generic: It implies no particular technology or tier. The application model is simply the programmatic interface to the application’s business logic. Model API design and model technology selection are both important design considerations.
Section 11.4.1.2 on page 369 describes MVC model design considerations and patterns.

4.4.5 Web Application Frameworks

As the Model 2 architecture has become more popular, quite a number of Web-tier application frameworks have appeared. Some are vendor-specific frameworks integrated with specific servers and tools; others are freely available, open-source projects. Benefits of Web-tier application frameworks appear on page 93. Three frameworks of particular interest are:

- **J2EE BluePrints Web Application Framework ("WAF")**—The Web Application Framework forms the infrastructure of the sample application. This framework offers a Front Controller servlet, an abstract action class for Web-tier actions, a templating service, several generic custom tags, and internationalization support. WAF demonstrates both the mechanisms and effective use of a Web-tier framework layer in an application design. It is suitable for small, non-critical applications, and for learning the principles of Web-tier application framework design and usage.

- **Apache Struts**—Struts is a free, open-source, Web-tier application framework under development at the Apache Software Foundation. Struts is highly configurable, and has a large (and growing) feature list, including a Front Controller, action classes and mappings, utility classes for XML, automatic population of server-side JavaBeans, Web forms with validation, and some internationalization support. It also includes a set of custom tags for accessing server-side state, creating HTML, performing presentation logic, and templating. Some vendors have begun to adopt and evangelize Struts. Struts has a great deal of mindshare, and can be considered an industrial-strength framework suitable for large applications. But Struts is not yet a "standard" for which J2EE product providers can interoperably and reliably create tools.

- **JavaServer Faces**—A Java Community Process effort (JSR-127) is currently defining a standardized Web application framework called JavaServer Faces. Current standard Web-tier technologies offer only the means for creating general content for consumption by the client. There is currently no standard server-side GUI component or dispatching model. JavaServer Faces will be an architecture and a set of APIs for dispatching requests to Web-tier model JavaBeans; for maintaining stateful, server-side representations of reusable
HTML GUI components; and for supporting internationalization, validation, multiple client types, and accessibility. Standardization of the architecture and API will allow tool interoperation and the development of portable, reusable Web-tier GUI component libraries.

4.4.6 Separating Business Logic from Presentation

Placing business logic and presentation code in separate software layers is good design practice. The business layer provides only application functionality, with no reference to presentation. The presentation layer presents the data and input prompts to the user (or to another system), delegating application functionality to the business layer.

Separating business logic from presentation has several important benefits:

- **Minimizes impact of change**—Business rules can be changed in their own layer, with little or no modification to the presentation layer. Application presentation or workflow can change without affecting code in the business layer.

- **Increases maintainability**—Most business logic occurs in more than one use case of a particular application. Business logic copied and pasted between components expresses the same business rule in two places in the application. Future changes to the rule require two edits instead of one. Business logic expressed in a separate component and accessed referentially can be modified in one place in the source code, producing behavior changes everywhere the component is used. Similar benefits are achieved by reusing presentation logic with server-side includes, custom tags, and stylesheets.

- **Provides client independence and code reuse**—Intermingling data presentation and business logic ties the business logic to a particular type of client. For example, business logic implemented in a scriptlet is not usable by a servlet or an application client; the code must be reimplemented for the other client types. Business logic that is available referentially as simple method calls on business objects can be used by multiple client types.

- **Separates developer roles**—Code that deals with data presentation, request processing, and business rules all at once is difficult to read, especially for a developer who may specialize in only one of these areas. Separating business logic and presentation allows developers to concentrate on their area of expertise.
4.4.7 Web-Tier State

Data that a Web-tier component uses to create a response is called state. Examples of such data include the inventory data needed by a JSP page that lists items for sale, the contents of an online shopping cart maintained by a servlet, and the timestamp placed on an incoming request by a servlet filter.

State maintenance decisions have an enormous impact on application performance, availability, and scalability. Such decisions include choosing the tier to manage state, selecting the appropriate scope for each item of state, and effectively tracking conversational state in a distributed environment.

Note that the J2EE platform specification does not require that session state be recoverable after a crash or restart of a component container. Some J2EE implementations provide, as an extension, containers that can recover session state after a restart. Choosing such an implementation can simplify application design, but makes an application less portable, because it relies on a non-standard extension.

4.4.7.1 State Scope

Each item of Web-tier state has scope, which determines the accessibility and lifetime of the item. Web-tier state is accessible to servlets, servlet filters, and JSP pages. Briefly, Web-tier state can be maintained in four scopes:

Application scope is “global memory” for a Web application. Application-scope state is stored in the Web container’s ServletContext object. (See the caveat on using context attributes in distributable servlets on page 126.) All servlets in an application share objects in application scope. The servlet developer is responsible for thread safety when accessing objects in application scope. An inventory object in application scope, for example, is accessible to all servlets, servlet filters, and JSP pages in the application. State in application scope exists for the lifetime of the application, unless it is explicitly removed.

Session scope contains data specific to a user session. HTTP is a “stateless” protocol, meaning that it has no way of distinguishing users from one another or for maintaining data on users’ behalf. Session attributes are named object references that are associated with a user session. The servlet API allows a developer to create a session attribute and access or update it in subsequent requests. Session-scope state for an HttpServlet is stored in the Web container’s HttpSession object (available from the HttpServletRequest argument to the service method). State in session scope is accessible to all Web components in the application and across multiple servlet invocations. However, it is accessible only within an individual user session. An online shopping cart is an example of data in
session scope, because the contents of the cart are specific to a single client session and available across multiple server requests. A session ends when it is explicitly closed, when it times out after a period of inactivity, or when its container is shut down or crashes. Unless removed explicitly, state in session scope lasts until the session ends.

Request scope contains data specific to an individual server request, and is discarded when the service method returns. A Web component can read or modify data in request scope and then “forward” the request to another component. The component to which the request is forwarded then has access to the state. State in request scope is stored in a ServletRequest object, so it is accessible to any component receiving the request. For example, when a servlet places a timestamp in a ServletRequest and then forwards the request to another servlet, the timestamp is in request scope.

Page scope, applicable only to JSP pages, contains data that are only valid in the context of a single page. Page scope state is stored in a JSP page’s PageContext object. When one JSP page forwards to or includes another, each page defines its own scope. Page scope state is discarded when the program flow of control exits the page.

### 4.4.7.2 Performance Implications of State Scope

Selecting the appropriate scope for an item of state depends largely on the purpose of that item in the application. It would not make sense, for example, to place a shopping cart class in application scope, because then all shoppers would have to share the same cart. Shopping carts, because they are specific to a user session, are most appropriately kept in session scope. But shopping cart contents maintained in Client-tier cookies would be in request scope, because they would be transmitted to the Web tier with each request. Maintaining session state in cookies is discouraged, even though this approach may be more easily scalable than using session attributes. See “Avoid Using Cookies Directly,” starting on page 122 for more details.

Each state scope has implications for scalability, performance, and reliability. State in page or request scope is less likely to cause trouble, since such data are usually not large or long-lived enough to cause resource problems. State in application scope is usually manageable if it is read-only. Entity enterprise beans are the recommended technology for maintaining writable application-scope state. Entity beans are designed for scalable, concurrent access to shared data and logic. See Section 5.4 on page 142 for more information.
State in session scope has the greatest impact on Web application scalability and performance. Separate session-scope state accumulates for each connected user, unlike application-scope state, which is shared between all users and servlets. Also, session-scope state exists across requests, unlike request-scope state, which is discarded when a response is served.

**4.4.7.2.1 How the Web Container Manages Session State**

Application servers typically track user sessions with some combination of cookies and/or URL rewriting to store a session ID on the client. The session ID identifies the session, and the server is responsible for associating each HttpServletRequest with its corresponding HttpSession object. The J2EE server handles the details of using cookies and URL rewriting. The section “Maintaining Client State” in The J2EE Tutorial explains in detail how to manage Web-tier session state.

**4.4.7.3 Web-Tier State Recommendations**

When using enterprise beans, it’s best to maintain session state with stateful session beans in the EJB tier. For Web-only applications, maintain the state in the Web tier as session attributes (using HttpSession). The following sections discuss the rationale for these recommendations.

**4.4.7.3.1 Maintain Session State with Stateful Session Beans**

Maintaining session state in stateful session beans is a BluePrints best practice. Web-tier components can access the session state through the stateful session bean’s component interface and store just the reference as a session attribute. You can maximize the runtime performance of this approach by choosing a J2EE server product that permits use of local EJB interfaces from co-located Web components.

Reasons to prefer stateful session beans over other approaches to maintaining session state include:

- **Thread safety**—Enterprise beans are thread-safe. By contrast, sophisticated thread-safe servlets are difficult to write.

- **Lifecycle management**—The EJB container manages the lifecycle of enterprise beans components, automatically creating new instances, and activating and passivating instances as necessary to optimize performance.
• **Client type neutrality**—Enterprise beans can be accessed, either directly or through some sort of adapter, from multiple client types. This contrasts with HTTP session attributes, which are available only to HTTP clients.

For example, the sample application stores session state in stateful session beans `ShoppingClientControllerEJB` and `EJBClientControllerEJB`. For more on stateful session beans, see Chapter 5.

### 4.4.7.3.2 Maintain Web-Tier Session State in Session Attributes

Applications that don’t use enterprise beans should maintain session state in session attributes, using `HttpSession`'s methods `getAttribute` and `setAttribute`. These methods allow the Web container to maintain the state in a way that is most effective for that particular application and server. Session attributes free the developer from the details of session state management, and ensure portability and scalability of Web components.

The alternative to using session attributes is to create your own solution. The Web container (via `HttpSession`) provides services such as cookie management, session IDs, and so on. Writing custom Web-tier state management code is usually redundant. Don’t make work for yourself!

For more guidelines, see Section 4.4.7 on page 116, and also the section “Maintaining Client State” in *The J2EE Tutorial*.

Advantages of using session attributes include:

• **Easy implementation**—Because the application server handles the implementation of `HttpSession`, the developer is freed from bothering with the details of designing, implementing, and testing code for managing session state.

• **Optimization**—An application server’s `HttpSession` implementation is optimized and tested for that server, and therefore will probably be more efficient and reliable than a custom solution.

• **Potentially richer feature set**—An application server’s implementation of session state management may include such features as failover, cluster support, and so on, that go beyond the base-level requirements of the J2EE platform specifications. The system architect can select a server platform with the differentiating features that best suit application requirements, while maintaining J2EE technology compatibility and portability.
• **Portability**—The HttpSession interface is standardized, so it must pass the J2EE Compatibility Test Suite (CTS) across all J2EE-branded application servers. For more on the role of the CTS and J2EE branding, see the compatibility reference listed in “References and Resources” on page 127.

• **Scalability**— HttpSession can most effectively manage storage of Web-tier session state in caches and/or server clusters.

• **Evolvability**—Application server vendors are constantly improving their offerings. Servers will maintain existing interfaces for backward compatibility, even as they add features that improve performance and reliability. An HttpSession implementation that works properly today will work better tomorrow as improved server versions become available, with little or no change to the source code.

But session attributes have these important disadvantages:

• **Limited to Web clients**—The Web tier is by definition limited to servicing Web clients, so HttpSession interface is limited to HTTP communications. Other client types will require reimplementation of session state management.

• **Session state not guaranteed to survive Web container crashes**—Some application servers maintain persistent session state or provide failover, so sessions can span container crashes or restarts. But not all servers support that functionality, because the specification doesn’t require it. As a result, restarting a container can invalidate all sessions in progress, losing all of their state. If this is a problem for your application, either consider selecting a server that provides persistent sessions or session failover (which compromises portability), or consider storing session state in the EIS tier.

4.4.7.3.3 Share Data among Servlets and JSP Pages with JavaBeans Components

The standard JSP tag useBean accesses an attribute in application, session, request, or page scope as a JavaBean component. Standard actions setProperty and getProperty get and set the attributes’ properties using JavaBeans property accessor. Servlets have access to these attributes as well, so data shared between JSP pages and servlets is best maintained in JavaBeans classes. Code Example 4.12 shows a servlet setting a session-scope attribute of type UserDataBean, naming it UserData.
public void service(HttpServletRequest req, 
                    HttpServletResponse res) throws... {
    HttpSession session = req.getSession();
    UserDataBean userData = new UserData;
    userData.setName("Moliere");
    session.setAttribute("userData", userData);
    ...

Code Example 4.12  Setting a Session Attribute’s Value to a JavaBean Instance

When servlets are called, or the same servlet is called again, it can access the UserDataBean using the method HttpSession.getAttribute, as shown in Code Example 4.13.

    HttpSession session = req.getSession();
    UserDataBean userData = (UserDataBean)session.getAttribute("userData");
    String userName = userData.getUsername();

Code Example 4.13  Accessing a Session Attribute JavaBean Instance from a Servlet

A JSP page can access the UserDataBean using the standard tag useBean, as shown in Code Example 4.14. This creates the named JavaBean instance if it does not already exist. The remainder of Code Example 4.14 shows how to get or set the properties of the userData attribute by using getProperty and setProperty.

    <!-- Declare that the page uses session attribute UserData -->
    <jsp:useBean id="userData" type="UserDataBean" scope="session"/>

    <!-- get the userData property userData-->
    <jsp:getProperty name="userData" property="username"/>

    <!-- set all userData properties to values of corresponding
    request parameter names -->
    <jsp:setProperty name="userData" property="*"/>

    <!-- set userData property "username" to value of request
    parameter "username" -->
<jsp:setProperty name="userData" property="username"/>

<!-- set userData property "username" to value of request parameter "new_user_name" -->
<jsp:setProperty name="userData" property="username"
    param="new_user_name"/>

<!-- set userData property "username" to string "Unknown User" -->
<jsp:setProperty name="userData" property="username"
    value="Unknown User"/>

Code Example 4.14  Using JavaBean Properties in a JSP Page

These examples show how to share information between components in session scope. These techniques work similarly for application, page, and request scopes.

4.4.7.3.4 Avoid Using Cookies Directly
Avoid using cookies directly for storing session state in most applications. Implementation details of session state storage are best left to the application server. Using either a stateful session bean or a Web container’s HttpSession implementation can provide reliable access to session state through a portable interface. Using the standard interface saves the development time of implementing and maintaining a custom solution.

Disadvantages to using cookies for session state include:

- Cookies are controlled by a low-level API, which is more difficult to use than the other approaches.
- All data for a session are kept on the client. Corruption, expiration, or purging of cookie files can result in incomplete, inconsistent, or missing information.
- Size limitations on cookies differ by browser type and version, but the least-common-denominator approach limits the maximum cookie size to 4,096 bytes. This limitation can be eliminated by storing just references to data (session ids, user ids, and so on) in cookies, and retrieving the data as necessary from another tier (at the cost of increased server complexity and resource usage).
• Servlets and JSP pages that rely exclusively on cookies for client-side session state will not operate properly for all clients. Cookies may not be available for many reasons: The user may have disabled them, the browser version may not support them, the browser may be behind a firewall that filters cookies, and so on.

• Because Web clients transmit to a server only those cookies that it created by that server, servers with different domain names can’t share cookie data. For example, JavaPetStore.com may want to allow users to shop from their own shopping sites, as well as from JavaPetFood.com. But because JavaPetFood.com can’t access JavaPetStore.com’s cookies, there’s no easy way to unify the shopping sessions between the two servers.

• Historically, cookie implementations in both browsers and servers have tended to be buggy, or vary in their conformance to standards. While you may have control of your servers, many people still use buggy or nonconformant versions of browsers.

• Browser instances share cookies, so users cannot have multiple simultaneous sessions.

• Cookies work only for HTTP clients, because they are a feature of the HTTP protocol. Notice that while package javax.servlet.http supports session management (via class HttpSession), package javax.servlet has no such support.

Exceptions to this guideline exist. For example, a browser cookie could contain a user’s login name and locale to facilitate sign on. Because of the drawbacks described here, cookies should be used to maintain session state only when there is a clear reason to do so.

4.4.8 Distributable Web Applications

The J2EE platform provides optional support for distributed Web applications. A distributed Web application runs simultaneously in multiple Web containers. When a Web application is marked distributable in its deployment descriptor, the container may (but is not required to) create multiple instances of the servlet, in multiple JVM instances, and potentially on multiple machines. Distributing a servlet improves scalability, because it allows Web request load to be spread across multi-
ple servers. It can also improve availability by providing transparent failover between servlet instances.

### 4.4.8.1 Distributed Servlet Instances

By default, only one servlet instance per servlet definition is allowed for servlets that are neither in an application marked distributable, nor implement SingleThreadModel. Servlets in applications marked distributable have exactly one servlet instance per servlet definition for each Java virtual machine (JVM). The container may create and pool multiple instances of a servlet that implements SingleThreadModel, but using SingleThreadModel is discouraged.

At any particular time, session attributes for a given session are local to a particular JVM. The distributed runtime environment therefore acts to ensure that all requests associated with a given session are handled by exactly one JVM at a time. A servlet’s session state may migrate to, or be failed-over to, some other JVM between requests.

### 4.4.8.2 Distributed Conversational State

Distributing multiple instances of a servlet across multiple JVM instances raises the issue of how to support conversational state. If each request a user makes can be routed to a different server, how can the system track that user’s session state?

J2EE product providers solve this problem in different ways. One approach, called sticky server affinity, associates a particular client with a particular servlet instance for the duration of the session. This solves the session state problem, because each session is “owned” by a particular servlet. But this approach can compromise availability, because when a servlet, JVM instance, or server crashes, all of the associated sessions can be lost. Sticky server affinity can also make load balancing more difficult, because sessions are “stuck” on the servers where they started.

Another approach to solving the distributed conversational state problem is state migration. State migration serializes and moves or copies session state between servlet instances. This solution maintains the availability benefits of servlet distribution and facilitates load balancing, because sessions can be moved from more- to less-loaded servers. But state migration can increase network traffic between clustered servers. Each time a client updates session state, all redundant copies of that state must be updated. If session state is stored in a database (as is often the case), the database can become a performance bottleneck. The contain-
ers must also cooperate to resolve simultaneous update collisions, where two clients accessing the same session (one browser window opened from another, for example) update different copies of the same session state.

The J2EE platform specification gives the J2EE product provider the opportunity to add value by solving the issue of distributed conversational state in the implementation while maintaining the consistent J2EE interface. A good solution to this problem can be a selling point for a J2EE vendor. Designers considering a Web-tier-only architecture for high-performance applications should be sure to understand how prospective J2EE product providers address this issue.

Stateful session beans are designed specifically for handling distributed conversational state, but do so in the EJB tier, rather than in the Web tier. See Section 4.4.7.3.1 for more details.

4.4.8.3 Distributable Servlet Restrictions

Servlets used in a distributable application require some additional constraints. Most are necessary conditions for session state migration. These restrictions also apply for code in JSP pages and custom tags.

- **Session attributes must be either serializable or supported in distributed sessions by the Web container**—The Web container must accept instances of serializable classes as session attributes. A container must also accept a few other J2EE object types as session attributes: enterprise bean home and remote references, transaction contexts (javax.transaction.UserTransaction), and the JNDI context object for java:comp/env (javax.naming.Context). For any other types, the container may throw an IllegalArgumentException to indicate that the object cannot be moved between JVMs.

  Implementing Serializable in a session attribute does not guarantee that the container will use native Java serialization. The container is also not required to use any defined custom serialization methods, such as readObject or writeObject, that the class may define. It does ensure that session attribute values are preserved if the session is migrated. See Section 7.2.2 of the Java Servlet specification 2.3, and Section J2EE.6.5 of the J2EE platform specification 1.3 for more details.

- **Don’t store application state in static or instance variables**—Web containers are not required to maintain static or instance variable values when a session migrates. Code that depends on state stored in such variables will likely
not operate properly after session migration. Such state should be stored either as a session attribute, in an enterprise bean, or in a database.

- **Don’t use context attributes to share state between servlets**—Context attributes are stored in `ServletContext` and are shared by all servlets in a Web application. But context attributes are specific to the JVM instance in which they were created. Servlets that communicate by sharing context attributes may not operate properly if distributed, because context attributes do not replicate between Web containers in different JVM instances. To share data between distributed servlets, place the data in a session object, store it in the EIS tier in a database or distributed cache, or use an enterprise bean.

One exception to this guideline is to use context attributes as a shared data cache between the servlets in each Web container. Cache hits and misses affect only an application’s performance, not its behavior.

- **Don’t depend on servlet context events or HTTP session events**—The Web container is not required to propagate such events between containers in a distributed environment.

### 4.5 Summary

The Web tier of the J2EE platform makes J2EE applications available on the World Wide Web. JSP pages and servlets are Web components that supersede legacy technologies by providing portable high-level system services. These services include transactions, data access, state maintenance, security, and distribution. Using custom tags and standard tag libraries in JSP pages improves code quality and eases maintenance.

The Model-View-Controller (MVC) architectural design pattern is recommended for most interactive Web applications. MVC makes application functionality more reusable, and simplifies adding and modifying client types, data views, and workflow.

A Model 1 application is a set of JSP pages that are statically linked to one another. A Model 2 application has a centralized controller that dynamically performs request dispatching and view selection. Model 2 is the preferred architecture for Web applications, because it provides more flexibility and is more maintainable than a Model 1 design.

A Web-tier application framework is a domain-neutral layer of services, usually based on MVC, that simplifies constructing an interactive Web applica-
tion. Such a framework can reduce an application’s time-to-market, improve code quality, and ease maintenance. It’s usually preferable to choose an existing framework rather than to build one.

The simplest framework design has a single controller that receives requests from browsers, dispatches calls to an application model, and displays results. Multiple controllers can support multiple types of Web-tier clients by communicating with them in their native protocols. A protocol router provides a single point of control for application-wide services such as security and logging.

A Web-tier templating mechanism can improve page layout consistency. The templating mechanism uses a template file to assemble individual views into a single composite view. A template file specifies layout for a set of composite views. Templating makes an application more flexible and makes content more reusable.

Servlets are useful for implementing application services, generating binary content, and controlling applications. JSP pages are best for creating textual content with embedded references to external data. Servlet filters can extend the functionality of an existing servlet, JSP page, or servlet filter.

Web application state resides in either application scope, session scope, request scope, or page scope. State in session scope has the greatest impact on scalability, because its size is proportional to the number of users. Using a stateful session bean is the recommended way to maintain session-scope state. Web-only applications should store session-scope state in HTTP session attributes.

Some J2EE products allow a Web application to be distributed for improved scalability and availability. How a platform implementation manages load in a distributed application is vendor-specific. JSP pages, custom tags, and servlets in a distributed Web application must follow additional programming restrictions.

### 4.6 References and Resources

For tutorials on using APIs described in this chapter, see the Web Technology section of:


• The Java™ Servlet 2.3 and JavaServer Pages™ 1.2 specifications (JSR-53) are available for download in PDF format at <http://www.jcp.org/aboutJava/communityprocess/final/jsr053/>

• For information on compatibility and the CTS, see <http://java.sun.com/j2ee/compatibility.html>

• The Apache Struts project can be found at <http://jakarta.apache.org/struts/index.html>

• The Apache Jakarta taglibs are available at <http://jakarta.apache.org/taglibs/index.html>
INTEGRATING SERVLETS AND JSP: THE MODEL VIEW CONTROLLER (MVC) ARCHITECTURE
INTEGRATING SERVLETS AND JSP: THE MODEL VIEW CONTROLLER (MVC) ARCHITECTURE

Topics in This Chapter

- Understanding the benefits of MVC
- Using RequestDispatcher to implement MVC
- Forwarding requests from servlets to JSP pages
- Handling relative URLs
- Choosing among different display options
- Comparing data-sharing strategies
- Forwarding requests from JSP pages
- Including pages instead of forwarding to them
Servlets are good at data processing: reading and checking data, communicating with databases, invoking business logic, and so on. JSP pages are good at presentation: building HTML to represent the results of requests. This chapter describes how to combine servlets and JSP pages to best make use of the strengths of each technology.

15.1 Understanding the Need for MVC

Servlets are great when your application requires a lot of real programming to accomplish its task. As illustrated earlier in this book, servlets can manipulate HTTP status codes and headers, use cookies, track sessions, save information between requests, compress pages, access databases, generate JPEG images on-the-fly, and perform many other tasks flexibly and efficiently. But, generating HTML with servlets can be tedious and can yield a result that is hard to modify.

That’s where JSP comes in: as illustrated in Figure 15–1, JSP lets you separate much of the presentation from the dynamic content. That way, you can write the HTML in the normal manner, even using HTML-specific tools and putting your Web content developers to work on your JSP documents. JSP expressions, scriptlets, and declarations let you insert simple Java code into the servlet that results from the JSP page, and directives let you control the overall layout of the page. For more complex requirements, you can wrap Java code inside beans or even define your own JSP tags.
Great. We have everything we need, right? Well, no, not quite. The assumption behind a JSP document is that it provides a single overall presentation. What if you want to give totally different results depending on the data that you receive? Scripting expressions, beans, and custom tags, although extremely powerful and flexible, don’t overcome the limitation that the JSP page defines a relatively fixed, top-level page appearance. Similarly, what if you need complex reasoning just to determine the type of data that applies to the current situation? JSP is poor at this type of business logic.

The solution is to use both servlets and JavaServer Pages. In this approach, known as the Model View Controller (MVC) or Model 2 architecture, you let each technology concentrate on what it excels at. The original request is handled by a servlet. The servlet invokes the business-logic and data-access code and creates beans to represent the results (that's the model). Then, the servlet decides which JSP page is appropriate to present those particular results and forwards the request there (the JSP page is the view). The servlet decides what business logic code applies and which JSP page should present the results (the servlet is the controller).

**MVC Frameworks**

The key motivation behind the MVC approach is the desire to separate the code that creates and manipulates the data from the code that presents the data. The basic tools needed to implement this presentation-layer separation are standard in the
15.2 Implementing MVC with RequestDispatcher

The servlet API and are the topic of this chapter. However, in very complex applications, a more elaborate MVC framework is sometimes beneficial. The most popular of these frameworks is Apache Struts; it is discussed at length in Volume 2 of this book. Although Struts is useful and widely used, you should not feel that you must use Struts in order to apply the MVC approach. For simple and moderately complex applications, implementing MVC from scratch with RequestDispatcher is straightforward and flexible. Do not be intimidated: go ahead and start with the basic approach. In many situations, you will stick with the basic approach for the entire life of your application. Even if you decide to use Struts or another MVC framework later, you will recoup much of your investment because most of your work will also apply to the elaborate frameworks.

Architecture or Approach?

The term “architecture” often connotes “overall system design.” Although many systems are indeed designed with MVC at their core, it is not necessary to redesign your overall system just to make use of the MVC approach. Not at all. It is quite common for applications to handle some requests with servlets, other requests with JSP pages, and still others with servlets and JSP acting in conjunction as described in this chapter. Do not feel that you have to rework your entire system architecture just to use the MVC approach: go ahead and start applying it in the parts of your application where it fits best.

15.2 Implementing MVC with RequestDispatcher

The most important point about MVC is the idea of separating the business logic and data access layers from the presentation layer. The syntax is quite simple, and in fact you should be familiar with much of it already. Here is a quick summary of the required steps; the following subsections supply details.

1. **Define beans to represent the data.** As you know from Section 14.2, beans are just Java objects that follow a few simple conventions. Your first step is define beans to represent the results that will be presented to the user.

2. **Use a servlet to handle requests.** In most cases, the servlet reads request parameters as described in Chapter 4.

3. **Populate the beans.** The servlet invokes business logic (application-specific code) or data-access code (see Chapter 17) to obtain the results. The results are placed in the beans that were defined in step 1.
4. **Store the bean in the request, session, or servlet context.** The servlet calls `setAttribute` on the request, session, or servlet context objects to store a reference to the beans that represent the results of the request.

5. **Forward the request to a JSP page.** The servlet determines which JSP page is appropriate to the situation and uses the `forward` method of `RequestDispatcher` to transfer control to that page.

6. **Extract the data from the beans.** The JSP page accesses beans with `jsp:useBean` and a scope matching the location of step 4. The page then uses `jsp:getProperty` to output the bean properties. The JSP page does not create or modify the bean; it merely extracts and displays data that the servlet created.

**Defining Beans to Represent the Data**

Beans are Java objects that follow a few simple conventions. In this case, since a servlet or other Java routine (never a JSP page) will be creating the beans, the requirement for an empty (zero-argument) constructor is waived. So, your objects merely need to follow the normal recommended practices of keeping the instance variables private and using accessor methods that follow the get/set naming convention.

Since the JSP page will only access the beans, not create or modify them, a common practice is to define value objects: objects that represent results but have little or no additional functionality.

**Writing Servlets to Handle Requests**

Once the bean classes are defined, the next task is to write a servlet to read the request information. Since, with MVC, a servlet responds to the initial request, the normal approaches of Chapters 4 and 5 are used to read request parameters and request headers, respectively. The shorthand `populateBean` method of Chapter 4 can be used, but you should note that this technique populates a form bean (a Java object representing the form parameters), not a result bean (a Java object representing the results of the request).

Although the servlets use the normal techniques to read the request information and generate the data, they do not use the normal techniques to output the results. In fact, with the MVC approach the servlets do not create any output; the output is completely handled by the JSP pages. So, the servlets do not call `response.setContentType`, `response.getWriter`, or `out.println`. 
Populating the Beans

After you read the form parameters, you use them to determine the results of the request. These results are determined in a completely application-specific manner. You might call some business logic code, invoke an Enterprise JavaBeans component, or query a database. No matter how you come up with the data, you need to use that data to fill in the value object beans that you defined in the first step.

Storing the Results

You have read the form information. You have created data specific to the request. You have placed that data in beans. Now you need to store those beans in a location that the JSP pages will be able to access.

A servlet can store data for JSP pages in three main places: in the HttpServletRequest, in the HttpSession, and in the ServletContext. These storage locations correspond to the three nondefault values of the scope attribute of jsp:useBean: that is, request, session, and application.

- **Storing data that the JSP page will use only in this request.**
  First, the servlet would create and store data as follows:

  ```java
  ValueObject value = new ValueObject(...);
  request.setAttribute("key", value);
  ```

  Next, the servlet would forward the request to a JSP page that uses the following to retrieve the data.

  ```xml
  <jsp:useBean id="key" type="somePackage.ValueObject"
               scope="request" />
  ```

  Note that request attributes have nothing to do with request parameters or request headers. The request attributes are independent of the information coming from the client; they are just application-specific entries in a hash table that is attached to the request object. This table simply stores data in a place that can be accessed by both the current servlet and JSP page, but not by any other resource or request.

- **Storing data that the JSP page will use in this request and in later requests from the same client.** First, the servlet would create and store data as follows:

  ```java
  ValueObject value = new ValueObject(...);
  HttpSession session = request.getSession();
  session.setAttribute("key", value);
  ```
Next, the servlet would forward to a JSP page that uses the following to retrieve the data:

```jsp
<jsp:useBean id="key" type="somePackage.ValueObject" scope="session" />
```

- **Storing data that the JSP page will use in this request and in later requests from any client.** First, the servlet would create and store data as follows:

```java
ValueObject value = new ValueObject(...);
getServletContext().setAttribute("key", value);
```

Next, the servlet would forward to a JSP page that uses the following to retrieve the data:

```jsp
<jsp:useBean id="key" type="somePackage.ValueObject" scope="application" />
```

As described in Section 15.3, the servlet code is normally synchronized to prevent the data changing between the servlet and the JSP page.

### Forwarding Requests to JSP Pages

You forward requests with the `forward` method of `RequestDispatcher`. You obtain a `RequestDispatcher` by calling the `getRequestDispatcher` method of `ServletRequest`, supplying a relative address. You are permitted to specify addresses in the `WEB-INF` directory; clients are not allowed to directly access files in `WEB-INF`, but the server is allowed to transfer control there. Using locations in `WEB-INF` prevents clients from inadvertently accessing JSP pages directly, without first going through the servlets that create the JSP data.

#### Core Approach

*If your JSP pages only make sense in the context of servlet-generated data, place the pages under the `WEB-INF` directory. That way, servlets can forward requests to the pages, but clients cannot access them directly.*

Once you have a `RequestDispatcher`, you use `forward` to transfer control to the associated address. You supply the `HttpServletRequest` and `HttpServletResponse` as arguments. Note that the `forward` method of `RequestDispatcher` is
15.2 Implementing MVC with RequestDispatcher

quite different from the `sendRedirect` method of `HttpServletRequest` (Section 7.1). With `forward`, there is no extra response/request pair as with `sendRedirect`. Thus, the URL displayed to the client does not change when you use `forward`.

Core Note

When you use the `forward` method of `RequestDispatcher`, the client sees the URL of the original servlet, not the URL of the final JSP page.

For example, Listing 15.1 shows a portion of a servlet that forwards the request to one of three different JSP pages, depending on the value of the operation request parameter.

**Listing 15.1 Request Forwarding Example**

```java
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException {
    String operation = request.getParameter("operation");
    if (operation == null) {
        operation = "unknown";
    }
    String address;
    if (operation.equals("order")) {
        address = "/WEB-INF/Order.jsp";
    } else if (operation.equals("cancel")) {
        address = "/WEB-INF/Cancel.jsp";
    } else {
        address = "/WEB-INF/UnknownOperation.jsp";
    }
    RequestDispatcher dispatcher =
        request.getRequestDispatcher(address);
    dispatcher.forward(request, response);
}
```

**Forwarding to Static Resources**

In most cases, you forward requests to JSP pages or other servlets. In some cases, however, you might want to send requests to static HTML pages. In an e-commerce site, for example, requests that indicate that the user does not have a valid account name might be forwarded to an account application page that uses HTML forms to
gather the requisite information. With GET requests, forwarding requests to a static HTML page is perfectly legal and requires no special syntax; just supply the address of the HTML page as the argument to getRequestDispatcher. However, since forwarded requests use the same request method as the original request, POST requests cannot be forwarded to normal HTML pages. The solution to this problem is to simply rename the HTML page to have a .jsp extension. Renaming somefile.html to somefile.jsp does not change its output for GET requests, but somefile.html cannot handle POST requests, whereas somefile.jsp gives an identical response for both GET and POST.

Redirecting Instead of Forwarding

The standard MVC approach is to use the forward method of RequestDispatcher to transfer control from the servlet to the JSP page. However, when you are using session-based data sharing, it is sometimes preferable to use response.sendRedirect.

Here is a summary of the behavior of forward.

- Control is transferred entirely on the server. No network traffic is involved.
- The user does not see the address of the destination JSP page and pages can be placed in WEB-INF to prevent the user from accessing them without going through the servlet that sets up the data. This is beneficial if the JSP page makes sense only in the context of servlet-generated data.

Here is a summary of sendRedirect.

- Control is transferred by sending the client a 302 status code and a Location response header. Transfer requires an additional network round trip.
- The user sees the address of the destination page and can bookmark it and access it independently. This is beneficial if the JSP is designed to use default values when data is missing. For example, this approach would be used when redisplaying an incomplete HTML form or summarizing the contents of a shopping cart. In both cases, previously created data would be extracted from the user’s session, so the JSP page makes sense even for requests that do not involve the servlet.
Extracting Data from Beans

Once the request arrives at the JSP page, the JSP page uses `jsp:useBean` and `jsp:getProperty` to extract the data. For the most part, this approach is exactly as described in Chapter 14. There are two differences however:

- **The JSP page never creates the objects.** The servlet, not the JSP page, should create all the data objects. So, to guarantee that the JSP page will not create objects, you should use

  ```jsp
  <jsp:useBean ... type="package.Class" />
  ```

  instead of

  ```jsp
  <jsp:useBean ... class="package.Class" />
  ```

- **The JSP page should not modify the objects.** So, you should use `jsp:getProperty` but not `jsp:setProperty`.

  The scope you specify should match the storage location used by the servlet. For example, the following three forms would be used for request-, session-, and application-based sharing, respectively.

  ```jsp
  <jsp:useBean id="key" type="somePackage.SomeBeanClass" scope="request" />
  <jsp:useBean id="key" type="somePackage.SomeBeanClass" scope="session" />
  <jsp:useBean id="key" type="somePackage.SomeBeanClass" scope="application" />
  ```

15.3 Summarizing MVC Code

This section summarizes the code that would be used for request-based, session-based, and application-based MVC approaches.

Request-Based Data Sharing

With request-based sharing, the servlet stores the beans in the `HttpServlet-Request`, where they are accessible only to the destination JSP page.
Servlet

ValueObject value = new ValueObject(...);
request.setAttribute("key", value);
RequestDispatcher dispatcher =
    request.getRequestDispatcher("/WEB-INF/SomePage.jsp");
dispatcher.forward(request, response);

JSP Page

<jsp:useBean id="key" type="somePackage.ValueObject"
    scope="request" />
<jsp:getProperty name="key" property="someProperty" />

Session-Based Data Sharing

With session-based sharing, the servlet stores the beans in the HttpSession, where
they are accessible to the same client in the destination JSP page or in other pages.

Servlet

ValueObject value = new ValueObject(...);
HttpSession session = request.getSession();
session.setAttribute("key", value);
RequestDispatcher dispatcher =
    request.getRequestDispatcher("/WEB-INF/SomePage.jsp");
dispatcher.forward(request, response);

JSP Page

<jsp:useBean id="key" type="somePackage.ValueObject"
    scope="session" />
<jsp:getProperty name="key" property="someProperty" />

Application-Based Data Sharing

With application-based sharing, the servlet stores the beans in the ServletContext, where they are accessible to any servlet or JSP page in the Web application. To guarantee that the JSP page extracts the same data that the servlet inserted, you should synchronize your code as below.

Servlet

synchronized(this) {
    ValueObject value = new ValueObject(...);
    getServletContext().setAttribute("key", value);
15.4 Interpreting Relative URLs in the Destination Page

Although a servlet can forward the request to an arbitrary location on the same server, the process is quite different from that of using the sendRedirect method of HttpServletResponse. First, sendRedirect requires the client to reconnect to the new resource, whereas the forward method of RequestDispatcher is handled completely on the server. Second, sendRedirect does not automatically preserve all of the request data; forward does. Third, sendRedirect results in a different final URL, whereas with forward, the URL of the original servlet is maintained.

This final point means that if the destination page uses relative URLs for images or style sheets, it needs to make them relative to the servlet URL or the server root, not to the destination page’s actual location. For example, consider the following style sheet entry:

```html
<Link REL=stylesheet
     HREF="my-styles.css"
     TYPE="text/css">
</Link>
```

If the JSP page containing this entry is accessed by means of a forwarded request, my-styles.css will be interpreted relative to the URL of the originating servlet, not relative to the JSP page itself, almost certainly resulting in an error. The simplest solution to this problem is to give the full server path to the style sheet file, as follows.

```html
<Link REL=stylesheet
     HREF="/path/my-styles.css"
     TYPE="text/css">
</Link>
```

The same approach is required for addresses used in `<img src=...>` and `<a href=...>`.
15.5 Applying MVC: Bank Account Balances

In this section, we apply the MVC approach to an application that displays bank account balances. The controller servlet (Listing 15.2) reads a customer ID and passes that to some data-access code that returns a BankCustomer value bean (Listing 15.3). The servlet then stores the bean in the HttpServletRequest object where it will be accessible from destination JSP pages but nowhere else. If the account balance of the resulting customer is negative, the servlet forwards to a page designed for delinquent customers (Listing 15.4, Figure 15–2). If the customer has a positive balance of less than $10,000, the servlet transfers to the standard balance-display page (Listing 15.5, Figure 15–3). Next, if the customer has a balance of $10,000 or more, the servlet forwards the request to a page reserved for elite customers (Listing 15.6, Figure 15–4). Finally, if the customer ID is unrecognized, an error page is displayed (Listing 15.7, Figure 15–5).

Listing 15.2  ShowBalance.java

package coreservlets;

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

/** Servlet that reads a customer ID and displays
  * information on the account balance of the customer
  * who has that ID. */

public class ShowBalance extends HttpServlet {
  public void doGet(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    BankCustomer customer =
      BankCustomer.getCustomer(request.getParameter("id"));
    String address;
    if (customer == null) {
      address = "*/WEB-INF/bank-account/UnknownCustomer.jsp";
    } else if (customer.getBalance() < 0) {
      address = "*/WEB-INF/bank-account/NegativeBalance.jsp";
      request.setAttribute("badCustomer", customer);
    } else if (customer.getBalance() >= 10000) {
      address = "*/WEB-INF/bank-account/EliteCustomer.jsp";
      request.setAttribute("eliteCustomer", customer);
    } else {
      address = "*/WEB-INF/bank-account/StandardBalance.jsp";
      request.setAttribute("standardCustomer", customer);
    }
    request.getRequestDispatcher(address).forward(request, response);
  }
}
15.5 Applying MVC: Bank Account Balances

Listing 15.2  ShowBalance.java (continued)

```java
} else if (customer.getBalance() < 10000) {
    address = "*/WEB-INF/bank-account/NormalBalance.jsp";
    request.setAttribute("regularCustomer", customer);
} else {
    address = "*/WEB-INF/bank-account/HighBalance.jsp";
    request.setAttribute("eliteCustomer", customer);
}
RequestDispatcher dispatcher =
    request.getRequestDispatcher(address);
dispatcher.forward(request, response);
```

Listing 15.3  BankCustomer.java

```java
package coreservlets;
import java.util.*;
/** Bean to represent a bank customer. */
public class BankCustomer {
    private String id, firstName, lastName;
    private double balance;

    public BankCustomer(String id,
                        String firstName,
                        String lastName,
                        double balance) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
        this.balance = balance;
    }

    public String getId() {
        return(id);
    }

    public String getFirstName() {
        return(firstName);
    }
```
Listing 15.3  BankCustomer.java (continued)

```java
public String getLastName() {
    return(lastName);
}

public double getBalance() {
    return(balance);
}

public double getBalanceNoSign() {
    return(Math.abs(balance));
}

public void setBalance(double balance) {
    this.balance = balance;
}

// Makes a small table of banking customers.
private static HashMap customers;

static {
    customers = new HashMap();
    customers.put("id001",
        new BankCustomer("id001",
            "John",
            "Hacker",
            -3456.78));

    customers.put("id002",
        new BankCustomer("id002",
            "Jane",
            "Hacker",
            1234.56));

    customers.put("id003",
        new BankCustomer("id003",
            "Juan",
            "Hacker",
            987654.32));
}

/** Finds the customer with the given ID.
 * Returns null if there is no match.
 */
public static BankCustomer getCustomer(String id) {
    return((BankCustomer)customers.get(id));
}
```
### Listing 15.4 NegativeBalance.jsp

```html
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title>You Owe Us Money!</title>
<link rel="stylesheet" href="/bank-support/JSP-Styles.css" type="text/css">
</head>
<body>
<table border="5" align="center">
<tr><th class="title">
We Know Where You Live!
</th></tr>
</table>
<p>
<img src="/bank-support/Club.gif" align="left">
<jsp:useBean id="badCustomer" type="coreservlets.BankCustomer" scope="request" />
Watch out,<jsp:getProperty name="badCustomer" property="firstName" />
we know where you live.
</p>
Pay us the 
$jsp:getProperty name="badCustomer" property="balanceNoSign" /
you owe us before it is too late!
</body></html>
```

### Listing 15.5 NormalBalance.jsp

```html
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title>Your Balance</title>
<link rel="stylesheet" href="/bank-support/JSP-Styles.css" type="text/css">
</head>
<body>
<table border="5" align="center">
<tr><th class="title">
Your Balance
</th></tr>
</table>
</body></html>
```
Listing 15.5 NormalBalance.jsp (continued)

```html
<body>
<img src="/bank-support/Money.gif" align="right">
<jsp:useBean id="regularCustomer"
    type="coreservlets.BankCustomer"
    scope="request" />

<ul>
    <li>First name: <jsp:getProperty name="regularCustomer"
        property="firstName" />
    </li>
    <li>Last name: <jsp:getProperty name="regularCustomer"
        property="lastName" />
    </li>
    <li>ID: <jsp:getProperty name="regularCustomer"
        property="id" />
    </li>
    <li>Balance: $<jsp:getProperty name="regularCustomer"
        property="balance" />
    </li>
</ul>
</body>
</html>
```

Figure 15–2  The ShowCustomer servlet with an ID corresponding to a customer with a negative balance.
15.5 Applying MVC: Bank Account Balances

Figure 15–3 The ShowCustomer servlet with an ID corresponding to a customer with a normal balance.

Listing 15.6 HighBalance.jsp

```html
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title>Your Balance</title>
<link rel=stylesheet href="/bank-support/JSP-Styles.css" type="text/css">
</head>
<body>
<table border=5 align="center">
<tr><th class="title">Your Balance</th></tr>
<tr><td><center><img src="/bank-support/Sailing.gif"></center>
<br clear="all">
<jsp:useBean id="eliteCustomer" type="coreServlets.BankCustomer" scope="request" />
It is an honor to serve you,
<jsp:getProperty name="eliteCustomer" property="firstName" />
<jsp:getProperty name="eliteCustomer" property="lastName" />
</td></tr>
</table>
</body>
</html>
```

It is an honor to serve you,
Since you are one of our most valued customers, we would like to offer you the opportunity to spend a mere fraction of your $\text{\$jsp:getProperty name="eliteCustomer" property="balance"} /$ on a boat worthy of your status. Please visit our boat store for more information.

</BODY></HTML>

Figure 15-4  The ShowCustomer servlet with an ID corresponding to a customer with a high balance.
15.6 Comparing the Three Data-Sharing Approaches

In the MVC approach, a servlet responds to the initial request. The servlet invokes code that fetches or creates the business data, places that data in beans, stores the beans, and forwards the request to a JSP page to present the results. But, where does the servlet store the beans?

The most common answer is, in the request object. That is the only location to which the JSP page has sole access. However, you sometimes want to keep the results

Listing 15.7 UnknownCustomer.jsp

```html
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
  <head>
    <title>Unknown Customer</title>
    <link rel="stylesheet" href="/bank-support/JSP-Styles.css" type="text/css">
  </head>
  <body>
    <table border="5" align="center">
      <tr><th class="title">
          Unknown Customer</th>
      </tr>
      <p>
        Unrecognized customer ID.
      </p>
    </table>
  </body>
</html>
```

Figure 15–5 The ShowCustomer servlet with an unknown customer ID.
around for the same client (session-based sharing) or store Web-application-wide data (application-based sharing).
This section gives a brief example of each of these approaches.

**Request-Based Sharing**

In this example, our goal is to display a random number to the user. Each request should result in a new number, so request-based sharing is appropriate.

To implement this behavior, we need a bean to store numbers (Listing 15.8), a servlet to populate the bean with a random value (Listing 15.9), and a JSP page to display the results (Listing 15.10, Figure 15–6).

**Listing 15.8 NumberBean.java**

```java
package coreservlets;

public class NumberBean {
    private double num = 0;

    public NumberBean(double number) {
        setNumber(number);
    }

    public double getNumber() {
        return(num);
    }

    public void setNumber(double number) {
        num = number;
    }
}
```

**Listing 15.9 RandomNumberServlet.java**

```java
package coreservlets;

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

/** Servlet that generates a random number, stores it in a bean, * and forwards to JSP page to display it. */
```
15.6 Comparing the Three Data-Sharing Approaches

Figure 15–6 Result of RandomNumberServlet.

Listing 15.9 RandomNumberServlet.java (continued)

```java
public class RandomNumberServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                        HttpServletResponse response)
                throws ServletException, IOException {
        NumberBean bean = new NumberBean(Math.random());
        request.setAttribute("randomNum", bean);
        String address = "/WEB-INF/mvc-sharing/RandomNum.jsp";
        RequestDispatcher dispatcher =
            request.getRequestDispatcher(address);
        dispatcher.forward(request, response);
    }
}
```

Listing 15.10 RandomNum.jsp

```html
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Random Number</TITLE>
<LINK REL=STYLESHEET
       HREF="/bank-support/JSP-Styles.css"
       TYPE="text/css">
</HEAD>
<BODY>
<jsp:useBean id="randomNum" type="coreservlets.NumberBean"
            scope="request" />
<H2>Random Number:
<jsp:getProperty name="randomNum" property="number" />
</H2>
</BODY></HTML>
```
Session-Based Sharing

In this example, our goal is to display users’ first and last names. If the users fail to tell us their name, we want to use whatever name they gave us previously. If the users do not explicitly specify a name and no previous name is found, a warning should be displayed. Data is stored for each client, so session-based sharing is appropriate.

To implement this behavior, we need a bean to store names (Listing 15.11), a servlet to retrieve the bean from the session and populate it with first and last names (Listing 15.12), and a JSP page to display the results (Listing 15.13, Figures 15–7 and 15–8).

Listing 15.11 NameBean.java

```java
package coreservlets;

public class NameBean {
    private String firstName = "Missing first name";
    private String lastName = "Missing last name";

    public NameBean() {}

    public NameBean(String firstName, String lastName) {
        setFirstName(firstName);
        setLastName(lastName);
    }

    public String getFirstName() {
        return(firstName);
    }

    public void setFirstName(String newFirstName) {
        firstName = newFirstName;
    }

    public String getLastName() {
        return(lastName);
    }

    public void setLastName(String newLastName) {
        lastName = newLastName;
    }
}
```
**Listing 15.12**  
RegistrationServlet.java

```java
package coreservlets;

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

/** Reads firstName and lastName request parameters and forwards * to JSP page to display them. Uses session-based bean sharing * to remember previous values. */

public class RegistrationServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                      throws ServletException, IOException {
        HttpSession session = request.getSession();
        NameBean nameBean =
            (NameBean) session.getAttribute("nameBean");
        if (nameBean == null) {
            nameBean = new NameBean();
            session.setAttribute("nameBean", nameBean);
        }

        String firstName = request.getParameter("firstName");
        if ((firstName != null) && (!firstName.trim().equals(""))) {
            nameBean.setFirstName(firstName);
        }

        String lastName = request.getParameter("lastName");
        if ((lastName != null) && (!lastName.trim().equals(""))) {
            nameBean.setLastName(lastName);
        }

        String address = "/WEB-INF/mvc-sharing/ShowName.jsp";
        RequestDispatcher dispatcher =
            request.getRequestDispatcher(address);
        dispatcher.forward(request, response);
    }
}
```
Listing 15.13  ShowName.jsp

```html
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title>Thanks for Registering</title>
<link rel=stylesheet href="/bank-support/JSP-Styles.css" type="text/css">
</head>
<body>
<h1>Thanks for Registering</h1>
<jsp:useBean id="nameBean" type="coreservlets.NameBean" scope="session"/>
<h2>First Name: <jsp:getProperty name="nameBean" property="firstName" /></h2>
<h2>Last Name: <jsp:getProperty name="nameBean" property="lastName" /></h2>
</body>
</html>
```

Figure 15–7  Result of RegistrationServlet when one parameter is missing and no session data is found.
15.6 Comparing the Three Data-Sharing Approaches

Figure 15–8 Result of RegistrationServlet when one parameter is missing and session data is found.

**Application-Based Sharing**

In this example, our goal is to display a prime number of a specified length. If the user fails to tell us the desired length, we want to use whatever prime number we most recently computed for *any* user. Data is shared among multiple clients, so application-based sharing is appropriate.

To implement this behavior, we need a bean to store prime numbers (Listing 15.14, which uses the Primes class presented earlier in Section 7.4), a servlet to populate the bean and store it in the ServletContext (Listing 15.15), and a JSP page to display the results (Listing 15.16, Figures 15–9 and 15–10).

```
Listing 15.14 PrimeBean.java

package coreservlets;
import java.math.BigInteger;
public class PrimeBean {
    private BigInteger prime;
    public PrimeBean(String lengthString) {
        int length = 150;
        try {
            length = Integer.parseInt(lengthString);
        } catch (NumberFormatException nfe) {} 
        setPrime(Primes.nextPrime(Primes.random(length)));
    }
```
Chapter 15 ■ Integrating Servlets and JSP: The MVC Architecture

Listing 15.14  PrimeBean.java (continued)

```java
public BigInteger getPrime() {
    return(prime);
}

public void setPrime(BigInteger newPrime) {
    prime = newPrime;
}
```

Listing 15.15  PrimeServlet.java

```java
package coreservlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class PrimeServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                              throws ServletException, IOException {
        String length = request.getParameter("primeLength");
        ServletContext context = getServletContext();
        synchronized(this) {
            if ((context.getAttribute("primeBean") == null) ||
                (length != null)) {
                PrimeBean primeBean = new PrimeBean(length);
                context.setAttribute("primeBean", primeBean);
            }
            String address = "/WEB-INF/mvc-sharing/ShowPrime.jsp";
            RequestDispatcher dispatcher =
                              request.getRequestDispatcher(address);
            dispatcher.forward(request, response);
        }
    }
}
```
15.6 Comparing the Three Data-Sharing Approaches

Listing 15.16 ShowPrime.jsp

```html
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title>A Prime Number</title>
<link rel=stylesheet href="/bank-support/JSP-Styles.css" type="text/css">
</head>
<body>
<h1>A Prime Number</h1>
<jsp:useBean id="primeBean" type="coreservlets.PrimeBean" scope="application" />
<jsp:getProperty name="primeBean" property="prime" />
</body></html>
```

**Figure 15–9** Result of PrimeServlet when an explicit prime size is given: a new prime of that size is computed.

**Figure 15–10** Result of PrimeServlet when no explicit prime size is given: the previous number is shown and no new prime is computed.
15.7 Forwarding Requests from JSP Pages

The most common request-forwarding scenario is one in which the request first goes to a servlet and the servlet forwards the request to a JSP page. The reason a servlet usually handles the original request is that checking request parameters and setting up beans requires a lot of programming, and it is more convenient to do this programming in a servlet than in a JSP document. The reason that the destination page is usually a JSP document is that JSP simplifies the process of creating the HTML content.

However, just because this is the usual approach doesn’t mean that it is the only way of doing things. It is certainly possible for the destination page to be a servlet. Similarly, it is quite possible for a JSP page to forward requests elsewhere. For example, a request might go to a JSP page that normally presents results of a certain type and that forwards the request elsewhere only when it receives unexpected values.

Sending requests to servlets instead of JSP pages requires no changes whatsoever in the use of the RequestDispatcher. However, there is special syntactic support for forwarding requests from JSP pages. In JSP, the jsp:forward action is simpler and easier to use than wrapping RequestDispatcher code in a scriptlet. This action takes the following form:

```jsp
<jsp:forward page="Relative URL" />
```

The page attribute is allowed to contain JSP expressions so that the destination can be computed at request time. For example, the following code sends about half the visitors to `http://host/examples/page1.jsp` and the others to `http://host/examples/page2.jsp`.

```jsp
<% String destination;  
    if (Math.random() > 0.5) {  
        destination = "*/examples/page1.jsp";  
    } else {  
        destination = "*/examples/page2.jsp";  
    }  
%>
<jsp:forward page="<%= destination %>" />
```

The jsp:forward action, like jsp:include, can make use of jsp:param elements to supply extra request parameters to the destination page. For details, see the discussion of jsp:include in Section 13.2.
15.8 Including Pages

The forward method of RequestDispatcher relies on the destination JSP page to generate the complete output. The servlet is not permitted to generate any output of its own.

An alternative to forward is include. With include, the servlet can combine its output with that of one or more JSP pages. More commonly, the servlet still relies on JSP pages to produce the output, but the servlet invokes different JSP pages to create different sections of the page. Does this sound familiar? It should: the include method of RequestDispatcher is the code that the jsp:include action (Section 13.1) invokes behind the scenes.

This approach is most common when your servlets create portal sites that let users specify where on the page they want various pieces of content to be displayed. Here is a representative example.

```java
String firstTable, secondTable, thirdTable;
if (someCondition) {
    firstTable = "WEB-INF/Sports-Scores.jsp";
    secondTable = "WEB-INF/Stock-Prices.jsp";
    thirdTable = "WEB-INF/Weather.jsp";
} else if (...) { ... }
RequestDispatcher dispatcher =
    request.getRequestDispatcher("WEB-INF/Header.jsp");
dispatcher.include(request, response);
dispatcher =
    request.getRequestDispatcher(firstTable);
dispatcher.include(request, response);
dispatcher =
    request.getRequestDispatcher(secondTable);
dispatcher.include(request, response);
dispatcher =
    request.getRequestDispatcher(thirdTable);
dispatcher.include(request, response);
dispatcher =
    request.getRequestDispatcher("WEB-INF/Footer.jsp");
dispatcher.include(request, response);
```
CHAPTER 3

A MODEL 2 FRAMEWORK

Excerpted from: Advanced JavaServer Pages Advanced JavaServer Pages by David Geary
A MODEL 2 FRAMEWORK

Topics in this Chapter

- A Model 2 Framework
  - The Action Interface
  - The Action Factory
  - Action Routers
  - The Action Servlet
- Refining the Design
- Adding Use Cases
- The Importance of Custom Tags
- JSP Scripts
Because the Model 2 architecture is a Model-View-Controller implementation and because it allows for a division of labor between page authors and software developers, it’s an excellent choice for developing web applications. But we can do better by encapsulating general code in a framework, thereby significantly reducing the amount of code required to implement applications.

This chapter discusses a simple Model 2 framework that simplifies web application development and lets software developers and page authors work independently. This framework is similar in concept to the Apache Struts framework; see http://www.apache.org for more information about Struts.

This chapter retrofits the Model 2 application discussed in “A Model 2 Example” on page 137 to the framework introduced in this chapter; therefore, that discussion is a prerequisite for this chapter.

A Model 2 Framework

The framework discussed in this chapter uses a single servlet that acts as a controller. That servlet is known as the action servlet. All HTTP requests ending in .do are handled by the action servlet, which dispatches requests to Java beans, known as actions, as shown in Figure 6-1.
Action beans update business objects and return an action router to the action servlet. The action servlet uses that router to forward or redirect requests to a JSP page. That JSP page subsequently accesses business objects—often with custom tags—and sends a response back to the browser.

The framework shown in Figure 6-1 contains four types of objects, summarized in Table 6-1.

Table 6-1 Model 2 Framework Classes and Interfaces

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>Application-specific actions implement this interface</td>
</tr>
<tr>
<td>ActionFactory</td>
<td>Creates action instances</td>
</tr>
<tr>
<td>ActionServlet</td>
<td>Maps requests to actions</td>
</tr>
<tr>
<td>ActionRouter</td>
<td>Forwards or redirects requests to JSP pages</td>
</tr>
</tbody>
</table>

1. *italics* indicates an interface.

Figure 6-2 shows how the types of objects listed in Table 6-1 work together.
The action servlet is typically invoked from a JSP page or another servlet, as the result of a form submission or link activation. That servlet retrieves the appropriate type of action, depending upon the request, from the action factory.

Because typical web applications handle numerous requests, the action factory maintains one action instance for each action type, and those actions are reused for a given type of request. Reusing a single action for multiple requests greatly reduces the number of actions that the framework must instantiate.

Keep in mind that multiple threads can issue identical requests concurrently, meaning that a single action instance can be accessed concurrently; therefore, actions must be thread safe. The easiest way to implement thread-safe actions is to avoid maintaining intrinsic state. That is accomplished by the use of local variables instead of class members, because local variables can only be accessed by one thread at a time.

After the action servlet obtains an action from the action factory, it invokes the action’s `perform` method, which implements application-specific functionality, typically updating business objects. `Action.perform` returns an action router that maintains a URI and a `boolean` variable indicating whether the request should be forwarded or redirected to that URI.
With an action router in hand, the action servlet invokes the router’s `route` method, which forwards or redirects the request to the appropriate web component. That web component is typically a JSP page, an HTML page, or another servlet, and that component usually contains a form or a link whose submission or activation, respectively, issues a request. Thus, the cycle begins anew.

The rest of this section begins with a discussion of the classes listed in Table 6-1 on page 156. Subsequently, we’ll look at modifications to the Model 2 example discussed in “A Model 2 Example” on page 137 that are necessary to retrofit that example to our framework. Finally, we’ll explore some refinements to our framework’s design.

The Action Interface

The `Action` interface is listed in Example 6-1.a.

**Example 6-1.a**  /WEB-INF/classes/actions/Action.java

```java
package actions;

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

// Application-specific actions implement this interface

public interface Action {
    public ActionRouter perform(HttpServlet servlet,
                                 HttpServletRequest req,
                                 HttpServletResponse res)
                          throws java.io.IOException,
                                  javax.servlet.ServletException,
                                  java.io.IOException,
                                  javax.servlet.ServletException;
}
```

The `Action` interface defines a single method—`perform`—that’s passed references to the action servlet and the HTTP request and response.
The Action Factory

The `ActionFactory` class is listed in Example 6-1.b.

**Example 6-1.b /WEB-INF/classes/actions/ActionFactory.java**

```java
package actions;
import java.util.Hashtable;

public class ActionFactory {
    private Hashtable actions = new Hashtable();

    // This method is called by the action servlet
    public Action getAction(String classname,
                               ClassLoader loader)
        throws ClassNotFoundException,
               IllegalAccessException,
               InstantiationException {
        Action action = (Action)actions.get(classname);

        if(action == null) {
            Class klass = loader.loadClass(classname);
            action = (Action)klass.newInstance();
            actions.put(classname, action);
        }
        return action;
    }
}
```

The action factory maintains a hash table of actions, and the factory’s `getAction` method, invoked by the action servlet, returns an action corresponding to a specific class name. If that action is not stored in the factory’s hash table, the factory creates it and stores it in its hash table. The factory handles subsequent requests for the same action by returning a reference from the hash table.
Action Routers

The `ActionRouter` class is listed in Example 6-1.c.

**Example 6-1.c  /WEB-INF/classes/actions/ActionRouter.java**

```java
package actions;

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

// Action routers are immutable

public class ActionRouter {
    private final String url;
    private final boolean isForward;

    public ActionRouter(String url) {
        this(url, true); // forward by default
    }
    public ActionRouter(String url, boolean isForward) {
        this.url = url;
        this.isForward = isForward;
    }

    // This method is called by the action servlet
    public void route(GenericServlet servlet,
                      HttpServletRequest req,
                      HttpServletResponse res)
        throws javax.servlet.ServletException,
               java.io.IOException {
        if(isForward) {
            servlet.getServletContext().getRequestDispatcher(
                res.encodeURL(url)).forward(req, res);
        } else {
            res.sendRedirect(res.encodeRedirectURL(url));
        }
    }
}
```

Action routers forward or redirect requests; how those requests are handled can only be specified when a router is constructed.
The Action Servlet

A deployment descriptor maps URLs that end in `.do` to the action servlet, like this:

```
// From /WEB-INF/web.xml
...
<web-app>
  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>ActionServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>

...
</web-app>
```

The `.do` suffix is easy to remember because it indicates that the application is about to do something. Immediately preceding the `.do` suffix is the name of an action; for example, for a login action class named `LoginAction` from the `actions` package, the URL for the login action would be specified as follows:

```
<form action='<%= response.encodeURL("actions.LoginAction.do") %>'
   method='post'>
...
</form>
```

Let’s see how the action servlet, listed in Example 6-1.d, maps URLs into action classes.

**Example 6-1.d**  
/WEB-INF/classes/ActionServlet.java

```java
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import actions.Action;
import actions.ActionFactory;
import actions.ActionRouter;

public class ActionServlet extends HttpServlet {
    private ActionFactory factory = new ActionFactory();
```
public void service(HttpServletRequest req,
HttpServletResponse res)
    throws java.io.IOException,
javax.servlet.ServletException {
    try {
        Action action = factory.getAction(getClassname(req),
            getClass().getClassLoader());
        ActionRouter router = action.perform(this, req, res);
        router.route(this, req, res);
    }
    catch(Exception e) {
        throw new ServletException(e);
    }
}

private String getClassname(HttpServletRequest req) {
    String path = req.getServletPath();
    int slash = path.lastIndexOf("/"),
        period = path.lastIndexOf(".");

    if(period > 0 && period > slash)
        path = path.substring(slash+1, period);

    return path;
}

The ActionServlet.service implementation is straightforward: An action is obtained from the action factory, and its perform method is invoked. The action’s perform method returns an action router, which routes the request.

ActionServlet.getClassname is responsible for mapping URLs to action class names. That method obtains a reference to the servlet path and subsequently extracts the string between the forward slash and the period. For the login URL discussed in “The Action Servlet” on page 161, the servlet path is /actions>LoginAction.do, and the class name is actions.LoginAction.

Retrofitting the Original Model 2 Example

This section retrofits the login and registration example from the previous chapter: “A Model 2 Example” on page 137 to this chapter’s Model 2 framework. Figure 6-3 shows the directory structure and files for the retrofitted application—Figure 5-3 on page 138 is a similar diagram for the original example.
Once the framework is in place—meaning the action servlet is in /WEBINF/classes and the supporting action classes are in /WEBINF/classes/actions—only two changes need to be made to the original application. First, servlet URLs are replaced in login.jsp and newAccount.jsp; the URLs used in the original JSP files looked like this:

```jsp
<%-- this is from the original login.jsp --%>
<form action='<%= response.encodeURL("login") %>' method='post'>
<%-- this is from the original newAccount.jsp --%>
<form action='<%= response.encodeURL("new-account") %>' method='post'>

The modified URLs look like this:

```jsp
<%-- this is from the new login.jsp --%>
<form action='<%= response.encodeURL("actions.LoginAction.do") %>' method='post'>
<%-- this is from the new newAccount.jsp --%>
<form action='<%= response.encodeURL("actions.NewAccountAction.do") %>' method='post'>
Second, the original application’s login and new account servlets are rewritten as actions. Those actions are listed in Example 6-2.a and Example 6-2.b, respectively.

**Example 6-2.a** /WEB-INF/classes/actions/LoginAction.java

```java
package actions;

import javax.servlet.*;
import javax.servlet.http.*;
import beans.LoginDB;
import beans.User;

public class LoginAction implements Action {
    public ActionRouter perform(HttpServlet servlet,
                                 HttpServletRequest req, HttpServletResponse res)
                               throws java.io.IOException, ServletException {
        LoginDB loginDB = getLoginDB(servlet.getServletContext());
        User user = loginDB.getUser(req.getParameter("userName"),
                                     req.getParameter("password"));

        if(user != null) { // user is in the login database
            req.getSession().setAttribute("user", user);
            return new ActionRouter("/welcome.jsp");
        }
        else
            return new ActionRouter("/loginFailed.jsp");
    }

    private LoginDB getLoginDB(ServletContext context) {
        LoginDB loginDB = (LoginDB)context.getAttribute("loginDB");
        if(loginDB == null)
            context.setAttribute("loginDB", loginDB = new LoginDB());

        return loginDB;
    }
}
```

**Example 6-2.b** /WEB-INF/classes/actions/NewAccountAction.java

```java
package actions;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import beans.LoginDB;
import beans.User;
```
public class NewAccountAction implements Action {
    public ActionRouter perform(HttpServlet servlet,
        HttpServletRequest req,
        HttpServletResponse res)
        throws java.io.IOException,
            javax.servlet.ServletException{
            LoginDB loginDB = (LoginDB)servlet.getServletContext().
            getAttribute("loginDB");
            String uname = req.getParameter("userName");
            loginDB.addUser(uname, req.getParameter("password"),
                req.getParameter("password-hint"));
            req.setAttribute("userName", uname);
            return new ActionRouter("/accountCreated.jsp");
        }
    }

Both actions are similar to their servlet counterparts, listed in Example 5-1.f on page 145 and Example 5-1.j on page 151, except the actions return action routers.

Refining the Design

Like any design, the Model 2 architecture discussed in “A Model 2 Framework” on page 155 has plenty of room for improvement; for example, JSP pages must explicitly refer to action classes, as illustrated by the form tag used by login-form.jsp:

    <form action='<%= response.encodeURL("actions.LoginAction.do") %>'
        method='post'>

And actions must explicitly refer to JSP pages, as the following excerpt from LoginAction.java illustrates:

    if(user != null) { // user is in the login database
        req.getSession().setAttribute("user", user);
        return new ActionRouter("/welcome.jsp");
    }
    else
    return new ActionRouter("/loginFailed.jsp");

This coupling between JSP pages and actions is undesirable, because changing the name of a JSP file results in a change to an action, and changing the name of an action—or even moving an action to a different package—results in a change to a JSP page.
Ideally, JSP pages and actions should be decoupled with logical names; for example, JSP pages could use logical names, like this:

```html
<form action='<%= response.encodeURL("login-action") %>'
    method='post'>
</form>
```

Actions could also use logical names, like this:

```java
if(user != null) { // user is in the login database
    req.getSession().setAttribute("user", user);
    return new ActionRouter("welcome-page");
} else
    return new ActionRouter("login-failed-page");
```

We can easily use a resource bundle\(^1\) to map logical names to action classes and JSP pages. First, we define a properties file, listed in Example 6-3.a, that resides in `/WEB-INF/classes`:

**Example 6-3.a /WEB-INF/classes/actions.properties**

```properties
# Action mappings used by ActionServlet
login-action=actions.LoginAction
new-account-action=actions.NewAccountAction

# JSP mappings used by Routers
login-failed-page=/loginFailed.jsp
welcome-page=/welcome.jsp
account-created-page=/accountCreated.jsp
```

We then add an `init` method to the action servlet, listed in Example 6-3.b, that creates a resource bundle from that properties file. That resource bundle is stored in application scope, for subsequent access by the action servlet and action routers.

**Example 6-3.b /WEB-INF/classes/ActionServlet.java**

```java
import java.util.MissingResourceException;
import java.util.ResourceBundle;

import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;

1. See “Resource Bundles” on page 213 for more information on resource bundles.
```
import javax.servlet.http.HttpServletResponse;

import actions.Action;
import actions.ActionFactory;
import actions.ActionRouter;

public class ActionServlet extends HttpServlet {
    private ActionFactory factory = new ActionFactory();

    public void init(ServletConfig config) throws ServletException{
        super.init(config);
        ResourceBundle bundle = null;
        try {
            bundle = ResourceBundle.getBundle("actions");
        } catch(MissingResourceException e) {
            throw new ServletException(e);
        }
        getServletContext().setAttribute("action-mappings", bundle);
    }

    public void service(HttpServletRequest req, HttpServletResponse res) throws java.io.IOException, ServletException {
        try {
            String actionClass = getActionClass(req);
            Action action = factory.getAction(actionClass, getClass().getClassLoader());
            ActionRouter router = action.perform(this, req, res);
            router.route(this, req, res);
        } catch(Exception e) {
            throw new ServletException(e);
        }
    }

    private String getClassname(HttpServletRequest req) {
        String path = req.getServletPath();
        int slash = path.lastIndexOf('/'), period = path.lastIndexOf('.');
        if(period > 0 && period > slash)
            path = path.substring(slash+1, period);
        return path;
    }

    private String getActionClass(HttpServletRequest req) {
        ResourceBundle bundle = (ResourceBundle)getServletContext().
getAttribute("action-mappings");

return (String)bundle.getObject(getActionKey(req));

private String getActionKey(HttpServletRequest req) {
    String path = req.getServletPath();
    int slash = path.lastIndexOf("/"),
               period = path.lastIndexOf(".");

    if(period > 0 && period > slash)
        path = path.substring(slash+1, period);

    return path;
}

The action servlet’s service method obtains the action class name through
getActionClass, which uses the resource bundle to map logical names to
action class names.

The ActionRouter class, listed in Example 6-3.c, is also modified to map logical
names to JSP pages.

Example 6-3.c /WEB-INF/classes/actions/ActionRouter.java

package actions;

import java.util.ResourceBundle;
import javax.servlet.GenericServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

// Action routers are immutable

public class ActionRouter {
    private final String key;
    private final boolean isForward;

    public ActionRouter(String key) {
        this(key, true); // forward by default
    }

    public ActionRouter(String key, boolean isForward) {
        this.key = key;
        this.isForward = isForward;
    }
}
// This method is called by the action servlet

public synchronized void route(GenericServlet servlet,
    HttpServletRequest req,
    HttpServletResponse res)
    throws java.io.IOException,
    javax.servlet.ServletException {
    ResourceBundle bundle = (ResourceBundle)servlet.
    getServletContext().
    getAttribute("action-mappings");
    String url = (String)bundle.getObject(key);
    if(isForward) {
        servlet.getServletContext().getRequestDispatcher(
            res.encodeURL(url)).forward(req, res);
    } else {
        res.sendRedirect(res.encodeRedirectURL(url));
    }
}

Now that the action servlet and action routers can map logical names, actions and JSP pages can use those names; for example, Example 6-3.d lists the NewAccountAction class and Example 6-3.e lists login-form.jsp.

Example 6-3.d /WEB-INF/classes/actions/NewAccountAction.java

```java
public class NewAccountAction implements Action {
    public ActionRouter perform(HttpServlet servlet,
        HttpServletRequest req, HttpServletResponse res)
        throws java.io.IOException, ServletException {
    LoginDB loginDB = (LoginDB)servlet.getServletContext().
    getAttribute("loginDB");
    String uname = req.getParameter("userName");
    loginDB.addUser(uname, req.getParameter("password"),
        req.getParameter("password-hint"));
    req.setAttribute("userName", uname);
    return new ActionRouter("account-created-page");
    }
```
Example 6-3.e /login-form.jsp

```jsp
<%@ taglib uri='utilities' prefix='util' %>

<font size='5' color='blue'>Please Login</font><hr>
<form action='<%= response.encodeURL("login-action.do") %>'
    method='post'>
    ...
</form>

Many other improvements can be made to this framework; for example, the action servlet listed in Example 6-3.b on page 166 hardcodes the name of the properties file; if that name changes, the action servlet must be modified and recompiled. That dependency can be eliminated with a servlet initialization parameter.

The deployment descriptor specifies that initialization parameter:

```jsp
<web-app>
    <servlet>
        <servlet-name>action</servlet-name>
        <servlet-class>ActionServlet</servlet-class>
        <init-param>
            <param-name>action-mappings</param-name>
            <param-value>actions</param-value>
        </init-param>
    </servlet>
    ...
</web-app>
```

The action servlet undergoes a simple modification, listed below, to use the initialization parameter:

```java
public class ActionServlet extends HttpServlet {
    ...
    public void init(ServletConfig config) throws ServletException{
        super.init(config);

        ResourceBundle bundle = null;
        try {
            bundle = ResourceBundle.getBundle(
                config.getInitParameter("action-mappings"));
        }
        catch(MissingResourceException e) {
            throw new ServletException(e);
        }
        ...
    }
}
```
Using an initialization parameter eliminates the dependency between the properties file and the action servlet but creates a dependency between the properties file and the deployment descriptor. Because changes to deployment descriptors do not involve code changes or recompilation, the latter dependency is preferred over the former.

Logical names can also be mapped with XML files. Because XML is widely used for configuration, in this case it’s probably preferable to a properties file; however, we opted for a properties file because it’s a simpler solution. See “XML” on page 330 for more information about XML and JSP.

Adding Use Cases

One measure of a framework’s usefulness is the ease with which applications can be modified or extended. For the framework discussed in this chapter, adding use cases to an application involves the following steps:

1. **Implement an action** that manipulates business objects (the model) and perhaps stores beans in an appropriate scope for a JSP page (the view) to access. *(software developers)*

2. **Implement a JSP page** that accesses business objects (that may have been modified in step #1) and beans from a specific scope (that may have been created in step #1). *(web page authors)*

3. **Add mappings** to the application’s properties file that equate the action and JSP page from steps #1 and #2 to logical names. *(web page authors or software developers)*

Notice the division of labor in the steps listed above: Software developers are responsible for actions, and web page authors are responsible for JSP pages.

Depending upon the use case, additional steps may be required, for example, modify existing actions *(software developers)* and JSP pages *(web page authors)*, or implement custom tags *(software developers)* and use custom tags *(web page authors)*. But for many use cases, the steps listed above will suffice.

This section extends the web application discussed in “A Model 2 Example” on page 137 and “Retrofitting the Original Model 2 Example” on page 162 with a password hint use case. That use case can be described as follows:

1. Login fails because of an incorrect password for an existing user.
2. The action servlet forwards to the login failed page, which provides a link to a password hint action.
3. The user activates the password hint link, which submits a request, through the action servlet, to the password hint action.
The password hint action retrieves the hint for the specified user from the login database, and stores the username and hint in request scope.

The action servlet forwards to a JSP page that displays the username and hint stored in request scope. That JSP page includes the login form so the user can retry login.

Figure 6-4 shows the login failed and password hint JSP pages.

Let’s look at the steps involved in adding this use case to our application.

Step #1: Implement a Password Hint Action

The password hint action—ShowHintAction—is listed in Example 6-4.a.

Example 6-4.a  /WEB-INF/classes/actions/ShowHintAction.java

```java
package actions;

import javax.servlet.ServletContext;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import beans.LoginDB;

public class ShowHintAction implements Action {
```
The action listed in Example 6-4.a implements use case step 4 listed above by storing the username and password hint in request scope. A logical name—show-hint-page—is specified for the router returned from the perform method. That logical name is mapped to the show hint JSP page in the application’s properties file.

Step #2: Implement a Password Hint JSP Page

The password hint JSP page—showHintAction.jsp—is listed in Example 6-4.b.

Example 6-4.b /showHintAction.jsp

```html
<html><title>Password Hint</title>
<body>

  Hint for <b><%= request.getAttribute("userName") %></b> is <i><%= request.getAttribute("hint") %></i> 

  <p><font size='5' color='blue'>Please Login</font></p><hr>
  <form action='<%= response.encodeURL("login-action.do") %>' method='post'>
    <table>
      <tr>
        <td>
          <input type="text" name="username"
```
The JSP page listed in Example 6-4.a implements use case step 5 listed above by retrieving the username and hint from request scope and displaying them to the user.

The login form’s action is specified as a logical name—login-action—which, like the logical name used by the show hint action, is defined in the application’s properties file.

Step #3: Add Mappings to the Properties File

Example 6-4.c lists the application’s properties file.

Example 6-4.c  /WEB-INF/classes/actions.properties

```
# Action mappings used by ActionServlet
login-action=actions.LoginAction
new-account-action=actions.NewAccountAction
show-hint-action=actions.ShowHintAction

# JSP mappings used by Routers
login-failed-page=/loginFailed.jsp
welcome-page=/welcome.jsp
account-created-page=/accountCreated.jsp
show-hint-page=/showHint.jsp
```
Step #4: Modify the Login Failed JSP Page

The login failed page is modified, as listed in Example 6-4.d, to provide different links depending upon whether a password hint exists for the username specified in the login form.

Example 6-4.d /loginFailed.jsp

```html
<html><head><title>Login Failed</title></head>
<body>

<font color='red' size='5'>Login Failed</font>
<font color='red' size='4'><p>
Please enter a valid username and password, or create a new account</p></font></p>

<%@ include file='/login-form.jsp' %>

<jsp:useBean id='loginDB' class='beans.LoginDB'
scope='application'/>

<%if(loginDB.getHint(request.getParameter("userName")) != null){%>
Click <a href='<%=response.encodeURL("show-hint-action.do")%>'>here</a> to see your password hint.
<% } else { %>
Click <a href='<%=response.encodeURL("newAccount.jsp")%>'>here</a> to open a new account.
<% } %>

</body>
</html>
```

The JSP page listed above accesses the login database to determine whether a password hint exists for the specified user. If so, a link is provided to the show hint action; if not, a link to the new account JSP page is provided.

The Importance of Custom Tags

One of the main benefits of the Model 2 architecture is that Java code is encapsulated in servlets—see “Model 2 Encapsulates Java Code in Servlets” on page 147. This encapsulation of Java code is critical for large web development projects, where web page authors code HTML and JSP pages, and software
developers provide underlying functionality. Keeping JSP pages free of Java code allows web page authors and software developers to work in parallel with few dependencies.

Even with a Model 2 framework, such as the one discussed in this chapter, it’s not uncommon for some Java code to creep into JSP pages. For example, the login failed JSP page listed in Example 6-4.d contains a small amount of Java code, listed below.

```html
<html><head><title>Login Failed</title></head>
...
<jsp:useBean id='loginDB' class='beans/LoginDB' scope='application'/>
...
<%if(loginDB.getHint(request.getParameter("userName")) != null){%>
  Click <a href='<%=response.encodeURL("show-hint-action.do")%>'>here</a> to see your password hint.
<% } else { %>
  Click <a href='<%=response.encodeURL("newAccount.jsp")%>'>here</a> to open a new account.
<% } %>
...
```

Application-specific custom tags can replace Java code in JSP pages; for example, the login failed page listed below uses two custom tags to replace Java code.

```html
<html><head><title>Login Failed</title></head>
...
<%@ taglib uri='application-tags' prefix='app' %>
...
<app:hintAvailable>
  Click <a href='<%=response.encodeURL("show-hint-action.do")%>'>here</a> to see your password hint.
</app:hintAvailable>
<app:hintNotAvailable>
  Click <a href='<%=response.encodeURL("login.jsp")%>'>here</a> to retry login.
</app:hintNotAvailable>
...
```

If a hint is available, the body of the `hintAvailable` tag is included in the generated HTML; otherwise, the body of the `hintNotAvailable` tag is included.

Most custom tags, such as those used above, are easy to implement. For example, Example 6-5.a lists the hint available tag, and Example 6-5.b lists the hint not available tag.
Example 6-5.a /WEB-INF/classes/tags/HintAvailableTag.java

```
package tags;

import javax.servlet.ServletRequest;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.TagSupport;
import beans.LoginDB;

public class HintAvailableTag extends TagSupport {
    public int doStartTag() throws JspException {
        ServletRequest req = pageContext.getRequest();
        LoginDB loginDB = (LoginDB)pageContext.
                     findAttribute("loginDB");
        if(loginDB.getHint(req.getParameter("userName")) != null)
            return EVAL_BODY_INCLUDE;
        else
            return SKIP_BODY;
    }
}
```

Example 6-5.b /WEB-INF/classes/tags/HintNotAvailableTag.java

```
package tags;

import javax.servlet.jsp.JspException;

public class HintNotAvailableTag extends HintAvailableTag {
    public int doStartTag() throws JspException {
        int available = super.doStartTag();
        return available == EVAL_BODY_INCLUDE ? SKIP_BODY : EVAL_BODY_INCLUDE;
    }
}
```

Both tags listed above access the login database to determine whether a hint is available for a specified username. Subsequently, those tags return either SKIP_BODY or EVAL_BODY_INCLUDE, which determines whether the body of the tag is included in the generated HTML. See “Custom Tag Fundamentals” on page 2 and “Custom Tag Advanced Concepts” on page 32 for more information on implementing custom tags.

JSP custom tags neatly encapsulate functionality for web page authors. One of the most important uses of custom tags is eliminating Java code from JSP pages. This use of custom tags is one reason why they are the single most important JSP feature.
JSP Scripts

Conventional wisdom advocates eliminating Java code from JSP pages. For the most part, that’s excellent advice, especially when JSP pages are used as views in an MVC architecture. But there are times when JSP pages crammed with Java can be quite useful; those types of JSP pages are referred to in this book as JSP scripts. Fortunately, JSP is flexible enough to support many different design philosophies.

JSP scripts are JSP pages that contain mostly Java sprinkled with HTML. Like custom tags, JSP scripts encapsulate functionality useful to web page authors; for example, consider the script listed in Example 6-6.a that prints request parameters, something that’s useful for debugging.

Example 6-6.a  /showRequestParameters.jsp: A JSP Script

```jsp
<% java.util.Enumeration e = request.getParameterNames(); boolean hasParams = false;

while(e.hasMoreElements()) {
    String name = (String)e.nextElement();
    String[] values = request.getParameterValues(name);

    hasParams = true;

    for(int i=0; i < values.length; ++i) {
        String next = values[i];
        if(i == 0) { <b><%= name %>:</b> <%= next %>
        } else { <i>, <%= next %>
        }</i>
    }
}<%
if(!hasParams) { <i>No parameters with this request</i>
<% }
```

Figure 6-5 shows a JSP page that uses the script listed in Example 6-6.a.
The top picture shown in Example 6-6.a is an HTML page, listed in Example 6-6.b.

Example 6-6.b  test.html

```html
<html><title>JSP Scripts</title>
<body>

<form action='showParams.jsp'>
  <font size='4' color='blue'>Select Your Age Bracket:</font>
  
  <input type='radio' name='age' value='1-10'>1-10
  <input type='radio' name='age' value='11-20'>11-20
  <input type='radio' name='age' value='21-30'>21-30
  <input type='radio' name='age' value='31-40'>31-40
  <input type='radio' name='age' value='41-50'>41-50
  
  <p>
  <font size='4' color='blue'>Select Your Favorite Fruits:</font>
  
  <input type='checkbox' name='fruit' value='Kiwi'>Kiwi
  <input type='checkbox' name='fruit' value='Apple'>Apple
  <input type='checkbox' name='fruit' value='Pear'>Pear

</form>
```

Figure 6-5 Using a JSP Script
Advanced JavaServer Pages

<input type='checkbox' name='fruit' value='Grape'>Grape</input>
</p>
<p><input type='submit'/></p>
</form>

</body>
</html>

In Example 6-6.b, showParams.jsp is specified as the form’s action; that file is listed in Example 6-6.c.

Example 6-6.c /showParams.jsp

<html><title>JSP Scripts</title>
<body>

<font size='4' color='blue'>Request Parameters:</font>
<%@ include file='showRequestParameters.jsp' %>

</body>
</html>

JSP scripts are included by another JSP page, as is the case for the JSP page listed in Example 6-6.c.

Both JSP scripts and custom tags encapsulate functionality useful to web page authors. Custom tags are more difficult to develop because they require coding and compilation, in addition to a tag library descriptor definition. In contrast, although JSP scripts are easier to develop, they’re not as natural as custom tags for web page authors because they require the use of the JSP include directive. All other things being equal, custom tags are more reusable than JSP scripts, but the latter can be useful tools in the JSP developer’s toolchest.

Conclusion

Some would have you believe that JSP is not viable for developing complex Web applications because it’s too easy to mix Java code with HTML in JSP pages.2 I disagree. Although I would not want to develop a large JSP-based web application without a Model 2 architecture, I would be more than comfortable using the techniques discussed in this chapter to implement any application.

This chapter provides insights into object-oriented design, the Model-View-Controller architecture and use cases, all of which are proven techniques for developing complex software systems. Books and articles on those topics abound; here is a short list of good references:

**Object-Oriented Design:**

**Model-View-Controller:**

**Use Cases:**
A Case Study

Topics in this Chapter

- The Fruitstand
  - The Homepage
  - The Storefront
  - The Checkout
  - The Purchase
- The Model 2 Framework
  - The Model
  - The Views—JSP Pages and Templates
  - The Controllers—Servlets and Actions
- Internationalization
- Authentication
- Sensitive Form Resubmissions
- SSL
- XML and DOM
This book has focused on singular techniques for implementing web applications, such as Model 2 frameworks, internationalization, and authentication. This chapter shows you how to use many of those techniques to implement a nontrivial Web application—an online fruitstand. That fruitstand:

- Is an e-commerce application with inventory, users, and shopping carts
- Is implemented with a Model 2 MVC architecture
- Is internationalized in three languages: English, German, and Chinese
- Implements custom authentication
- Uses JSP templates
- Accesses a database
- Uses XML and DOM
- Guards against sensitive form resubmissions

You can use this chapter in one of two ways:

- As a guide to developing nontrivial web applications using the concepts discussed in this book
- As an introduction to the concepts discussed in this book
Because of this chapter’s dual role, you can read—or most likely skim—this chapter first, even though it’s the last chapter in the book. If you do that, be aware that concepts are not explained in this chapter, so don’t expect all of the code to make sense at first glance.

This chapter uses many of the 50 or so custom tags discussed throughout this book. Those tags are useful, but the concepts behind those tags, and not the tags themselves, are the emphasis of this book.

The online fruitstand contains a significant amount of code, so we will cover it in three passes:

- **The main use case:** Homepage→Storefront→Checkout→Purchase, starting at “The Homepage” on page 395
- **The MVC architecture:** The Model, Views, and Controllers, starting at “The Model 2 Framework” on page 416
- **Other features:** I18n, authentication, HTML forms, sensitive form resubmissions, SSL, and XML, starting at “Internationalization” on page 438

Because the topics listed above are discussed extensively elsewhere in this book, this chapter is purposely short on words and long on code and diagrams.

**The Fruitstand**

The online fruitstand provides one-stop shopping for 10 different fruits, as shown in Figure 12-1. The fruitstand’s home page reveals its true intent by providing a summary of the JSP techniques used to develop the application. From the homepage, users can access the storefront, purchase fruit, and subsequently check out their purchase.

Figure 12-1 depicts a single use case for the fruitstand application; namely, a *logged-in user purchases fruit*. The fruitstand also implements a number of other use cases ranging from *user switches languages* to *user opens a new account*.

---

1. You can use this book’s custom tags for any purpose.
The use case shown in Figure 12-1 can be described as follows:

1. A user accesses the homepage and activates the Go Shopping button.
2. The application forwards to the storefront page. That page accesses inventory from a database, lets the user select items, and displays those items in the sidebar’s shopping cart.
3. The user activates the Checkout button in the storefront’s sidebar.
4. The user has already logged in\(^2\) so the application forwards to the checkout page. That page displays an invoice of the items in the user’s shopping cart.

5. The user activates the Purchase the Items Listed Above button in the checkout page.

6. The application forwards to a purchase page, which thanks the user for the purchase, and displays the expected ship date.

We will discuss the implementation of the use case listed above, starting with “The Homepage” on page 395. But first, take a look at Figure 12-2, which shows an overview of the fruitstand’s directory structure.

Except for graphics and a couple of files in the top-level directory, the fruitstand’s files all reside under `/WEB-INF`. According to the servlet specification, files under `/WEB-INF` must not be directly accessed, so nearly all of the fruitstand application is hidden from direct access by a browser.

2. See “Authentication” on page 443 for a discussion of the create account case.
The fruitstand application uses the Model 2 framework discussed in "A Model 2 Framework" on page 154. That framework is a Model-View-Controller (MVC) implementation that allows applications to be built from interchangeable parts. The fruitstand’s model consists of a database and beans, the views are JSP pages, and the controllers—which orchestrate use cases—are servlets and actions.

The fruitstand’s JSP files all reside under /WEB-INF/jsp, one JSP page per directory; for example, there’s a /WEB-INF/jsp/homepage directory that contains the JSP files used by the home page.

The fruitstand application was developed and tested with both Resin 1.2.1 and Tomcat 3.2 final. You can run the application in one of two ways. The easiest way is to create a JAR file of the application and place that JAR file in $TOMCAT_HOME/webapps or $RESIN_HOME/webapps, for Tomcat and Resin, respectively, where $TOMCAT_HOME and $RESIN_HOME are the installation directories for those servlet containers. If you want to modify the application, it’s best to specify that application in the Tomcat and Resin configuration files; for example, for Tomcat 3.2 final, you would add the following to $TOMCAT_HOME/conf/server.xml:

```xml
<Context path="/case-study"
  docBase="f:/books/jsp/src/case_study"/>
```

For Resin, you would add the following to $RESIN_HOME/conf/resin.conf:

```xml
<web-app id='case-study'
  app-dir='f:/books/jsp/src/case_study/final'/>
```

Now that we have a high-level view of how the fruitstand application is organized, let’s explore the implementation of the logged-in user purchases fruit use case.

The Homepage

The fruitstand application defines a welcome file list containing one file: /index.jsp. That welcome file designation, from /WEB-INF/web.xml, is listed below:

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">
...
```
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
</welcome-file-list>

/index.jsp is invoked when the following URL is accessed with either Tomcat
or Resin: http://localhost:8080/case-study. Example 12-1.a lists
/index.jsp.

Example 12-1.a  /index.jsp

```jsp
<%@ page contentType='text/html; charset=UTF-8' %>
<%@ taglib uri='regions' prefix='region' %>
<%@ include file='/WEB-INF/jsp/regionDefinitions.jsp' %>

<region:render region='HOMEPAGE_REGION'/>
```

Because the fruitstand supports Chinese, all of the application’s JSP pages use the
UTF-8 charset. See “Templates” on page 96 for more information concerning
UTF-8 and support for non-Western languages.

/index.jsp uses the regions custom tag library discussed in “Templates” on
page 96 and includes another JSP file—/WEB-INF/jsp/
regionDefinitions.jsp—that contains region definitions. Subsequently,
/index.jsp renders the homepage region. The homepage is shown in Figure 12-3.

Like all the other JSP pages in the fruitstand application, the homepage comprises
one region with four sections: header, sidebar, content, and footer. The homepage
sidebar contains three flags—used to select a language—and the Go Shopping
button. The header displays the Welcome to Fruitstand.com message and a
horizontal rule. The homepage content is the main text, and the footer contains a
horizontal rule and a greeting with the current date.

3. The port number, in this case 8080, may change with other servlet containers.
Each of the homepage sections is implemented with one or more JSP files, collectively known as a component. Every JSP page in this application is composed in this same way, with templates that insert interchangeable components. See “Templates” on page 96 for more information on templates and their benefits.

regionDefinitions.jsp, which defines all of the regions used in the fruitstand application, is partially listed in Example 12-1.b.4

4. See Example 12-7.g on page 428 for a complete listing of regionDefinitions.jsp.
Example 12-1.b  /WEB-INF/jsp/regionDefinitions.jsp (partial listing)

```java
<%@ taglib uri='regions' prefix='region' %>

<region:define id='STOREFRONT_REGION'
    template='/WEB-INF/jsp/templates/hscf.jsp'>
    <region:put section='title'
        content='FruitStand.com'
        direct='true'/>

    <region:put section='background'
        content='graphics/blueAndWhiteBackground.gif'
        direct='true'/>

    <region:put section='header'
        content='/WEB-INF/jsp/storefront/header.jsp'/>

    <region:put section='sidebar'
        content='/WEB-INF/jsp/storefront/sidebar.jsp'/>

    <region:put section='content'
        content='/WEB-INF/jsp/storefront/content.jsp'/>

    <region:put section='footer'
        content='/WEB-INF/jsp/storefront/footer.jsp'/>
</region:define>
...

<region:define id='HOMEPAGE_REGION' region='STOREFRONT_REGION'>
    <region:put section='sidebar'
        content='/WEB-INF/jsp/homepage/sidebar.jsp'/>

    <region:put section='content'
        content='/WEB-INF/jsp/homepage/content.jsp'/>
</region:define>
...

regionDefinitions.jsp defines a STOREFRONT_REGION for the application's storefront. That region is listed in Example 12-1.b because the HOMEPAGE_REGION extends it and overrides the sidebar and content sections. The homepage reuses the storefront's title, background, header, and footer sections.

Example 12-1.c and Example 12-1.d list the storefront header and footer JSP pages, respectively.
Example 12-1.c  /WEB-INF/jsp/storefront/header.jsp

```jsp
<%@ page contentType='text/html; charset=UTF-8' %>
<%@ taglib uri='i18n' prefix='i18n' %>

<font size='6' color='blue'>
    <i18n:message key="storefront.title"/>
</font>
<hr/>
<br/>
<br/>
```

Example 12-1.d  /WEB-INF/jsp/storefront/footer.jsp

```jsp
<%@ page contentType='text/html; charset=UTF-8' %>
<%@ taglib uri='i18n' prefix='i18n' %>

<br/>
<p>
<table>
<tr>
<td><img src='graphics/duke.gif'/></td>
<td>
<i18n:message key='login.footer.message'/><i>
<i18n:format date='<%=new java.util.Date()%>'
    dateStyle='<%=java.text.DateFormat.FULL%>'/>
</i>
</td>
</tr>
</table>
</p>
```

All of the text displayed in the fruitstand application is rendered by the `i18n:message` and `i18n:format` tags, which are used to internationalize text, numbers, dates, and currency. Those tags are used by the storefront’s header and footer.

The `i18n:message` tag displays text defined in a properties file; for example, the partial listing below is from `/WEB-INF/classes/app_en.properties`, which is the English properties file for the fruitstand application.

```properties
...  
storefront.title=Welcome to FruitStand.com
storefront.form.title=Please select from our fresh fruts.

storefront.table.header.picture=Picture
storefront.table.header.item=Item
storefront.table.header.description=Description
storefront.table.header.price=Price
storefront.table.header.addToCart=Add To Cart
...  
```
The storefront application also has properties files for German and Chinese; see “Internationalization” on page 438 for more information about the storefront’s internationalization capabilities.

The homepage content page is listed in Example 12-1.e.

Example 12-1.e /WEB-INF/jsp/homepage/content.jsp

```jsp
<%@ page contentType='text/html; charset=UTF-8' %>
<%@ taglib uri='i18n' prefix='i18n' %>

<font size='5' color='blue'>
  <i18n:message key='homepage.title'/>
</font>

<i18n:message key='homepage.text'/>
```

Although the text displayed in the homepage is rather lengthy, the JSP file that produces it is not. That’s because the JSP file listed above also uses the `i18n:message` tag to retrieve the rather lengthy text associated with `homepage.text`.

Going Shopping

There are only two interesting things to do from the fruitstand’s homepage: change languages by clicking on one of the flags, or go shopping by clicking on the Go Shopping button. Both the flags and the buttons are contained in the homepage’s sidebar, which is listed in Example 12-2.a.

Example 12-2.a /WEB-INF/jsp/homepage/sidebar.jsp

```jsp
<%@ page contentType='text/html; charset=UTF-8' %>
<jsp:include page='../shared/flags.jsp' flush='true'/>

<form action='go-shopping-action.do'>
  <input type='submit' value='Go Shopping'/>
</form>
```

The homepage sidebar includes another JSP file that displays the flags. That file—/WEB-INF/jsp/shared/flags.jsp—is used by all of the fruitstand’s JSP pages so that users can change languages at any time. See “Internationalization” on page 438 for more information about `flags.jsp`.

The homepage sidebar also contains a simple form with a submit button. The action associated with that button is `go-shopping-action.do`. URIs that end in `.do` are handled by the fruitstand’s action servlet. That servlet, which is part of
a simple Model 2 framework discussed in “A Model 2 Framework” on page 154, forwards the go-shopping-action.do request to an action, which is listed in Example 12-2.b.

Example 12-2.b  /WEB-INF/classes/actions/GoShoppingAction.java

```java
text
```

The GoShoppingAction.perform method, invoked by the action servlet, makes sure the user has a shopping cart in the session and returns an action router that the action servlet uses to forward the request to the storefront-page. That page is defined in the application’s actions.properties file, which is listed in Example 12-2.c.
Advanced JavaServer Pages

Example 12-2.c  /WEB-INF/classes/actions.properties

# Action mappings used by the action servlet

go-shopping-action = actions.GoShoppingAction
query-account-action = actions.QueryAccountAction
new-account-action = actions.NewAccountAction
show-hint-action = actions.ShowHintAction
update-locale-action = actions.UpdateLocaleAction
add-selection-to-cart-action = actions.AddToCartAction
checkout-action = actions.CheckoutAction
validate-account-action = actions.ValidateAccountAction
purchase-action = actions.PurchaseAction

# JSP mappings used by Routers

storefront-page = /WEB-INF/jsp/storefront/page.jsp
login-failed-page = /WEB-INF/jsp/loginFailed/page.jsp
query-account-page = /WEB-INF/jsp/createAccount/page.jsp
account-created-page = /WEB-INF/jsp/accountCreated/page.jsp
show-hint-page = /WEB-INF/jsp/showHint/page.jsp
checkout-page = /WEB-INF/jsp/checkout/page.jsp
purchase-page = /WEB-INF/jsp/purchase/page.jsp

The actions.properties file defines two sets of logical names. The first set maps requests to actions; for example, the go-shopping-action.do request is mapped to the actions.GoShoppingAction class listed in Example 12-2.b. The storefront-page logical name used by that action is also defined in the actions.properties file.

The Storefront

Now we come to the fruitstand’s storefront. Let’s briefly review how we got here. First, the user accesses the fruitstand’s homepage with the URL http://localhost:8080/case_study. That causes the welcome file—/index.jsp—to be displayed, which renders the homepage. The user clicks on the Go Shopping button, which submits a go-shopping-action.do request. That request is mapped to an action—GoShoppingAction—which creates a shopping cart and forwards the request to the storefront page. The storefront page is shown in Figure 12-4.
The storefront page is listed in Example 12-3.a.

Example 12-3.a  /WEB-INF/jsp/storefront/page.jsp

```jsp
<%@ taglib uri='regions' prefix='region' %>

<region:render region='STOREFRONT_REGION'/>
```

The storefront page renders the storefront region, which is defined in /WEBINF/jsp/regionDefinitions.jsp. That JSP file is partially listed in Example 12-1.b on page 398.
The storefront comprises four files: header.jsp, sidebar.jsp, content.jsp, and footer.jsp. Because the storefront header and footer are reused by the homepage, we’ve already seen their listings in Example 12-1.c and Example 12-1.d on page 399. The storefront sidebar is listed in Example 12-3.b.

**Example 12-3.b**  /WEB-INF/jsp/storefront/sidebar.jsp

```jsp
<%@ page contentType='text/html; charset=UTF-8' %>
<jsp:include page='../shared/flags.jsp' flush='true'/><p>
<jsp:include page='../shared/cart.jsp' flush='true'/></p>
```

Like the homepage sidebar, the storefront sidebar includes flags.jsp, so the user can change languages. The storefront sidebar includes another shared component that displays the items in the user’s shopping cart. That component—/WEB-INF/jsp/shared/cart.jsp—is listed in Example 12-4.b on page 408.

The main content displayed by the storefront is created by /WEB-INF/jsp/storefront/content.jsp, which is listed in Example 12-3.c.

**Example 12-3.c**  /WEB-INF/jsp/storefront/content.jsp

```jsp
<%@ page contentType='text/html; charset=UTF-8' %>
<%@ taglib uri='database' prefix='database' %>
<%@ taglib uri='html' prefix='html' %>
<%@ taglib uri='i18n' prefix='i18n' %>
<%@ taglib uri='logic' prefix='logic' %>

<font size='4' color='blue'>
<i18n:message key='storefront.form.title'/></font><p>
</p>
<database:query id='inventory' scope='session'>
SELECT * FROM Inventory
</database:query>

<% String currentItem = null, currentSku = null; %>
<table border='1' cellpadding='5'>
<tr><th><i18n:message key='storefront.table.header.picture'/></th>
<database:columnNames query='inventory' id='name'>
<logic:stringsNotEqual compare='SKU' to='<%= name %>'>
<% String hdrKey = “storefront.table.header.” + name.toLowerCase(); %>
```
## A Case Study

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fruit 1</td>
<td>Fresh Produce</td>
<td>0.00</td>
</tr>
<tr>
<td>Fruit 2</td>
<td>Organic Vegetables</td>
<td>1.00</td>
</tr>
<tr>
<td>Fruit 3</td>
<td>Juicy Berries</td>
<td>1.50</td>
</tr>
<tr>
<td>Fruit 4</td>
<td>Ripe Peaches</td>
<td>2.00</td>
</tr>
<tr>
<td>Fruit 5</td>
<td>Sweet Grapes</td>
<td>2.50</td>
</tr>
<tr>
<td>Fruit 6</td>
<td>Tasty Oranges</td>
<td>3.00</td>
</tr>
<tr>
<td>Fruit 7</td>
<td>Juicy Apples</td>
<td>3.50</td>
</tr>
<tr>
<td>Fruit 8</td>
<td>Sweet Tomatoes</td>
<td>4.00</td>
</tr>
<tr>
<td>Fruit 9</td>
<td>Juicy Pineapples</td>
<td>4.50</td>
</tr>
<tr>
<td>Fruit 10</td>
<td>Ripe Bananas</td>
<td>5.00</td>
</tr>
</tbody>
</table>

For each item, you can choose the quantity from the dropdown menu.

*Note: Prices are in USD.*
The storefront content page listed in Example 12-3.c reads inventory from a database. Subsequently, the content page iterates over column names and the columns themselves to create a table, as shown in Figure 12-4 on page 403.

The storefront content page also uses the \texttt{html:links} custom tag discussed in “Custom Tag Advanced Concepts” on page 32 to create HTML options that generate a request when they are selected. That request is \texttt{add-selection-to-cart-action.do}, which is mapped to \texttt{actions.AddToCartAction} in the \texttt{actions.properties} file listed in Example 12-2.c on page 402. That action class is discussed next.

The Shopping Cart

When a selection is made from one of the options in the storefront page shown in Figure 12-4 on page 403, a request is generated that maps to the \texttt{actions.AddToCartAction} action class. That class is listed in Example 12-4.a

\textbf{Example 12-4.a} \texttt{/WEB-INF/classes/AddToCartAction.java}

```java
package actions;

import java.util.Enumeration;
import java.util.Iterator;
import java.util.StringTokenizer;

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

import beans.app.Item;
import beans.app.ShoppingCart;

import action.ActionBase;
import action.ActionRouter;
```
public class AddToCartAction extends ActionBase
    implements beans.app.Constants {
    public ActionRouter perform(HttpServlet servlet,
        HttpServletRequest req,
        HttpServletResponse res)
        throws ServletException {
        Enumeration e = req.getParameterNames();
        String skuAndFruit = (String)e.nextElement();
        String amount = req.getParameterValues(skuAndFruit)[0];
        ShoppingCart cart = (ShoppingCart)req.getSession().
            getAttribute(SHOPPING_CART_KEY);

        if(cart == null) {
            throw new ServletException("No cart found");
        }

        StringTokenizer tok = new StringTokenizer(skuAndFruit, ";");
        String sku = (String)tok.nextElement(),
            fruit = (String)tok.nextElement(),
            price = (String)tok.nextElement();
        Iterator it = cart.getItems().iterator();
        boolean fruitWasInCart = false;

        while(it.hasNext()) {
            Item item = (Item)it.next();

            if(item.getName().equals(fruit)) {
                fruitWasInCart = true;
                item.setAmount(item.getAmount() +
                    Float.parseFloat(amount));
            }
        }

        if(!fruitWasInCart) {
            cart.addItem(new Item(Integer.parseInt(sku), fruit,
                Float.parseFloat(price),
                Float.parseFloat(amount)));
        }

        return new ActionRouter("storefront-page");
    }
}
The add-to-cart action listed in Example 12-4.a is invoked with a single request parameter of the form `sku-fruit-price=amount`; for example, if two pounds of grapefruit are selected at $0.49/lb, that request parameter will be 1004-grapefruit-0.49=2.0. The add-to-cart action parses that parameter and uses the resulting information to update the user’s shopping cart.

The action’s `perform` method returns an action router that points back to the storefront page, which causes the storefront to be redisplayed and the contents of the cart in the sidebar to be updated. The JSP page for that cart is listed in Example 12-4.b.

**Example 12-4.b** /WEB-INF/jsp/shared/cart.jsp

```jsp
<%@ taglib uri='application' prefix='app' %>

<app:iterateCart id='cartItem'>
  <tr>
    <td><%= cartItem.getName() %></td>
    <td><%= cartItem.getAmount() %></td>
  </tr>
</app:iterateCart>

<form action='checkout-action.do'>
  <input type='submit'value='checkout'/>
</form>
```

The JSP page listed in Example 12-4.b uses an application-specific custom tag to iterate over the items in the user’s cart. That custom tag is listed in Example 12-4.c.

---

5. Sku is an accounting term that stands for stock keeping unit, which simply means something in stock. Grapefruit was arbitrarily assigned an sku of 1004.
Example 12-4.c  /WEB-INF/classes/tags/app/CartIteratorTag.java

package tags.app;

import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.TagSupport;
import beans.app.User;
import beans.app.Users;
import beans.app.ShoppingCart;

public class CartIteratorTag extends tags.util.IteratorTag
    implements beans.app.Constants {
    public int doStartTag() throws JspException {
        ShoppingCart cart = (ShoppingCart)pageContext.getAttribute(
            SHOPPING_CART_KEY,
            PageContext.SESSION_SCOPE);
        if(cart == null) {
            throw new JspException("CartIteratorTag can’t find " +
                "cart");
        }
        setCollection(cart.getItems());
        return super.doStartTag();
    }
}

The custom tag listed in Example 12-4.c iterates over the items in the user’s shopping cart and makes the current item available as a scripting variable named by the tag’s id attribute. But none of that functionality is evident in Example 12-4.c because it’s inherited from IteratorTag, which is discussed in “Iteration” on page 36. CartIteratorTag.doStartTag invokes the setCollection method defined in its superclass and calls super.doStartTag before returning. The IteratorTag superclass takes care of the rest.

The cart in the storefront’s sidebar, which is listed in Example 12-4.b on page 408, contains a Checkout button that generates a checkout-action.do request. That request is mapped to the actions.CheckoutAction class in the actions.properties file; that properties file is listed in Example 12-2.c on page 402. The CheckoutAction class is listed in Example 12-5.a.
Example 12-5.a  /WEB-INF/classes/CheckoutAction.java

    package actions;

    import javax.servlet.ServletException;
    import javax.servlet.http.HttpServlet;
    import javax.servlet.http.HttpServletRequest;
    import javax.servlet.http.HttpServletResponse;
    import javax.servlet.http.HttpSession;
    import beans.app.ShoppingCart;
    import action.ActionBase;
    import action.ActionRouter;

    public class CheckoutAction extends ActionBase
        implements beans.app.Constants {
        public ActionRouter perform(HttpServlet servlet,
            HttpServletRequest req,
            HttpServletResponse res)
            throws ServletException {
            HttpSession session = req.getSession();
            ShoppingCart   cart = (ShoppingCart)session.getAttribute(
                SHOPPING_CART_KEY);
            if(cart == null) {
                throw new ServletException("Cart not found");
            }
            return new ActionRouter("checkout-page");
        }
    }

The checkout action checks to make sure the user has a shopping cart; if not, CheckoutAction.perform throws a servlet exception. After the shopping cart has been found, the checkout action returns an action router that forwards the request to the checkout page.

The Checkout

The checkout page, shown in Figure 12-5, displays an invoice of the items in the user’s shopping cart and lists the billing address. To complete the transaction, the user must activate the Purchase The Items Listed Above button.
The checkout page shown in Figure 12-5 is listed in Example 12-5.b.

Example 12-5.b  /WEB-INF/jsp/checkout/page.jsp

```jsp
<%@ taglib uri='security' prefix='security'%>
<%@ taglib uri='regions' prefix='region' %>

<security:enforceLogin
    loginPage='/WEB-INF/jsp/login/page.jsp'
    errorPage='/WEB-INF/jsp/loginFailed/page.jsp'/>

<region:render region='CHECKOUT_REGION'/>
```

Figure 12-5  The Checkout Page

Here are the items you selected.

<table>
<thead>
<tr>
<th>Item</th>
<th>Amount</th>
<th>Price/Lb</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>apple</td>
<td>2.0 lbs</td>
<td>$0.29</td>
<td>$0.58</td>
</tr>
<tr>
<td>banana</td>
<td>3.0 lbs</td>
<td>$0.50</td>
<td>$1.77</td>
</tr>
<tr>
<td>cantaloupe</td>
<td>1.0 lbs</td>
<td>$0.19</td>
<td>$0.19</td>
</tr>
<tr>
<td>grapes</td>
<td>2.5 lbs</td>
<td>$0.49</td>
<td>$1.23</td>
</tr>
<tr>
<td>grapes</td>
<td>3.0 lbs</td>
<td>$0.79</td>
<td>$2.37</td>
</tr>
</tbody>
</table>

Total: $0.14

The items listed above will be billed to:

James Wilson
24996 Regent Lane
Boise, New York
USA
Visa

Purchase The Items Listed Above

Thanks for stopping by. Today is Wednesday, February 21, 2002.
Like the storefront page listed in Example 12-3.a on page 403, the checkout page renders a region, in this case CHECKOUT_REGION, which is defined like this:

```xml
<region:define id='CHECKOUT_REGION' region='LOGIN_REGION'>
  <region:put section='content' content='/WEB-INF/jsp/checkout/content.jsp'/>
</region:define>
```

The CHECKOUT_REGION extends LOGIN_REGION and overrides only the content section. That means that the checkout region is identical to the login region, except for the main content of the page. The main content of the checkout region is supplied by `/WEB-INF/jsp/checkout/content.jsp`, which is listed in Example 12-5.c.

Example 12-5.c /WEB-INF/jsp/checkout/content.jsp

```jsp
<%@ page contentType='text/html; charset=UTF-8' %>
<%@ page import='beans.app.User' %>
<%@ taglib uri='application' prefix='app' %>
<%@ taglib uri='i18n' prefix='i18n' %>

<font size='4' color='blue'>
<i18n:message base='app' key='checkout.title'/>
</font><p>

<img src='graphics/cart.gif'/>

<table cellpadding='10'>
  <th><i18n:message base='app' key='checkout.table.header.item'/></th>
  <th><i18n:message base='app' key='checkout.table.header.amount'/></th>
  <th><i18n:message base='app' key='checkout.table.header.pricePerLb'/></th>
  <th><i18n:message base='app' key='checkout.table.header.price'/></th>
  <tbody>
    <% double total = 0.0; %>
    <app:iterateCart id='item'>
      <% String  name = item.getName();
      float    amt = item.getAmount(),
      price = item.getPrice();
      %>
      <tr>
        <td><%= name %></td>
Like the storefront sidebar listed in Example 12-4.b on page 408, the checkout content page uses the application-specific `CartIterator` custom tag to iterate over the items in the user’s shopping cart. Those items are used to construct the invoice and display a total price.

To obtain billing and shipping information, the checkout content page accesses a `User` bean from the current session. See “Authentication” on page 443 for more information about how that `User` bean is created and stored in session scope.

6. The fruitstand application assumes that billing and shipping information are the same.
The checkout content page also contains a form with a submit button that creates a `purchase-action.do` request, which maps to `actions.PurchaseAction`. That action class is listed in Example 12-5.d.

Example 12-5.d  /WEB-INF/classes/actions/PurchaseAction.java

```java
class PurchaseAction extends ActionBase
    implements beans.app.Constants,
          tags.security.Constants {
    public ActionRouter perform(HttpServlet servlet,
                               HttpServletRequest req,
                               HttpServletResponse res)
        throws ServletException {
        HttpSession session = req.getSession();
        ShoppingCart cart = (ShoppingCart)session.getAttribute(
                           SHOPPING_CART_KEY);
        if(cart == null) {
            throw new ServletException("Cart not found");
        }
        return new ActionRouter("purchase-page");
```  

Like the checkout action listed in Example 12-5.a on page 410, the purchase action checks to make sure the user has a shopping cart in the session. If so, the purchase action forwards the request to the purchase page.

The Purchase

The purchase page is a simple JSP page that thanks the user for their purchase and displays an expected shipping date. That JSP page is shown in Figure 12-6.
The purchase-page referenced by the purchase action listed in Example 12-5.d maps to /WEB-INF/jsp/purchase/content.jsp, which is listed in Example 12-5.e.

Example 12-5.e /WEB-INF/jsp/purchase/page.jsp

```jsp
<%@ taglib uri='regions' prefix='region' %>
<region:render region='PURCHASE_REGION'/>
```

The purchase page, like the homepage, storefront, and checkout pages, is defined by a region. That region is listed below:

```jsp
<region:define id='PURCHASE_REGION' region='LOGIN_REGION'>
  <region:put section='content'
    content='/WEB-INF/jsp/purchase/content.jsp'/>
</region:define>
```

Like CHECKOUT_REGION, PURCHASE_REGION extends LOGIN_REGION and redefines only the content section. That section is generated by /WEB-INF/jsp/purchase/content.jsp, which is listed in Example 12-5.f.

Example 12-5.f /WEB-INF/jsp/purchase/content.jsp

```jsp
<%@ page contentType='text/html; charset=UTF-8' %>
<%@ taglib uri='i18n' prefix='i18n' %>
<font size='4' color='blue'>
  <i18n:message base='app' key='purchase.title'/></font>
```
Advanced JavaServer Pages

```
<i18n:message base='app' key='purchase.willBeShippedOn'/>
<i18n:format date='<%= new java.util.Date() %>'
  dateStyle='<%= java.text.DateFormat.SHORT %>'/></p>
</font>
```

The purchase page uses the `i18n:message` tag to display its messages. That page also uses the `i18n:format` tag to predict an optimistic shipping date.

This concludes our walkthrough of the fruitstand’s main use case. The rest of this chapter examines the fruitstand application from two other perspectives: the MVC framework upon which the application is built and an examination of some of the application’s other features, such as internationalization and authentication.

The Model 2 Framework

From “The Fruitstand” on page 392, it’s apparent that the fruitstand application is implemented with small chunks of functionality that are plugged into a framework. That framework is the Model 2 framework discussed in “A Model 2 Framework” on page 154 that allows web applications to be implemented in a Model-View-Controller (MVC) style. This section examines the fruitstand’s use of that framework, starting with the model, followed by the views and controllers.

The Model

The fruitstand’s model consists of a database and beans, as depicted in Figure 12-7. Those beans, which reside in `WEB-INF/classes/beans/app`, are listed below:

- **User**: A fruitstand customer
- **Users**: A collection of users initialized from a database
- **Item**: An item for sale
- **Inventory**: A collection of items
- **Shopping Cart**: An inventory of the items a user has selected

In addition to the beans listed above, the fruitstand application also defines a number of constants that are used throughout the application.
The fruitstand application uses a database that maintains the fruitstand’s inventory and list of users. That database is created by the CreateDB Java application that resides in the `/WEB-INF/classes/util` directory.

**The Database**

Figure 12-8 shows the inventory table from the fruitstand’s database. That table maintains a stock keeping unit (sku), the name of the sku’s corresponding fruit, and that fruit’s price.

The fruitstand’s database also maintains a table of users. That table has 14 columns, which is too wide to show effectively in Figure 12-8. The `User` table stores a user’s first and last name, address, credit card information, username and password, and the user’s role.
Advanced JavaServer Pages

The `CreateDB` application, which creates the fruitstand’s database, is listed in Example 12-6.

Example 12-6  /WEB-INF/util/CreateDB.java

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class CreateDB {
    private Connection conn;
    private Statement stmt;

    public static void main(String args[]) {
        new CreateDB();
    }

    public CreateDB() {
        try {
            loadJDBCDriver();
            conn = getConnection("F:/databases/sunpress");
            stmt = conn.createStatement();

            createTables(stmt);
            populateTables(stmt);

            stmt.close();
            conn.close();
        }
    }

    private void loadJDBCDriver() {
        try {
            Class.forName("com.sun.jdbc.driver.jdbc30.jdbc30Driver");
        }
        catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    private Connection getConnection(String dbURL) {
        try {
            return DriverManager.getConnection(dbURL);
        }
        catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }

    private void createTables(Statement stmt) {
        try {
            stmt.execute("CREATE TABLE Inventory (SKU INT NOT NULL PRIMARY KEY, NAME VARCHAR(20) NOT NULL, PRICE DECIMAL(5,2) NOT NULL);
                      CREATE TABLE Sales (SKU INT NOT NULL PRIMARY KEY, QUANTITY INT);");
        }
        catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }

    private void populateTables(Statement stmt) {
        try {
            stmt.execute("INSERT INTO Inventory VALUES (1001, 'apple', 0.29);
                      INSERT INTO Inventory VALUES (1002, 'basamic', 0.69);
                      INSERT INTO Inventory VALUES (1003, 'cantaloupe', 0.19);
                      INSERT INTO Inventory VALUES (1004, 'grapefruit', 0.49);
                      INSERT INTO Inventory VALUES (1005, 'grapes', 0.79);
                      INSERT INTO Inventory VALUES (1006, 'kiwi', 0.99);
                      INSERT INTO Inventory VALUES (1007, 'peach', 0.39);
                      INSERT INTO Inventory VALUES (1008, 'pear', 0.69);
                      INSERT INTO Inventory VALUES (1009, 'pineapple', 0.29);
                      INSERT INTO Inventory VALUES (1010, 'strawberry', 0.89);
                      INSERT INTO Inventory VALUES (1011, 'watermelon', 0.29);
                      
                      INSERT INTO Sales VALUES (1001, 10);
                      INSERT INTO Sales VALUES (1002, 20);
                      INSERT INTO Sales VALUES (1003, 30);
                      INSERT INTO Sales VALUES (1004, 40);
                      INSERT INTO Sales VALUES (1005, 50);
                      INSERT INTO Sales VALUES (1006, 60);
                      INSERT INTO Sales VALUES (1007, 70);
                      INSERT INTO Sales VALUES (1008, 80);
                      INSERT INTO Sales VALUES (1009, 90);
                      INSERT INTO Sales VALUES (1010, 100);
                      INSERT INTO Sales VALUES (1011, 110);");
        }
        catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }

    private void close(Statement stmt) {
        try {
            stmt.close();
        }
        catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }

    private void close(Connection conn) {
        try {
            conn.close();
        }
        catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
}

Figure 12-8  The Inventory Table

<table>
<thead>
<tr>
<th>SKU</th>
<th>NAME</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>apple</td>
<td>0.29</td>
</tr>
<tr>
<td>1002</td>
<td>basamic</td>
<td>0.69</td>
</tr>
<tr>
<td>1003</td>
<td>cantaloupe</td>
<td>0.19</td>
</tr>
<tr>
<td>1004</td>
<td>grapefruit</td>
<td>0.49</td>
</tr>
<tr>
<td>1005</td>
<td>grapes</td>
<td>0.79</td>
</tr>
<tr>
<td>1006</td>
<td>kiwi</td>
<td>0.99</td>
</tr>
<tr>
<td>1007</td>
<td>peach</td>
<td>0.39</td>
</tr>
<tr>
<td>1008</td>
<td>pear</td>
<td>0.69</td>
</tr>
<tr>
<td>1009</td>
<td>pineapple</td>
<td>0.29</td>
</tr>
<tr>
<td>1010</td>
<td>strawberry</td>
<td>0.89</td>
</tr>
<tr>
<td>1011</td>
<td>watermelon</td>
<td>0.29</td>
</tr>
</tbody>
</table>
DriverManager.getConnection(
    "jdbc:cloudscape;shutdown=true");

// Code for creating and populating tables omitted for brevity.
"('1008', 'pear', '0.69')," +
"('1009', 'pineapple', '0.29')," +
"('1010', 'strawberry', '0.89')," +
"('1011', 'watermelon', '0.29')")

} catch(SQLException ex) {
    ex.printStackTrace();
}

private void loadJDBCDriver() {
    try {
        Class.forName("COM.cloudscape.core.JDBCDriver");
    } catch(ClassNotFoundException e) {
        e.printStackTrace();
    }
}

private Connection getConnection(String dbName) {
    Connection con = null;
    try {
        con = DriverManager.getConnection("jdbc:cloudscape:" + dbName + ";create=true");
    } catch(SQLException sqe) {
        System.err.println("Couldn’t access “ + dbName);
    }
    return con;
}

The application listed in Example 12-6 connects to a Cloudscape database and creates a database in f:/databases/sunpress. The application subsequently populates the database with the fruitstand's two tables, shuts down the database, and exits.

It's easy to adapt the application listed in Example 12-6 for a different database vendor by changing the driver name used in the loadJDBCDriver method and the database URL used in getConnection.

The Beans

The beans used in the fruitstand application are simple versions of the canonical User, Users, Inventory, and Shopping Cart objects. The User class is listed in Example 12-7.a.
package beans.app;

// Users are immutable to eliminate multithreading concerns.

public class User implements java.io.Serializable {
    private final String firstName, lastName, address, city, state;
    private final String country, creditCardType, creditCardNumber;
    private final String creditCardExpiration;
    private final String userName, password, pwdHint, roles;

    public User(String firstName, String lastName, String address,
                 String city, String state, String country,
                 String creditCardType, String creditCardNumber,
                 String creditCardExpiration, String userName,
                 String password, String pwdHint, String roles) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.address = address;
        this.city = city;
        this.state = state;
        this.country = country;
        this.creditCardType = creditCardType;
        this.creditCardNumber = creditCardNumber;
        this.creditCardExpiration = creditCardExpiration;
        this.userName = userName;
        this.password = password;
        this.pwdHint = pwdHint;
        this.roles = roles;
    }

    public String getFirstName() { return firstName; }
    public String getLastName()  { return lastName;  }
    public String getAddress()   { return address;   }
    public String getCity()      { return city;      }
    public String getState()     { return state;     }
    public String getCountry()   { return country;   }
    public String getCreditCardType()   { return creditCardType; }
    public String getCreditCardNumber() { return creditCardNumber; }
    public String getCreditCardExpiration() {
        return creditCardExpiration;
    }

    public String getUserName() { return userName; }
    public String getPassword() { return password; }
    public String getPwdHint()  { return pwdHint;  }
    public String getRoles()    { return roles;    }
}
public boolean equals(String uname, String pwd) {
    return getUserName().equals(uname) &&
    getPassword().equals(pwd);
}

The User class maintains read-only information about a single user. That class is immutable to eliminate multithreading concerns. If you want to change information about a user, the original User instance must be replaced. Because modifying user data is typically infrequent and multithreaded access to users can be common, an immutable User class makes sense in this case.

Users are maintained by a Users class, which is listed in Example 12-7.b.

Example 12-7.b /WEB-INF/classes/beans/app/Users.java

```java
package beans.app;

import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.util.Enumeration;
import java.util.Hashtable;

public class Users {
    private final static int FIRST_NAME=1, LAST_NAME=2, ADDRESS=3,
    CITY=4, STATE=5, COUNTRY=6,
    CREDIT_TYPE=7, CREDIT_NUMBER=8,
    CREDIT_EXPIRE=9, USER_NAME=10,
    PASSWORD=11, PASSWORD_HINT=12,
    ROLES=13;

    private final Hashtable users = new Hashtable();

    public Users(ResultSet rs) {
        try {
            ResultSetMetaData rsmd = rs.getMetaData();

            if(rsmd.getColumnCount() > 0) {
                boolean moreRows = rs.next(); // point to first
                // row initially

                while(moreRows) {
                    addUser(new User(
                        ((String)rs.getObject(FIRST_NAME)).trim(),
                        ((String)rs.getObject(LAST_NAME)).trim(),
                        ((String)rs.getObject(ADDRESS)).trim(),
                        ((String)rs.getObject(CITY)).trim(),
```
```
((String)rs.getObject(STATE)).trim(),
((String)rs.getObject(COUNTRY)).trim(),
((String)rs.getObject(CREDIT_TYPE)).trim(),
((String)rs.getObject(CREDIT_NUMBER)).trim(),
((String)rs.getObject(CREDIT_EXPIRE)).trim(),
((String)rs.getObject(USER_NAME)).trim(),
((String)rs.getObject(PASSWORD)).trim(),
((String)rs.getObject(PASSWORD_HINT)).trim(),
((String)rs.getObject(ROLES)).trim()));

moreRows = rs.next(); // move to next row

} catch(SQLException ex) {
    // can’t throw an exception from a constructor
}

public int getNumberOfUsers() {
    return users.size();
}

public User addUser(User user) {
    users.put(user.getUserName(), user);
    return user;
}

public User getUser(String username, String password) {
    User user = getUser(username);
    boolean found = false;

    if(user != null) {
        found = user.equals(username, password);
    }

    return found ? user : null;
}

public User getUser(String username) {
    return (User)users.get(username);
}

public Hashtable getUsers() {
    return users;
}

public String getPasswordHint(String username) {
    User user = getUser(username);
    return user != null ? user.getPwdHint() : null;
}
Because users are stored in a database, the Users constructor is passed a ResultSet object, which is used to create users. Besides the constructor, the Users class provides a number of accessor methods for retrieving information associated with a user.

Like the User class, the Users class is thread safe because its members are immutable.

Example 12-7.c lists the Item class, which represents an item in an inventory.

Example 12-7.c  /WEB-INF/classes/beans/app/Item.java

    package beans.app;

    // This is an item in an inventory or shopping cart (same thing)
    // with four properties: sku, (stock keeping unit) name, price,
    // and amount. Items are nearly immutable to eliminate
    // multithreading concerns.

    public class Item implements java.io.Serializable {
        private final int sku; // stock keeping unit
        private final float price;
        private final String name;
        private float amount;

        public Item(int sku, String name, float price, float amount) {
            this.sku    = sku;
            this.name   = name;
            this.amount = amount;
            this.price  = price;
        }

        public int    getSku()    { return sku;    }
        public String getName()   { return name;   }
        public float  getPrice()  { return price;  }
        public synchronized float getAmount() {
            return amount;
        }
        public synchronized void setAmount(float amount) {
            this.amount = amount;
        }
    }

The Item class maintains a stock keeping unit, name, price, and an amount for each item. All that information is read-only, except for the price, which is expected to change relatively frequently. The setter and getter methods for the amount are both synchronized, so the Item class is thread safe.
Items are maintained by an inventory; the Inventory class is listed in Example 12-7.d.

Example 12-7.d /WEB-INF/classes/beans/app/Inventory.java

```java
package beans.app;

import java.util.Iterator;
import java.util.Vector;

public class Inventory implements java.io.Serializable {
    final protected Vector items;

    public Inventory() {
        items = new Vector();
    }
    public void addItem(Item item) {
        items.add(item);
    }
    public void removeItem(Item item) {
        items.remove(item);
    }
    public Vector getItems() {
        return items;
    }
}
```

The Inventory class is a simple façade for a vector of items; it allows items to be added and removed from the inventory.\(^7\) The Inventory class also provides a getItems method, for convenience, that returns the vector of items. The caller of that method should not modify that vector.

Example 12-7.e lists the ShoppingCart class.

Example 12-7.e /WEB-INF/classes/beans/app/ShoppingCart.java

```java
package beans.app;

public class ShoppingCart extends Inventory {
}
```

For the purposes of the fruitstand application, a shopping cart is the same thing as an inventory; however, the ShoppingCart class exists as a placeholder for functionality specific to shopping carts that may be added in the future.

---

7. See “Façade Design Pattern for HTML Forms” on page 79 for more information about the Façade design pattern.
The fruitstand application uses a number of constants, which are defined in the Constants interface. That class is listed in Example 12-7.f.

Example 12-7.f  /WEB-INF/classes/beans/app/Constants.java

```java
package beans.app;

// These constants are mostly used by this application’s actions
// -- see /WEB-INF/classes/actions.

public interface Constants {
    // this prefix provides some degree of uniqueness for the
    // constants defined below.
    static final String prefix = "beans.app";

    // Keys for attributes
    static final String/locale = prefix + ".locale",
    SHOPPING_CART_KEY = prefix + ".cart",
    USERS_KEY = prefix + ".users",
    USERNAME_KEY = prefix + ".username",
    PASSWORD_KEY = prefix + ".password",
    CONFIRM_PASSWORD_KEY = prefix + ".cnfrmpwd",
    PASSWORD_HINT_KEY = prefix + ".pwdhint",

    // Default values
    DEFAULT_I18N_BASE = "app";
}
```

The constants defined by the Constants interface are accessed by implementing that interface; for example, the PurchaseAction class listed in Example 12-5.d on page 414 implements the Constants interface and uses the SHOPPING_CART_KEY.

The Views—JSP Pages and Templates

The most dynamic and interesting aspect of the fruitstand application is its views, which are constructed with regions and templates. The goal of this section is to illustrate how the fruitstand application uses regions and templates. This section does not show how regions or templates work; for that discussion, see “Templates” on page 96.
Figure 12-9 shows the files used to create all nine of the fruitstand’s JSP pages. Each directory under `/WEB-INF/jsp`, with the exception of the `shared` and `templates` directories, contains all of the JSP files specific to one JSP page; for example, `/WEB-INF/jsp/accountCreated` contains all of the JSP files—`page.jsp` and `content.jsp`—that are specific to the account-created JSP page.

The `shared` directory contains JSP files that are used by more than one JSP page, and the `templates` directory contains the fruitstand application’s lone template.
Each JSP page in the fruitstand application is defined by a region, which uses a template, and the content that's inserted into that template. All of the regions in the application use the same template: /WEB-INF/jsp/templates/hscf.jsp. That template displays four regions: header, sidebar, content, and footer; thus, the template is named hscf.jsp.

All of the fruitstand’s regions are defined in one file, /WEB-INF/jsp/regionDefinitions.jsp, which is listed in Example 12-7.g.

Example 12-7.g /WEB-INF/jsp/regionDefinitions.jsp

```jsp
<%@ taglib uri='regions' prefix='region' %>

<region:define id='STOREFRONT_REGION' template='/WEB-INF/jsp/templates/hscf.jsp'>
  <region:put section='title' content='FruitStand.com' direct='true'/>
  <region:put section='background' content='graphics/blueAndWhiteBackground.gif' direct='true'/>
  <region:put section='header' content='/WEB-INF/jsp/storefront/header.jsp'/>
  <region:put section='sidebar' content='/WEB-INF/jsp/storefront/sidebar.jsp'/>
  <region:put section='content' content='/WEB-INF/jsp/storefront/content.jsp'/>
  <region:put section='footer' content='/WEB-INF/jsp/storefront/footer.jsp'/>
</region:define>

<region:define id='LOGIN_REGION' region='STOREFRONT_REGION'>
  <region:put section='header' content='/WEB-INF/jsp/login/header.jsp'/>
  <region:put section='sidebar' content='/WEB-INF/jsp/login/sidebar.jsp'/>
  <region:put section='content' content='/WEB-INF/jsp/login/content.jsp'/>
</region:define>
```
Most of the regions defined in Example 12.7 reuse content from another region; for example, the HOMEPAGE_REGION reuses the title, background, and footer from the STOREFRONT_REGION, as discussed in “The Homepage” on page 395. All the other regions reuse content from either LOGIN_REGION or STOREFRONT_REGION.
Defining all of an application’s regions in one file eases maintenance of those regions; for example, it’s easy to change the content displayed by any JSP page simply by modifying the file listed in Example 12-7.g. The ability for regions to extend each other enables regions to share content and facilitates a more readable definition of those regions.

The only template defined by the fruitstand application is listed in Example 12-7.h.

**Example 12-7.h**  /WEB-INF/jsp/templates/hscf.jsp

```html
<html><head>
   <%@ taglib uri='regions' prefix='region' %>
   <title><region:render section='title'/></title>
</head>

<body background='&lt;region:render section='background'/&gt;'>

<table>
   <tr valign='top'>
      <td><region:render section='sidebar'/></td>
      <td>
         <table>
            <tr><td><region:render section='header'/></td></tr>
            <tr><td><region:render section='content'/></td></tr>
            <tr><td><region:render section='footer'/></td></tr>
         </table>
      </td>
   </tr>
</table>

</body></html>
```

The template listed in Example 12-7.h uses an HTML table to display content. That content is supplied by JSP files that are referenced by a region. So, for example, when LOGIN_REGION is rendered, the template listed in Example 12-7.h uses the content defined in that region; namely, the storefront’s title, background, and footer and the login page’s content section.

Notice that each directory in the fruitstand application that corresponds to a JSP page contains a JSP file titled page.jsp. That page renders a corresponding region; for example, Example 12-7.i lists the page.jsp file for the login page. That JSP file renders the login region.
Example 12-7.i  /WEB-INF/jsp/login/page.jsp

```jsp
<%@ taglib uri='regions' prefix='region' %>

<region:render region='LOGIN_REGION'/>
```

The fruitstand’s page.jsp files are responsible for features central to pages; for example, the page.jsp file listed in Example 12-7.j restricts access to the checkout page to users that have logged in.

Example 12-7.j  /WEB-INF/jsp/checkout/page.jsp

```jsp
<%@ taglib uri='security' prefix='security' %>
<%@ taglib uri='regions' prefix='region' %>

<security:enforceLogin
    loginPage='/WEB-INF/jsp/login/page.jsp'
    errorPage='/WEB-INF/jsp/loginFailed/page.jsp'/>

<region:render region='CHECKOUT_REGION'/>
```

The security:enforceLogin tag used in Example 12-7.j evaluates the rest of the JSP file only if the user has logged in; if not, the request is forwarded to the login page. You can read more about the security:enforceLogin tag in “Security” on page 250. Authentication for the fruitstand application is discussed in “Authentication” on page 443.

Templates are powerful because they separate page layout from page content, both of which can change frequently during development. Because of that separation, you can change the layout of multiple JSP pages by changing a single template. Conversely, you can make changes to content without affecting layout.

Regions specify content used by templates. The ability to specify regions somewhere other than where they are used, as is the case for the fruitstand application, lets you centralize an application’s JSP page descriptions. That ability simplifies the implementation and maintenance of JSP pages.

Regions can also be defined in terms of another region. That allows one region to “inherit” sections from another region, thereby simplifying page construction and encouraging reuse.

Finally, both templates and regions encourage you to construct JSP pages out of small, reusable components, each of which represents a particular section of a region. Constructing JSP pages in that manner results in web applications that are flexible, malleable, and extensible.
Now that we’ve seen how the fruitstand’s model and views are implemented, let’s take a look at the application’s controllers.

The Controllers—Servlets and Actions

In a Model-View-Controller (MVC) application, data is stored in a model and displayed by views. Controllers are the glue that connects models and views. Controllers react to events and may access the model before forwarding control to a view.

The fruitstand’s controllers are servlets and actions, both of which are discussed in the following sections.

Servlets

The fruitstand uses four servlets that reside in `/WEB-INF/classes`, as shown by Figure 12-10.

The `ActionServlet` is an integral part of the fruitstand’s Model 2 framework and handles all HTTP requests that end in `.do`. Because that servlet is discussed extensively in “A Model 2 Framework” on page 154, it is not discussed here.

The `AuthenticateServlet` is an abstract base class that’s extended by `AppAuthenticateServlet`. Those servlets are used to enforce authentication, as discussed in “Authentication” on page 443.
The `SetupServlet` loads at startup and initializes the database. That servlet is listed in Example 12-8.b on page 435.

All of the fruitstand’s servlets are configured in the application’s `web.xml` file, which is listed in Example 12-8.a.

Example 12-8.a /WEB-INF/web.xml

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app 
 PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN" "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
  <servlet>
    <servlet-name>authenticate</servlet-name>
    <servlet-class>AppAuthenticateServlet</servlet-class>
  </servlet>

  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>ActionServlet</servlet-class>
    <init-param>
      <param-name>action-mappings</param-name>
      <param-value>actions</param-value>
    </init-param>
  </servlet>

  <servlet>
    <servlet-name>setup</servlet-name>
    <servlet-class>SetupServlet</servlet-class>
    <init-param>
      <param-name>jdbcDriver</param-name>
      <param-value>COM.cloudscape.core.JDBCDriver</param-value>
    </init-param>
    <init-param>
      <param-name>jdbcURL</param-name>
      <param-value>jdbc:cloudscape:f:/databases/sunpress;create=false</param-value>
    </init-param>
    <init-param>
      <param-name>jdbcUser</param-name>
      <param-value>roymartin</param-value>
    </init-param>
  </servlet>
</web-app>
```
The `web.xml` file defines servlet names and classes and, where appropriate, servlet initialization parameters. The `SetupServlet` has initialization parameters for the JDBC URL and driver and a username and password for the application’s database.

Additionally, `web.xml` specifies the custom tag libraries used by the fruitstand application. The listing in Example 12-8.a is truncated in the interests of brevity because the application uses a number of custom tag libraries. Besides the application custom tag library listed above, the fruitstand application uses the following tag libraries:

- database
- DOM
The custom tag libraries listed above are discussed throughout this book, and therefore are not covered here.

The SetupServlet, which initializes the database, is listed in Example 12-8.b.

**Example 12-8.b** /WEB-INF/classes/SetupServlet.java

```java
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.Statement;
import java.sql.SQLException;
import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import beans.app.User;
import beans.app.Users;
import beans.jdbc.DbConnectionPool;

public class SetupServlet extends HttpServlet
    implements beans.app.Constants,
           tags.jdbc.Constants {

    private DbConnectionPool pool;

    public void init(ServletConfig config) throws ServletException{
        super.init(config);

        ServletContext ctx = config.getServletContext();
        createDbConnectionPool(config, ctx);

        try {
            ctx.setAttribute(USERS_KEY, loadUsers(ctx));
        }
        catch(SQLException ex) {
            throw new ServletException(ex);
        }
    }

    public void destroy() {
```
ServletContext ctx = getServletConfig().getServletContext();
ctx.removeAttribute(DBPOOL_KEY);
ctx.removeAttribute(USERS_KEY);

pool.shutdown();
pool = null;
super.destroy();
}

private void createDbConnectionPool(ServletConfig config,
        ServletContext ctx) {
    pool = new DbConnectionPool(
        config.getInitParameter("jdbcDriver"),
        config.getInitParameter("jdbcURL"),
        config.getInitParameter("jdbcUser"),
        config.getInitParameter("jdbcPwd"));
    ctx.setAttribute(DBPOOL_KEY, pool);
}

private Users loadUsers(ServletContext ctx)
        throws SQLException {
    Connection conn = null;
    if(pool != null) {
        try {
            // wait for a maximum of 10 seconds for a connection
            // if pool is full
            conn = (Connection)pool.getConnection(10000);
        }
        catch(Exception ex) {
            throw new SQLException(ex.getMessage());
        }
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT * FROM USERS");
        Users users = new Users(rs);
        pool.recycleConnection(conn);
        return users;
    }
    return null;
}

The servlet listed in Example 12-8.b creates a database connection pool with the initialization parameters specified in web.xml. The implementation and use of that database connection pool is discussed in “Connection Pooling” on page 302.

After creating the database connection pool, the SetupServlet performs a query on the database and constructs the application’s users based on the results of that query.
The servlet container invokes `SetupServlet` when the application is first accessed, as specified by the `load-on-startup` tag in `web.xml`.

Servlets are only half of the fruitstand’s controllers. Much of the responsibility for reacting to events and forwarding to views is shouldered by the fruitstand’s actions.

**Actions**

When an HTTP request ending in `.do` is initiated in the fruitstand application, it is ultimately handled by an action. Exactly how that is implemented is discussed extensively in “A Model 2 Framework” on page 154 and therefore is not covered here. Nevertheless, actions are integral to a Model 2 web application, and the fruitstand uses them heavily, as you can see from Figure 12-11.

*Figure 12-11  Model 2 Framework Action Classes*
Advanced JavaServer Pages

The /WEB-INF/classes/action and /WEB-INF/classes/action/events directories contain the classes that constitute the Model 2 framework. Those classes are discussed in “A Model 2 Framework” on page 154 and “Event Handling and Sensitive Form Resubmissions” on page 182.

The /WEB-INF/classes/actions directory houses the fruitstand’s application-specific actions. Each of those nine actions represents a single use case, which is evident from their names. None of those actions are discussed further in this section, but you can find discussions of them throughout this chapter; for example, see “The Shopping Cart” on page 406 for a discussion of AddToCartAction and “The Checkout” on page 410 for a discussion of CheckoutAction.

Internationalization

We’ve already seen that the fruitstand application uses the custom tags discussed in “I18n” on page 206 to internationalize text, numbers, dates, and currency. This section explores how fruitstand users can change languages on the fly.

Figure 12-12 shows the fruitstand’s login page in three different languages. To switch from one language to another, the user clicks on one of the flags in the sidebar; that action changes the application’s locale and updates the current page.

The steps in the user switches language use case are listed below.

1. The user clicks on a flag in the fruitstand’s sidebar.
2. The application updates its locale according to the selected flag.
3. The application redisplays the current page.
Figure 12-12 The FruitStand in English, German, and Chinese
Figure 12-13 shows the files involved in the use case outlined above.

The fruitstand application maintains three properties files that each define all of the text displayed by the application. Each of those files defines the same messages in a different language. The English properties file is partially listed in Example 12-9.a.

Example 12-9.a  /WEB-INF/classes/app_en.properties

```properties
    click=click
    here=here

    messages.login-header-title=FruitStand.com
    messages.today=Today is {0, date}

    homepage.title=A Model 2 JSP Application
```
This mockup of a fruit stand uses many of the techniques discussed in Advanced JavaServer Pages, published by Sun Microsystems Press and Prentice-Hall in May 2001.

In the interests of brevity, the properties files listed in this section are all truncated. The German properties file is listed in Example 12-9.b.

Example 12-9.b /WEB-INF/classes/app_de.properties

click=Klick
here=Hier

messages.login-header-title=FruitStand.com
messages.today=Huite ist {0, date}

homepage.title=Eine Model 2 JSP Anwendung

Example 12-9.c lists the Chinese properties file, which unlike the English and German versions defines messages with Unicode escape sequences. See “Unicode” on page 208 for more information concerning Unicode.

Example 12-9.c /WEB-INF/classes/app_zh.properties

click=\u70b9\u51fb
here=\u8fd9\u91cc

messages.login-header-title=\u6b22\u8fce\u5149\u4e34
messages.today=\u4eca\u5929\u662f {0, date}

homepage.title=\u4e00\u4e2a\u6a21\u5f0f\u4e8cJSP\u8303\u4f8b
Example 12-9.d lists the flags.jsp file.

Example 12-9.d /WEB-INF/jsp/shared/flags.jsp

```jsp
<% String thisPage = request.getServletPath();
    String updateLocaleAction = "update-locale-action.do?page=" +
        thisPage + "&country=";
%>
<table width='160'>
<tr>
    <a href='<%= updateLocaleAction + "EN" %>'>
        <img src='graphics/flags/britain_flag.gif'/></a>
    <a href='<%= updateLocaleAction + "DE" %>'>
        <img src='graphics/flags/german_flag.gif'/></a>
    <a href='<%= updateLocaleAction + "ZH" %>'>
        <img src='graphics/flags/chinese_flag.gif'/></a>
</tr>
</table>
```

The JSP file listed in Example 12-9.d constructs actions for each of the flags according to the current page and the country those flags represent; for example, the flags.jsp file is included by /WEB-INF/jsp/storefront/sidebar, so if a user clicks on the Chinese flag while viewing the storefront, the following action would be invoked:

http://localhost:8080/case-study/update-locale-
action.do?page=/WEB-INF/jsp/storefront/page.jsp&country=ZH

The actions associated with the flags from flags.jsp result in a call to the UpdateLocaleAction, which is listed in Example 12-9.e.

Example 12-9.e /WEB-INF/classes/actions/UpdateLocaleAction.java

```java
package actions;

import java.util.Locale;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import action.ActionBase;
import action.ActionRouter;
```
The update locale action obtains the country associated with the desired language from a request parameter that is generated by the JSP file listed in Example 12-9 on page 442. That action subsequently sets a session attribute that indicates the current locale, sets the response’s locale, and then forwards the request to the original page.

When the original page is redisplayed, i18n custom tags render internationalized text. Those tags use the locale in the user’s session.

### Authentication

Anyone can shop at the fruitstand, but only logged-in users can check out their purchase. That security constraint is implemented with a use case called authenticate user. The steps involved in that use case are listed below.

1. When the user tries to check out, a custom tag checks to see that the user is logged in.
2. If the user is not logged in, the custom tag in step #1 forwards to the login page.
3. The user clicks on the create account link on the login page.
4. The application displays a form for creating a new account.
5. The user fills out the form and activates the create account button.
6. The application redisplays the login form and the user logs in.

The files that participate in the authenticate user use case are shown in Figure 12-14.
Let’s see how the files shown in Figure 12-14 are used to implement the authenticate user use case. First, users are directed to the checkout page when they activate the storefront’s Checkout button, as discussed in “The Checkout” on page 410. That checkout page is listed in Example 12-10.a.
Example 12-10.a  /WEB-INF/jsp/checkout/page.jsp

```jsp
<%@ taglib uri='security' prefix='security'%>
<%@ taglib uri='regions' prefix='region' %>

<security:enforceLogin
    loginPage='/WEB-INF/jsp/login/page.jsp'
    errorPage='/WEB-INF/jsp/loginFailed/page.jsp'/>

<region:render region='CHECKOUT_REGION'/>
```

The `security:enforceLogin` tag used in Example 12-10.a is listed in Example 12-10.b.

Example 12-10.b  /WEB-INF/jsp/classes/tags/security/EnforceLoginTag.java

```java
package tags.security;
...
public class EnforceLoginTag extends TagSupport
    implements Constants {
    private String loginPage, errorPage;
    ...
    public int doEndTag() throws JspException {
        HttpSession session = pageContext.getSession();
        HttpServletRequest req = (HttpServletRequest)pageContext.getRequest();
        String protectedPage = req.getServletPath();

        if (session.getAttribute(USER_KEY) == null) {
            session.setAttribute(LOGIN_PAGE_KEY, loginPage);
            session.setAttribute(ERP_ERROR_PAGE_KEY, errorPage);
            session.setAttribute(PROTECTED_PAGE_KEY, protectedPage);

            try {
                pageContext.forward(loginPage);
                return SKIP_PAGE;
            }
            catch(Exception ex) {
                throw new JspException(ex.getMessage());
            }
        }
        return EVAL_PAGE;
    }
    ...
}
```
The `security:enforceLogin` tag checks to see if there is a `User` object in session scope; if so, the rest of the checkout page is evaluated; otherwise, that tag forwards to the login page. The content for the login page is listed in Example 12-10.c.

**Example 12-10.c** /WEB-INF/jsp/login/content.jsp

```jsp
<%@ page contentType='text/html; charset=UTF-8' %>
<%@ taglib uri='i18n' prefix='i18n' %>

<font size='5' color='blue'>
<i18n:message key='login.form.title'/>
</font><hr>

<jsp:include page='form.jsp' flush='true'/>
<jsp:include page='../shared/createAccountLink.jsp' flush='true'/>
```

The JSP page listed in Example 12-10.c includes two other JSP files. The first file generates the login form and is a file of its own because it’s used by another JSP page in the application. That form JSP page is listed in Example 12-10.k on page 453.

The second JSP page included by the login content page is `createAccountLink.jsp`, which is listed in Example 12-10.d.

**Example 12-10.d** /WEB-INF/jsp/shared/createAccountLink.jsp

```jsp
<%@ taglib uri='i18n' prefix='i18n' %>
<%@ taglib uri='utilities' prefix='util' %>

<i18n:message base='app' key='click'/>
<a href='<util:encodeURL url="query-account-action.do"/>'>
<i18n:message base='app' key='here'/></a>
<i18n:message base='app' key='password.hint.toOpenAccount'/>
```

`createAccountLink.jsp` provides a link that forwards to the query account action. Figure 12-15 shows the login page.
The link in `createAccountLink.jsp` listed in Example 12-10.d submits a `query-account-action.do` request that maps to `QueryAccountAction`. That action is listed in Example 12-10.e.

**Example 12-10.e**  
`/WEB-INF/classes/QueryAccountAction.java`

```java
package actions;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import action.ActionBase;
import action.ActionRouter;
```

import `action.ActionBase`;
import `action.ActionRouter`;

**Figure 12-15 The Login Page**

The link in `createAccountLink.jsp` listed in Example 12-10.d submits a `query-account-action.do` request that maps to `QueryAccountAction`. That action is listed in Example 12-10.e.
public class QueryAccountAction extends ActionBase {
    public QueryAccountAction() {
        // this action forwards to a JSP page with sensitive forms
        hasSensitiveForms = true;
    }
    public ActionRouter perform(HttpServlet servlet,
                                 HttpServletRequest req,
                                 HttpServletResponse res)
        throws ServletException {
        return new ActionRouter("query-account-page");
    }
}

The `QueryAccountAction.perform` method does nothing other than forward
to the `query-account-page`. That action class exists because it forwards to a
JSP page that has sensitive forms, which is evident from the action’s constructor,
which sets `hasSensitiveForms` (an inherited member variable) to `true`. That
designation is used to trap sensitive form resubmissions; see “Sensitive Form
Resubmissions” on page 464 for more information about how that trapping is
accomplished.

The `query-account-page` maps to
`/WEBINF/jsp/createAccount/page.jsp`. The main content for that page is
supplied by `/WEB-INF/jsp/createAccount/content.jsp`, which is listed
in Example 12-10.f.

Example 12-10.f  `/WEB-INF/jsp/createAccount/content.jsp`

```jsp
<%@ page contentType='text/html; charset=UTF-8' %>
...

<form action='validate-account.jsp' method='post' >
<table width='450'><tr>
    ...  
    the rest of this table listing is omitted for brevity
    ...
</table>
<br>
<input type='submit' value='create account'>
</form>
```

The JSP page listed in Example 12-10.f contains a rather lengthy table definition,
which is truncated in the interests of brevity. Figure 12-16 shows the JSP page
listed in Example 12-10.f.

8. That table is listed in its entirety in Example 12-11.a on page 455.
The table shown in Figure 12-16 resides in a form whose action is `validateaccount.jsp`, so when the create account button is activated, the browser forwards to that JSP page. That JSP page is listed in Example 12-10.g.
Advanced JavaServer Pages

Example 12-10.g /validate-account.jsp

```jsp
<jsp:useBean id='form' scope='request'
    class='beans.app.forms.CreateAccountForm'/>

<jsp:setProperty name='form' property='*'/>

<jsp:forward page='/validate-account-action.do'/>
```

The JSP page listed in Example 12-10.g creates a `CreateAccountForm` bean and populates that bean with information from the form shown in Figure 12-16. That bean is used to validate the form; see “HTML Forms” on page 455 for more information on exactly how that validation is accomplished. Subsequently, the JSP page listed in Example 12-10.g forwards to `validate-account-action.do`, which maps to the action class listed in Example 12-10.h.

Example 12-10.h /WEB-INF/classes/actions/ValidateAccountAction.java (partial listing)

```java
package actions;
...
public class ValidateAccountAction extends ActionBase
    implements beans.app.Constants {
    public ActionRouter perform(HttpServlet servlet,
        HttpServletRequest req,
        HttpServletResponse res)
        throws ServletException {
        CreateAccountForm form = (CreateAccountForm)
            req.getAttribute("form");
        ...
        boolean errorDetected = false;
        ...
        return errorDetected ?
            new ActionRouter("query-account-page") :
            new ActionRouter("/new-account-action.do", true, false);
    }
}
```

`ValidateAccountAction` checks to see if the create account form was filled in properly; if not, that action forwards to the `query-account-page`, which redisplay the form shown in Figure 12-16 with appropriate error messages. If the form was filled in properly, that action forwards to `new-account-action.do`, which maps to the action class listed in Example 12-10.i.
Example 12-10.i  /WEB-INF/classes/actions/NewAccountAction.java

```
package actions;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import beans.app.User;
import beans.app.Users;
import action.ActionBase;
import action.ActionRouter;

public class NewAccountAction extends ActionBase
    implements beans.app.Constants {
    public NewAccountAction() {
        isSensitive = true; // this is a sensitive action
    }
    public ActionRouter perform(HttpServlet servlet,
        HttpServletRequest req,
        HttpServletResponse res)
        throws ServletException {
        Users users = (Users)servlet.getServletContext().
            getAttribute(USERS_KEY);
        if(users == null) {
            throw new ServletException("Users not found " +
                "in application scope");
        }
        HttpSession session = req.getSession();
        String  firstName = req.getParameter("firstName");
        String  lastName  = req.getParameter("lastName");
        String  address   = req.getParameter("address");
        String  city      = req.getParameter("city");
        String  state     = req.getParameter("state");
        String  country   = req.getParameter("country");
        String  creditCardType = req.getParameter("creditCardType");
        String  creditCardNumber =
            req.getParameter("creditCardNumber");
        String  creditCardExpiration =
            req.getParameter("creditCardExpiration");
    }
```
String  uname = req.getParameter("userName");
String  pwd   = req.getParameter("password");
String  pwdConfirm = req.getParameter("pwdConfirm");
String  pwdHint    = req.getParameter("pwdHint");

session.setAttribute(USERNAME_KEY, uname);
session.setAttribute(PASSWORD_KEY, pwd);
users.addUser(
    new User(firstName, lastName, address, city, state, 
country, creditCardType, creditCardNumber, 
creditCardExpiration, uname, pwd, pwdHint, 
"customer"); // customer is a role

req.setAttribute(USERNAME_KEY, uname);
return new ActionRouter("account-created-page");
};

The new account action listed in Example 12-10.i checks to make sure that there is 
a collection of users present in application scope; if not, that action throws a 
servlet exception.

If the collection of users is present in application scope, the new account action 
creates a new user from the information specified in the create account form and 
adds that user to the collection of users in application scope. That action also 
stores the username and password in application scope, where they are 
subsequently retrieved by the login form listed in Example 12-10.k. That action 
then forwards to the account-created-page. The content for that page is 
listed in Example 12-10.j.

Example 12-10.j /WEB-INF/jsp/accountCreated/content.jsp

<%@ page contentType='text/html; charset=UTF-8' %>
<%@ taglib uri='i18n' prefix='i18n' %>

<font size='4' color='blue'>
<i18n:message base='app' key='accountCreated.text'/>
</font>

<p><jsp:include page='../login/form.jsp' flush='true'/></p>

The JSP page listed in Example 12-10.j displays a message stating that a new 
account has been created and includes the login form JSP page. That page is listed 
in Example 12-10.k.
The login form listed in Example 12-10.k is unremarkable, except that it populates the username and password fields with information stored in session scope so the user does not have to retype them. That username and password were placed in session scope by the new-account-action, which is listed in Example 12-10.i.

The login form’s action is authenticate, which is mapped in web.xml to AppAuthenticateServlet. That servlet is listed in Example 12-10.l.

Example 12-10.l  /WEB-INF/classes/AppAuthenticateServlet.java

```java
import javax.servlet.ServletContext;

import beans.app.Users;
import beans.app.User;

public class AppAuthenticateServlet extends AuthenticateServlet
    implements beans.app.Constants,
        tags.jdbc.Constants {
```
public Object getUser(String username, String password) {
    ServletContext ctx = getServletContext();
    Users users = (Users)ctx.getAttribute(USERS_KEY);
    return users.getUser(username, password);
}

AppAuthenticateServlet extends the abstract AuthenticateServlet class and implements the getUser method, which returns a User object if a user exists with the specified username and password; otherwise, that method returns null. The AuthenticateServlet class is listed in Example 12-10.m.

Example 12-10.m /WEB-INF/classes/AuthenticateServlet.java

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import java.io.IOException;
public abstract class AuthenticateServlet extends HttpServlet
    implements beans.app.Constants,
                tags.security.Constants {
    abstract Object getUser(String username, String pwd);

    public void service(HttpServletRequest req,
                        HttpServletResponse res)
                        throws IOException, ServletException {
        HttpSession session = req.getSession();
        String      uname   = req.getParameter("userName");
        String      pwd     = req.getParameter("password");
        Object      user    = getUser(uname, pwd);

        session.setAttribute(USERNAME_KEY, uname);
        session.setAttribute(PASSWORD_KEY, pwd);

        if(user == null) { // not authorized
            String loginPage = (String)session.
                getAttribute(LOGIN_PAGE_KEY);
            String errorPage = (String)session.
                getAttribute(ERROR_PAGE_KEY);
            String forwardTo = errorPage != null ? errorPage :
                                loginPage;
            session.setAttribute(LOGIN_ERROR_KEY,
                                 "Username: " + uname + " and " +
                                 "Password: " + pwd + " are not valid.");
        }
    }
getServletContext().getRequestDispatcher(
    res.encodeURL(forwardTo)).forward(req,res);
}
else { // authorized
    String protectedPage = (String)session.
        getAttribute(PROTECTED_PAGE_KEY);
    session.removeAttribute(LOGIN_PAGE_KEY);
    session.removeAttribute(ERROR_PAGE_KEY);
    session.removeAttribute(PROTECTED_PAGE_KEY);
    session.removeAttribute(LOGIN_ERROR_KEY);
    session.setAttribute(USER_KEY, user);

    getServletContext().getRequestDispatcher(
        res.encodeURL(protectedPage)).forward(req,res);
}
}

The AuthenticateServlet class listed above is similar to the class of the same
name discussed in “A Model 2 Framework” on page 154, except that the servlet
listed in Example 12-10.m defines an abstract method for looking up a user. That
servlet’s service method stores the new user in session scope and forwards to
the page that was protected in first place; in this case, that page is
/WEBINF/jsp/checkout/page.jsp, which is listed in Example 12-10.a on
page 445. Now that a User object exists in session scope, the entire checkout page
is evaluated and the checkout region is displayed.

HTML Forms

It is vital for web applications to validate HTML forms. If a form is filled out
incorrectly, web applications should redisplay that form with appropriate error
messages. Those topics and more are covered in “HTML Forms” on page 60. This
section discusses how the fruitstand application handles forms submissions for
the create account form, which is the most complicated form in the application.
That form is shown in Figure 12-16 on page 449 and is listed in its entirety in
Example 12-11.a.

Example 12-11.a /WEB-INF/jsp/createAccount/content.jsp
<%@ page contentType='text/html; charset=UTF-8' %>
<%@ taglib uri='application' prefix='app' %>
<%@ taglib uri='i18n' prefix='i18n' %>
<%@ taglib uri='tokens' prefix='tokens' %>
<%@ taglib uri='utilities' prefix='util' %>

Advanced JavaServer Pages

```jsp
<jsp:useBean id='form' scope='request' 
    class='beans.app.forms.CreateAccountForm'/>

<font size='5' color='blue'><u>
    <i18n:message key='createAccount.header.title'/></u>
</font>

<p>
<% String error = (String)request.getAttribute(
    "createAccount-form-error");
if(error != null) { %>
    <font size='5' color='red'>
        <i18n:message key='createAccount.error.fix'/>
    </font>
    <%= error %></p></font>
<%    request.removeAttribute("createAccount-form-error"); %>
</p>

<form action='validate-account.jsp' method='post' >
<table width='450'><tr>
    <td colspan='2'><font size='4' color='blue'>
        <i18n:message key='createAccount.header.personal'/></font></td></tr><tr height='10'></tr>
<tr><td>
    <i18n:message key='createAccount.field.firstName'/></td>
    <td><input type='text' name='firstName'
        value='<%= form.getFirstName() %>'/> 
    </td></tr>
<tr><td>
    <i18n:message key='createAccount.field.lastName'/></td>
    <td><input type='text' name='lastName'
        value='<%= form.getLastName() %>'/> 
    </td></tr>
<tr><td>
    <i18n:message key='createAccount.field.address'/></td>
    <td><input type='text' name='address' size='39'
        value='<%= form.getAddress() %>'/> 
    </td></tr>
<tr><td>
    <i18n:message key='createAccount.field.city'/></td>
    <td><input type='text' name='city'
        value='<%= form.getCity() %>'/> 
    </td></tr>
```

```
<i8n:message key='createAccount.field.state'/></td><td><input type='text' name='state' value='<%= form.getState() %>'/></td></tr><tr><td><i8n:message key='createAccount.field.country'/></td><td><select name='country'></select></td></tr><tr><td><i8n:message key='createAccount.field.phone'/></td><td><input type='text' name='phone' value='<%= form.getState() %>'/></td></tr><tr height='20'></tr><tr><td colspan='2'><font size='4' color='blue'><i8n:message key='createAccount.header.credit'/></font></td></tr><tr height='10'></tr><tr><td><i8n:message key='createAccount.field.creditCardType'/></td><td><select name='creditCardType'></select></td></tr>
Advanced JavaServer Pages

```html
<option <%= form.getCreditCardTypeSelectionAttr("Visa")%>>
  Visa
</select></td></tr>
<tr><td>
<i18n:message
  key='createAccount.field.creditCardNumber'/></td>
<td><input type='text' name='creditCardNumber'
  value='<%= form.getCreditCardNumber() %>'/>
</td>
</tr></tr>
<tr><td>
<i18n:message
  key='createAccount.field.creditCardExpiration'/></td>
<td><input type='text' name='creditCardExpiration'
  value='<%= form.getCreditCardExpiration() %>'/>
</td>
</tr>
<tr height='20'></tr>
<tr><td colspan='2'><font size='4' color='blue'>
<i18n:message
  key='createAccount.header.unameAndPwd'/></td>
</font></td></tr><tr height='10'></tr>
<tr><td>
<i18n:message
  key='createAccount.field.username'/></td>
<td><input type='text' name='userName'
  value='<%= form.getUserName() %>'/></td>
</tr>
<tr><td>
<i18n:message
  key='createAccount.field.password'/></td>
<td><input type='password' name='password' size='8'
  value='<%= form.getPassword() %>'/></td>
</tr>
<tr><td>
<i18n:message
  key='createAccount.field.pwdConfirm'/></td>
<td><input type='password' name='pwdConfirm' size='8'
  value='<%= form.getPwdConfirm() %>'/></td>
</tr><tr><td>
<i18n:message
  key='createAccount.field.pwdHint'/></td>
</tr>
```
When the JSP page listed in Example 12-11.a is displayed, it accesses a CreateAccountForm bean from request scope. If the form was not previously filled out correctly, the JSP page listed in Example 12-11.a uses that bean to populate fields so the user does not have to retype them.

Next, the JSP page listed in Example 12-11.a checks to see if there is an error named createAccount-form-error in request scope; if so, that error is displayed above the form.

The action associated with the login form is a JSP page that creates the CreateAccountForm bean and initializes that bean's values according to the information that was entered into the form. That JSP page is listed in Example 12-10.g on page 450; it is relisted in Example 12-11.b for convenience.

Example 12-11.b /validate-account.jsp

```jsp
<jsp:useBean id='form' scope='request' class='beans.app.forms.CreateAccountForm'/>

<jsp:setProperty name='form' property='*' />

<jsp:forward page='/validate-account-action.do'/>
```

The JSP page listed in Example 12-11.c forwards to validate-account-action.do, which maps to the action class listed in Example 12-11.c.

Example 12-11.c /WEB-INF/classes/actions/ValidateAccountAction.java

```java
package actions;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import action.ActionBase;
```
import action.ActionRouter;
import beans.app.forms.CreateAccountForm;

public class ValidateAccountAction extends ActionBase implements beans.app.Constants {
    public ActionRouter perform(HttpServlet servlet,
                                 HttpServletRequest req,
                                 HttpServletResponse res)
                              throws ServletException {
        CreateAccountForm form = (CreateAccountForm)
                                req.getAttribute("form");
        if(form == null) {
            throw new ServletException("Can't find form");
        }
        String  errMsg;
        boolean errorDetected = false;
        if(!form.validate()) {
            errMsg = form.getValidationError();
            errorDetected = true;
            req.setAttribute("createAccount-form-error", errMsg);
        }
        return errorDetected ?
                           new ActionRouter("query-account-page") :
                           new ActionRouter("/new-account-action.do", true, false);
    }
}

The action class listed in Example 12-11.c retrieves the form from request scope and invokes its validate method. If that method returns false, meaning the form is invalid, the validate account action forwards control to the query-account-page, which is listed in Example 12-11.a on page 455.

The CreateAccountForm bean is listed in Example 12-11.d.

Example 12-11.d /WEB-INF/classes/beans/app/forms/CreateAccountForm.java

package beans.app.forms;

import beans.html.NameElement;
import beans.html.OptionsElement;
import beans.html.TextElement;
import beans.html.ValidatedElement;

public class CreateAccountForm implements ValidatedElement {
    private NameElement firstName = new NameElement("First Name");
private NameElement lastName = new NameElement("Last Name");
private TextElement address = new TextElement();
private TextElement city = new TextElement();
private TextElement state = new TextElement();
private TextElement phone = new TextElement();
private TextElement creditCardNumber = new TextElement();
private TextElement creditCardExpiration = new TextElement();
private TextElement userName = new TextElement();
private TextElement password = new TextElement();
private TextElement pwdConfirm = new TextElement();
private TextElement pwdHint = new TextElement();
private OptionsElement creditCardType = new OptionsElement();
private OptionsElement country = new OptionsElement();
private String error = "";

public String getFirstName() { return firstName.getValue(); }
public String getLastName() { return lastName.getValue(); }
public String getAddress() { return address.getValue(); }
public String getCity() { return city.getValue(); }
public String getState() { return state.getValue(); }
public String[] getCountry() { return country.getValue(); }
public String getPhone() { return phone.getValue(); }
public String[] getCreditCardType() { return creditCardType.getValue(); }
public String getCreditCardNumber() { return creditCardNumber.getValue(); }
public String getCreditCardExpiration() { return creditCardExpiration.getValue(); }
public String getUserName() { return userName.getValue(); }
public String getPassword() { return password.getValue(); }
public String getPwdConfirm() { return pwdConfirm.getValue(); }
public String getPwdHint() { return pwdHint.getValue(); }

public String getCountrySelectionAttr(String s) { return country.selectionAttr(s); }
public String getCreditCardTypeSelectionAttr(String s) { return creditCardType.selectionAttr(s); }

public void setFirstName(String s) { firstName.setValue(s); }
public void setLastName(String s) { lastName.setValue(s); }
public void setAddress(String s) { address.setValue(s); }
public void setCity(String s) { city.setValue(s); }
public void setState(String s) { state.setValue(s); }
public void setCountry(String[] s) { country.setValue(s); }
public void setPhone(String s) { phone.setValue(s); }
public void setCreditCardType(String[] s) { creditCardType.setValue(s); }
public void setCreditCardNumber(String s) { creditCardNumber.setValue(s); }
public void setCreditCardExpiration(String s) { creditCardExpiration.setValue(s); }
public void setUserName(String s) { userName.setValue(s); }
public void setPassword(String s) { password.setValue(s); }
public void setPwdConfirm(String s) { pwdConfirm.setValue(s); }
public void setPwdHint(String s) { pwdHint.setValue(s); }

public boolean validate() {
    error = "";
    if(!firstName.validate()) {
        error += firstName.getValidationError();
    }
    if(!lastName.validate()) {
        if(error.length() > 0)
            error += "<br>");
        error += lastName.getValidationError();
    }
    return error == "";
}

public String getValidationError() {
    return error;
}

The bean listed in Example 12-11.d is rather lengthy because of the size of its associated form, but it’s a simple class that stores form values and provides access to those values. Additionally, that bean implements a validate method that validates the first and last name. That validate method should be expanded to validate all the fields in the form, but it focuses solely on the first and last name fields for simplicity and brevity.

The bean listed in Example 12-11.d uses NameElement instances for the first and last name fields. The NameElement class is discussed in “HTML Forms” on page 60; it is relisted in Example 12-11.e for convenience.
Example 12-11.e  /WEB-INF/classes/beans/html/NameElement.java

package beans.html;

public class NameElement extends TextElement {
    String error, fieldName;

    public NameElement(String fieldName) {
        this.fieldName = fieldName;
    }

    public boolean validate() {
        boolean valid = true;
        String value = getValue();

        error = "";

        if(value.length() == 0) {
            valid = false;
            error = fieldName + " must be filled in";
        } else {
            for(int i=0; i < value.length(); ++i) {
                char c = value.charAt(i);

                if(c == ' ' || (c > '0' && c < '9')) {
                    valid = false;
                    if(c == ' ')
                        error = fieldName + " cannot contain spaces";
                    else
                        error = fieldName + " cannot contain digits";
                }
            }
        }

        return valid;
    }

    public String getValidationError() {
        return error;
    }
}

The NameElement class extends TextElement and overrides the validate method to disallow spaces and digits in a name field. See “HTML Forms” on page 60 for more information concerning the NameElement and TextElement classes.
Sensitive Form Resubmissions

All web applications should guard against sensitive form resubmissions, as discussed in “Event Handling and Sensitive Form Resubmissions” on page 182. This section shows how the fruitstand application guards against resubmitting the create account form listed in Example 12-11.a on page 455.

First, the QueryAccountAction class is designated as an action that forwards to a JSP page with sensitive forms, as you can see from the partial listing of that class below.

```java
package actions;
...
public class QueryAccountAction extends ActionBase {
    public QueryAccountAction() {
        // this action forwards to a JSP page with sensitive forms
        hasSensitiveForms = true;
    }
    ...
}
```

After an action that has sensitive forms—such as QueryAccountAction—is performed, the Model 2 framework creates two identical tokens (strings). One of those tokens is placed in session scope and the other is placed in request scope. Subsequently, before a sensitive action is performed, the Model 2 framework checks to see that both tokens are present and identical. If that requirement is not met, the framework throws an exception; otherwise, it performs the sensitive action. That mechanism guards against resubmitting sensitive forms through a bookmark or the Back button.

The NewAccountAction class, which creates a new account, is designated as a sensitive action, as illustrated by the partial listing below.

```java
package actions;
...
public class NewAccountAction extends ActionBase implements beans.app.Constants {
    public NewAccountAction() {
        isSensitive = true; // this is a sensitive action
    }
    ...
}
```

Finally, the create account form, which is partially listed in Example 12-11.f, uses the tokens:token custom tag that copies the existing token in request scope to the request generated by the form submission.
Example 12-11.f  /WEB-INF/jsp/createAccount/content.jsp (partial listing)

```jsp
<%@ page contentType='text/html; charset=UTF-8' %>
...
<%@ taglib uri='tokens' prefix='tokens' %>
...
<form action='validate-account.jsp' method='post' >
...
  <tokens:token/>
</form>

SSL

In addition to guarding against sensitive form resubmissions, web applications should also use SSL when transporting confidential information, such as credit card numbers.

Two steps are involved in using SSL. First, not all servlet containers support SSL out of the box. Fortunately, it's usually a simple matter to add SSL support. For Tomcat, adding SSL support is well documented; see $TOMCAT_HOME/doc/tomcat-ssl-howto.html for Tomcat 3.2 final.

Second, you must specify the resources—typically, JSP pages—that require SSL. Those resources are specified in an application's web.xml file; for example, the fruitstand application can specify that the create account page, which transmits a credit card number, requires SSL with the following addition to web.xml:

```xml
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Credit Card Page</web-resource-name>
    <url-pattern>/WEB-INF/jsp/createAccount/content.jsp</url-pattern>
  </web-resource-collection>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

The security constraint listed above specifies that SSL should be used to access /WEB-INF/jsp/createAccount/content.jsp. See “Security” on page 250 for more information concerning security constraints.
Unfortunately, when this book was written neither Tomcat 3.2 final or Tomcat 4.0 implemented this feature correctly. By the time you read this, Tomcat 4.0 should work properly.

XML and DOM

It’s often beneficial for web applications to handle data internally as XML, for two reasons. First, data can be stored in a Document Object Model (DOM), which is a standard data structure that is widely accepted and for which numerous tools are available. Second, XML is quickly becoming the de facto standard for transferring data from one business to another.

Using XML with JSP has already been covered extensively in “XML” on page 330. This section discusses using XML for the fruitstand’s inventory.

The storefront content page is responsible for reading the fruitstand’s inventory from a database, as discussed in “The Fruitstand” on page 392. Example 12-12.a lists a version of the storefront content page modified to use the DOM to store the inventory in XML format.

Example 12-12.a /WEB-INF/jsp/storefront/content.jsp (DOM version)

```jsp
<%@ page contentType='text/html; charset=UTF-8' %>
<%@ taglib uri='i18n' prefix='i18n' %>
<%@ taglib uri='dom' prefix='dom' %>
<%@ taglib uri='html' prefix='html' %>

<font size='4' color='blue'>
  <i18n:message base='app' key='storefront.form.title'/>
</font><p>
<dom:parse id='inventory' scope='application'>
  <%@ include file='inventory-to-xml.jsp' %>
</dom:parse>

<table border='1' cellPadding='3'>
  <th><i18n:message base='app'
    key='storefront.table.header.picture'/></th>
  <th><i18n:message base='app'
    key='storefront.table.header.item'/></th>
  <th><i18n:message base='app'
    key='storefront.table.header.description'/></th>
  <th><i18n:message base='app'
    key='storefront.table.header.price'/></th>
```
<th><i18n:message base='app' key='storefront.table.header.addToCart'/></th>

<% String currentItem = null, currentSku = null; %>
<dom:iterate node='<%=inventory.getDocumentElement()%>' id='item'>
  <dom:iterate node='<%= item %>' id='itemField'>
    <dom:ifNodeNameEquals node='<%= itemField %>' names='SKU'>
      <dom:elementValue id='name' element='<%= itemField %>'/>
    </dom:ifNodeNameEquals>
    <dom:ifNodeNameEquals node='<%= itemField %>' names='NAME'>
      <dom:elementValue id='name' element='<%= itemField %>'/>
    </dom:ifNodeNameEquals>
    <dom:ifNodeNameEquals node='<%= itemField %>' names='PRICE'>
      <dom:elementValue id='price' element='<%= itemField %>'/>
    </dom:ifNodeNameEquals>
  </dom:iterate>
  <tr><td>
    <img src='<%= "graphics/fruit/" + name.trim() + ".jpg" %>'/>
  </td><td>
    <i18n:message key='<%=name + ".description"%>'/>
  </td><td>$&nbsp;<%= price %>&nbsp;/lb.</td><td>
    <form action='add-selection-to-cart-action.do'>
      <html:links name='<%= currentSku + "-" + currentItem + "-" + price %>'>
        <option value='0.00'>0.00</option>
        <option value='1.00'>1.00</option>
        <option value='1.50'>1.50</option>
        <option value='2.00'>2.00</option>
        <option value='2.50'>2.50</option>
        <option value='3.00'>3.00</option>
        <option value='3.50'>3.50</option>
        <option value='4.00'>4.00</option>
        <option value='4.50'>4.50</option>
        <option value='5.00'>5.00</option>
        <option value='5.50'>5.50</option>
      </html:links>
    </form>
  </td>
</dom:iterate>
The JSP page listed in Example 12-12.a uses the DOM custom tags discussed in “DOM Custom Tags” on page 364 to create a DOM document representing the fruitstand’s inventory. The \texttt{dom:parse} tag, by default, only creates that document once, so the JSP page listed in Example 12-12.a will only access the database one time.

The \texttt{dom:parse} tag used in Example 12-12.a interprets its body content as XML. That content is generated by \texttt{inventory-to-xml.jsp}, which is listed in Example 12-12.b.

\textbf{Example 12-12.b} /WEB-INF/jsp/storefront/inventory-to-xml.jsp

```java
<%@ taglib uri='database' prefix='database' %>

<database:query id='inventory' scope='session'>
  SELECT * FROM Inventory
</database:query>

<?xml version="1.0" encoding="ISO-8859-1"?>
<FRUITS>
  <database:rows query='inventory'>
    <ITEM>
      <database:columns query='inventory' columnName='name' columnValue='value'>
      <%= "<" + name + ">" %>
      <%= value %>
      <%= ">
    </database:columns>
    </ITEM>
  </database:rows>
</FRUITS>

<database:release query='inventory'/>
```

The JSP file listed in Example 12-12.b uses the database tags discussed in “Databases” on page 282 to extract information from the database. That JSP file generates XML, which is used by the \texttt{dom:parse} tag in Example 12-12.a on page 466.

It would be a simple matter to modify the JSP file listed in Example 12-12.b to store the generated XML in a file; that modification would make that information readily available to other businesses.
Conclusion

This chapter has presented a nontrivial case study that uses the techniques discussed throughout this book; however, this chapter did not discuss all of the case study’s features. To get the most benefit out of this chapter, you should download the code for this book from www.phptr.com/advjsp and experiment with the case study.
AN OVERVIEW OF J2EE 1.4 WEB SERVICES

Excerpted from: J2EE Web Services by Richard Monson-Haefel
In a nutshell, J2EE Web Services is about interoperability. That goal is perhaps the most important thing to remember while reading this book. Not just interoperability between different brands of J2EE application servers, but between J2EE servers and any other Web services platform, including the .NET Framework, Perl, Apache Axis, Python, and C++, to name a few. Web service technologies provide J2EE with an opportunity to become truly interoperable with any other system. That’s because Web service technologies are platform-agnostic; in other words, the medium used to communicate is not specific to any combination of programming language, operating system, and hardware.

At the heart of J2EE Web Services interoperability, and of Web services interoperability in general, is the Basic Profile 1.0 (BP), published by the Web Services Interoperability Organization (WS-I). The BP provides a set of rules that govern how applications make use of common Web service technologies so that everyone is speaking the same language. The BP makes Web service interoperability practical, and coverage of it is a critical aspect of this book.

Although this book assumes you will want to develop Web services that comply with WS-I, only J2EE 1.4 vendors are required to support the BP. It’s not mandatory that you use it. In fact, J2EE 1.4 Web Services APIs are also required to support technologies that don’t conform to the BP, like SOAP Messages with Attachments and RPC/Encoded messaging. These two non-BP-conformant technologies are covered in appendices.

The purpose of this chapter is to provide an architectural overview of the J2EE 1.4 platform, Web service technologies, and the J2EE 1.4 Web Services APIs. Before you read any further, you should read the Preface if you haven’t already, because it provides important information on the scope and organization of this book.
1.1 The J2EE Platform

The J2EE (Java 2 Platform, Enterprise Edition) specification describes how a variety of enterprise Java APIs are integrated into a complete platform. Specifically, the J2EE 1.4 specification tells us how an application server configures and deploys Enterprise JavaBeans (EJB) 2.1, Java Servlets 2.4, and JavaServer Pages (JSP) 2.0 components, and manages them at runtime. The J2EE specification also tells us how these server-side components interact with each other, the application server, and resource APIs like Java Database Connectivity (JDBC) and Java Message Service (JMS). Figure 1–1 shows the various components and APIs supported by the J2EE specification.

Figure 1–1 doesn’t show a lot of detail, but it does give you an overview of the J2EE architecture. The EJB and servlet/JSP components reside in their own containers, which manage their life cycles, incoming and outgoing messages, transactions, security, and other qualities of service. EJB and Servlet/JSP components have access to all of the resource APIs, such as Web Services, J2EE Connectors (including JDBC), JMS, JavaMail, Security, Transactions, and J2EE Management APIs. The entire platform is built on the Java 2 Platform, Standard Edition (J2SE) 1.4, and runs in one or more Java Virtual Machines.

The J2EE specification depends on several other specifications. For example, the programming model and behavior of the EJB components are described in a separate EJB specification; the same is true of the servlet and JSP components as well as the J2EE Connector, JDBC, JMS, JavaMail, JMX, and other resource APIs. It’s the job
of the J2EE specification to glue all of these specifications together. As I mentioned in the Preface, this book doesn’t go into detail about J2EE components or APIs other than those related to Web services—the J2EE platform is simply too big. That said, this book does cover the J2EE Web Services components and APIs in detail.

The Web services components and APIs are actually described by the Web Services for J2EE 1.1 specification. WS-J2EE 1.1 requires compliance with the WS-I Basic Profile 1.0. The BP is discussed later in this chapter.

The primary goal of J2EE is portability of components and developer skills. Because every J2EE vendor must support exactly the same server-side component models (EJBs, servlets, and JSPs) you can expect the components that you developed for J2EE brand A to run on J2EE brand B—with a little bit of work. Unfortunately, component portability in J2EE is not as simple as originally envisioned because each J2EE vendor provides very different configuration and deployment facilities. That said, portability in J2EE is far simpler than in any of J2EE’s predecessors, including CORBA and DCE components.

In addition to component portability, J2EE offers developer portability. A developer who learns to develop components on one vendor’s J2EE platform can easily transfer those skills to another vendor’s. Of course, the developer may have to learn new procedures for deployment and server administration, but otherwise the J2EE developer’s skills are applicable to any brand of J2EE.

While portability has been the main focus of J2EE, interoperability, specifically the ability of one brand of J2EE application server to talk to another, has recently become an important focus as well. In the past interoperability was based on the CORBA IIOP protocol, which is fine in theory but in practice is difficult to get working properly. CORBA interoperability has never been easy with pure CORBA products, and that trend continues with J2EE products.

As of J2EE 1.4, however, interoperability takes a new shape in the form of Web services, the topic of this book. Specifically, this book covers the four primary Web service standards (XML, SOAP, WSDL, and UDDI) and the J2EE 1.4 Web Services APIs (JAX-RPC, SAAJ, JAXR, and JAXP). The next two sections provide an overview of Web services, the Web service standards, and the J2EE Web Services APIs.

1.2 The Technologies of Web Services

The term Web services has many different definitions, depending on your perspective. Some definitions that have been proposed by the media and Web services experts are so ambiguous as to be almost useless. This is not the fault of the people who have tried to define the term. They are simply attempting to verbalize something that has, until recently, been a somewhat sketchy idea.

In the J2EE 1.4 environment, however, the term “Web services” has a very specific meaning, which is based on a standard adopted by Sun Microsystems, Microsoft,
IBM, BEA, Software AG, and just about every other major enterprise vendor. In a nutshell, Web services can be defined as follows:

*A Web service is a software application that conforms to the Web Service Interoperability Organization’s Basic Profile 1.0.*

That’s a bit of a mouthful, but it’s probably the most exact definition of the term you will encounter today. Before the WS-I defined the Basic Profile 1.0, the term “Web service” was simply too general to fulfill the technology’s main purpose, interoperability. In other words, the main purpose of Web service technologies is to allow applications on different platforms to exchange business data. Web service technologies are used for Application-to-Application (A2A) integration or Business-to-Business (B2B) communication. A2A refers to disparate applications within a single organization communicating and exchanging data—A2A is also known as Enterprise Application Integration (EAI). B2B refers to multiple organizations, typically business partners, exchanging data. Web service technologies today are used mostly in A2A/EAI settings, but they are also seeing growth in the B2B arena.

For a computer programmer, interoperability is the capability of two different software applications to communicate. For example, the ability to access an application written in C++ running on Windows from a Java application running on Linux is a matter of interoperability. In order for a Java application on a Linux platform to access a C++ application on a Windows platform, you have to use network technologies that are independent of both the operating system and the hardware. This capability is already met by TCP/IP, DNS, and HTTP, and the Web service standards: XML, SOAP, WSDL, and UDDI.

XML (eXtensible Markup Language), SOAP (Simple Object Access Protocol), WSDL (Web Services Description Language), and UDDI (Universal Description, Discovery, and Integration) are used in concert to provide Web service applications with a type system (XML), a messaging protocol (SOAP), an interface definition language (WSDL), and a registry for publishing Web services (UDDI). XML documents contain the information being exchanged between two parties, while SOAP provides a packaging and routing standard for exchanging XML documents over a network. WSDL allows an organization to describe the types of XML documents and SOAP messages that must be used to interact with their Web services. Finally, UDDI allows organizations to register their Web services in a uniform manner within a common directory, so clients can locate their Web services and learn how to access them. Figure 1–2 shows how these pieces fit together in a Web service interaction between two parties.

The first three parts of this book cover XML, SOAP, WSDL, and UDDI. These core Web services technologies are also the basis of J2EE 1.4 Web Services. Each of the J2EE Web Services APIs (JAX-RPC, SAAJ, JAXR, JAXP) conforms with one or more of the Web services technologies, so developing a fundamental understanding of XML, SOAP, WSDL, and UDDI is crucial to understanding how J2EE 1.4 Web Services works. In addition to the specifications of XML, SOAP, WSDL, and UDDI,
this book covers the WS-I Basic Profile 1.0, which is necessary to ensure interoperability across a broad range of platforms. The rest of this section provides an overview of the WS-I Basic Profile 1.0, XML 1.0, SOAP 1.1, WSDL 1.1, and UDDI 2.0. These topics are covered in more detail in Parts I through III.

1.2.1 WS-I Basic Profile 1.0

The Web Services Interoperability Organization is an organization of Web services vendors that are committed to defining a standard for Web services interoperability. The first deliverable of the WS-I was the Basic Profile 1.0, which details how to use the four primary Web services specifications together in Web services. The BP defines a set of conformance rules that clear up ambiguities in the specifications of XML, WSDL, SOAP, and UDDI, and defines in concrete terms how to use these technologies in concert to register, describe, and communicate with Web services.

The BP is necessary because the primary specifications are too broad and open to interpretation to make interoperability a given. For example, WSDL is a very
Chapter 1
An Overview of J2EE 1.4 Web Services

generalized technology, which allows you to describe any kind of Web service. Unfortunately, WSDL is so general that it’s difficult in some circumstances to determine exactly how, for example, a SOAP message exchanged with the Web service should be formatted. In addition, WSDL defines features that have, in practice, made interoperability more difficult, such as the SMTP and MIME bindings. The BP fixes these interoperability problems by telling us exactly how WSDL should describe SOAP-based Web services, and by restricting the use of WSDL.

No one is required to adhere to the BP, but doing so makes interoperability a lot easier. When you encounter a Web service that conforms with the BP, you have a much easier time locating and communicating with that Web service than you have with a non-conformant Web service. To be perfectly honest, interoperability without the BP is pretty difficult to achieve. While XML, SOAP, WSDL, and UDDI have important roles, their use together is not well coordinated because each specification was developed independently. The BP defines the glue that binds these technologies together into one coherent specification.

Microsoft, IBM, Sun Microsystems, BEA, and many others have agreed to support the BP, which means you should be able to communicate with WS-I-conformant Web services hosted by products from any of these vendors with little or no interoperability problems. The BP makes reality of what was basically a pipe dream in the past; it makes it possible to define applications that are interoperable across most application platforms.

The BP is the specification around which this book revolves, because J2EE 1.4 vendors are required to support it. This book assumes you will want to develop applications that conform to the BP because, if they don’t, general interoperability simply isn’t practical. Unless you have complete control over all the parties involved in a B2B or A2A system, defining a Web service that is interoperable with arbitrary systems is very difficult, if not impossible. The BP makes it possible to define Web services that are easily understood and operated on by any application written in any programming language (Java, C++, C#, Perl, et al.), on any operating system (Unix, Microsoft, Linux, Mac, et al.), using any development platform (the .NET Framework, WebLogic Application Server, WebSphere Application Server, et al.)

Although the BP is important, there is no chapter dedicated to the subject. Instead, this book addresses the requirements of the BP on Web service technologies and J2EE 1.4 Web Service APIs as they come up—directly in the text of each chapter. You’ll know when you are seeing a BP-conformance rule because I will explicitly call it out, or will append the superscript * to the end of a sentence that states a BP-conformance rule.

1.2.2 XML

XML (eXtensible Markup Language) is used to organize documents and business data. XML files can be stored or transmitted between two applications on a network. Basically, they are just plain text documents that contain special tags that
label different parts of a document or fields of data. For example, the XML document in Listing 1–1 contains Amazon.com’s mailing address.

Listing 1–1

Sample XML Document

```xml
<?xml version="1.0" encoding="UTF-8"?>
<address>
  <name>Amazon.com</name>
  <street>1516 2nd Ave</street>
  <city>Seattle</city>
  <state>WA</state>
  <zip>90952</zip>
</address>
```

Notice how each piece of data (name, street, city, state, and zip) is demarcated by a pair of tags, such as `<name>` and `</name>`. This syntax makes it very easy to tell where a field of data begins and ends, and what it represents. The address information in Listing 1–1 might be stored in a text file or sent across the network to another application, perhaps to synchronize mailing information in a database.

XML is the foundation for all the other Web services technologies, so having a good understanding of XML is critical. Part I (Chapters 2 and 3) of this book provides a detailed overview of XML and XML schema, which is the typing system used to validate the contents of XML documents.

Currently there is only one version, XML 1.0, which is managed by the World Wide Web Consortium (W3C). XML is dialect of SGML, which the publishing industry and the U.S. military among others have used for years to provide structure to written documents. Although XML has its roots in SGML, it has greatly surpassed its parent in popularity and usefulness.

1.2.3 SOAP

When we think of Web services, we are basically discussing the exchange of XML documents, or data that is organized into an XML format. Although XML is an excellent technology for exchanging business data, until SOAP came along there was no widely accepted standard for routing and packaging XML data exchanged between two applications on a network. If you wanted to build an EAI or B2B system based on XML, you had to define your own networking, addressing, and routing protocols, or use a proprietary system provided by a vendor. No two vendors were using the same protocols for exchanging data, so interoperability was

---

1 XML 1.1 is currently a W3C candidate recommendation.
difficult, to say the least. Before SOAP, interoperability usually required tedious
custom-built solutions to get any two XML-based systems to communicate.

SOAP (from Simple Object Access Protocol) defines a standard packaging format
for transmitting XML data between applications on a network. It is not specific to
any programming language, product, or hardware platform, so it can be used by just
about any kind of application (C++, Java, .NET, Perl, legacy systems, and others). A
SOAP message is just an XML document. SOAP is specially designed, however, to
contain and transmit other XML documents as well as information related to rout-
ing, processing, security, transactions, and other qualities of service. For example,
Listing 1–2 shows a SOAP message that contains XML address information. This
message might be sent from one application to another to synchronize contact infor-
mation on two different systems.

Listing 1–2
A SOAP Message That Contains Address Information

```xml
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  <soap:Body>
    <addr:address>
      <addr:name>Amazon.com</addr:name>
      <addr:street>1516 2nd Ave</addr:street>
      <addr:city>Seattle</addr:city>
      <addr:state>WA</addr:state>
      <addr:zip>90952</addr:zip>
    </addr:address>
  </soap:Body>
</soap:Envelope>
```

SOAP takes advantage of advanced XML features like XML namespaces (similar
to Java package names) and XML schemas (used to type data), all of which are
covered in detail in Part I of this book. The important thing to understand at this
point is that SOAP messages serve as a network envelope for exchanging XML docu-
ments and data. Having a single industry standard for packaging and exchanging
XML data makes Web service applications more interoperable, more understand-
able, and easier to implement. Part II (Chapters 4 and 5) provides a very detailed
discussion of the SOAP messaging protocol and processing requirements, as
described by the SOAP 1.1 and WS-I Basic Profile 1.0 standards.

There are actually two versions of SOAP today, versions 1.1 and 1.2. SOAP 1.1
was defined as a Note by the W3C in 2000 and has been the basis for most Web serv-
ces development since that time. The BP requires the use of SOAP 1.1 and provides
a number of clarifications and restrictions that largely eliminate interoperability problems associated with this sometimes poorly specified standard. SOAP 1.2 wasn’t finalized until 2003 and, while it represents a significant improvement over SOAP 1.1, adoption of SOAP 1.2 by vendors may take a while because they are currently focused on providing solutions based on SOAP 1.1 and the BP. This book covers only SOAP 1.1—SOAP 1.2 was only in draft form when this book was written, and it’s not supported by J2EE 1.4 Web Services.

There is an ancillary specification called SOAP Messages with Attachments (SwA) that is covered in Appendix E. SwA defines a message format for attaching binary data (images, sound files, documents, and so on) to SOAP messages. Although SwA is not sanctioned by the BP, support for it is required by the J2EE 1.4 Web Services specifications—SwA will also be supported in the next version of the BP, version 1.1, which is still in development.

1.2.4 WSDL

WSDL (Web Services Description Language) is a standard for describing the structure of the XML data exchanged between two systems using SOAP. When you create a new Web service, you can also create a WSDL document that describes the type of data you’re exchanging. WSDL is complicated and unless you’re already familiar with it an example WSDL document will make no sense at all. Suffice it to say that WSDL is based on XML. It’s used to describe the SOAP messages that can be transmitted between Web service clients and servers. WSDL is covered in detail in Part II of this book.

There are two versions of WSDL today, versions 1.1 and 1.2. Like SOAP 1.1, WSDL 1.1 was defined as a Note by the W3C (February 2001) and has been used a lot in Web services development. WSDL 1.1 is a good specification, but its flexibility creates interoperability problems. The BP requires the use of WSDL 1.1, but also provides strict rules on how it’s used, to improve interoperability. WSDL 1.2 wasn’t completed when this book was being written, and is not supported by J2EE 1.4 Web Services, so it’s not covered by this book.

1.2.5 UDDI

Although WSDL provides an excellent format for describing the types of SOAP messages used by a Web service, it provides no guidance on where to store the WSDL documents or how to find them. In other words, WSDL doesn’t describe where the WSDL documents should be kept so that others can find them easily and use them to communicate with your Web services.

UDDI (Universal Description, Discovery, and Integration) defines a standard set of Web service operations (methods) that are used to store and look up information about other Web service applications. In other words, UDDI defines a standard SOAP-based interface for a Web services registry. You can use a UDDI registry to find a particular type of Web service, or to find out about the Web services hosted by a
specific organization. A UDDI registry is often referred to as a “Yellow Pages” for Web services. When you look up information about a Web service in a UDDI registry, you can narrow your search using various categories (technologies used, business types, industry, and so on). Each entry in a UDDI registry provides information on where the Web service is located and how to communicate with it. The UDDI registry also provides information about the organization that hosts a particular Web service.

The UDDI specification is now maintained by the Organization for the Advancement of Structured Information Standards (OASIS), but was originally created by Microsoft and IBM, which led a multi-vendor organization called UDDI.org. The UDDI.org has set up a free UDDI registry open to everyone—something like a Yahoo! for Web services. Most UDDI registries, however, are private systems deployed by individual companies or trade organizations. For example, a very large organization like General Electric might set up a private UDDI registry that contains information about all the Web service applications offered within the corporation. Another example is a trade organization, like the Association of American Manufacturers, which might set up a UDDI registry that contains information about all its members and their Web service applications.

Of the four principal Web services technologies, UDDI is the only one the BP says is optional. You don’t have to use UDDI, but if you need a Web service registry, UDDI is the preferred technology. UDDI is covered in detail in Part III (Chapters 6–8).

There are three versions of UDDI at this time, versions 1.0, 2.0, and 3.0. The BP specifies the use of UDDI version 2.0, the one covered by this book. While version 3.0 is more recent, it was only newly completed when the BP was finalized and so wasn’t supported by enough vendors for the WS-I to consider it.

1.3 The J2EE Web Service APIs

Parts IV through VI of this book cover the four J2EE Web Service APIs: JAX-RPC, SAAJ, JAXR, and JAXP. These are the APIs you will need to understand if you want to implement Web service applications using the J2EE platform. Perhaps the most important Web service API is JAX-RPC, which is used to implement J2EE Web service clients and endpoints (services). A substantial portion of this book (seven chapters) covers JAX-RPC.

1.3.1 JAX-RPC

You can think of JAX-RPC (Java API for XML-based RPC) as Java RMI over SOAP. While this characterization provides a good vantage point for learning about JAX-RPC, it’s not the complete story. You can obtain a good understanding of JAX-RPC only by studying it in detail. Part IV (Chapters 9–15) provides detailed coverage of JAX-RPC.

2 UDDI can also store data about other types of services, such as a Web site or a phone service. This book, however, is concerned only with UDDI’s role in Web services.
JAX-RPC is divided into two parts: a set of client-side APIs, and a set of server-side components, called endpoints. The client-side APIs allow you to communicate with Web service endpoints hosted on some other platform. For example, you can use one of the client-side APIs to send SOAP messages to a VB.NET or an Apache Axis Web service. The client-side APIs can be used from standalone Java applications or from J2EE components like servlets, JSPs, or EJBs. There are three client-side APIs: generated stub, dynamic proxy, and DII (Dynamic Invocation Interface). The generated stub is the one you will use the most, and its semantics closely resemble those of Java RMI. The dynamic proxy API also follows many of the Java RMI semantics, but is used less often. The DII is a very low-level API used primarily by vendor tools, but can also be employed by Web services developers if necessary.

The server-side components include the JAX-RPC service endpoint (JSE) and the EJB endpoint. The JSE component is actually a type of servlet that has been adapted for use as a Web services component. It's very easy to implement, yet it has access to the full array of services and interfaces common to servlets. The EJB endpoint is simply a type of stateless session EJB that has been adapted for use as a Web service endpoint. The EJB endpoint provides all the transactional and security features of a normal stateless session bean, but it's specifically designed to process SOAP requests.

There are currently two versions of JAX-RPC, 1.0 and 1.1. Version 1.0 is used with J2EE 1.3, while version 1.1 is used with J2EE 1.4. The primary difference between 1.0 and 1.1 is that 1.1 is required to conform with the BP. Because this book focuses on BP-conformant applications, version 1.0 is not covered in this book.

1.3.2 SAAJ

SAAJ (SOAP with Attachments API for Java) is a low-level SOAP API that complies with SOAP 1.1 and the SOAP Messages with Attachments specification. SAAJ allows you to build SOAP messages from scratch as well as read and manipulate SOAP messages. You can use it alone to create, transmit, and process SOAP messages, but you're more likely to use it in conjunction with JAX-RPC. In JAX-RPC, SAAJ is used primarily to process SOAP header blocks (the SOAP message meta-data). Chapter 13 covers the SAAJ API in detail.

Although the BP does not support SOAP Messages with Attachments, J2EE 1.4 requires SwA, so it's covered in Appendix F: SAAJ Attachments.

1.3.3 JAXR

JAXR (Java API for XML Registries) provides an API for accessing UDDI registries. It simplifies the process of publishing and searching for Web service endpoints. JAXR was originally intended for ebXML registries, a standard that competes with UDDI, but was adapted for UDDI and works pretty well in most cases.
JAXR has a set of business-domain types like Organization, PostalAddress, and Contact as well as technical-domain types like ServiceBinding, ExternalLink, and Classification. These domain models map nicely to UDDI data types. JAXR also defines APIs for publishing and searching for information in a UDDI registry. There is only one version of JAXR, version 1.0, which is the subject of Part V (Chapters 16–19) of this book.

1.3.4 JAXP

JAXP (Java API for XML Processing) provides a framework for using DOM 2 and SAX2, standard Java APIs that read, write, and modify XML documents.

DOM 2 (Document Object Model, Level 2) is a Java API that models XML documents as trees of objects. It contains objects that represent elements, attributes, values, and so on. DOM 2 is used a lot in situations where speed and memory are not factors, but complex manipulation of XML documents is required. DOM 2 is also the basis of SAAJ 1.1.

SAX2 (Simple API for XML, version 2) is very different in functionality from DOM 2. When a SAX parser reads an XML document, it fires events as it encounters start and end tags, attributes, values, etc. You can register listeners for these events, and they will be notified as the SAX2 parser detects changes in the XML document it is reading.

JAXP comes in several versions including 1.1, 1.2, and 1.3. Version 1.3 is very new and is not supported by J2EE 1.4 Web Services. Instead, J2EE 1.4 requires support for JAXP 1.2, which is the subject of Part VI (Chapters 20 and 21) in this book.

1.4 Wrapping Up

The Web service standards and J2EE 1.4 APIs covered in this book do not represent the full complement of Web service technologies available today. Web services is a fast-growing field, and it seems like a new messaging standard or API is released every week. The Web services technologies this book covers, however, are the ones that are absolutely fundamental to understanding the others. Before you can learn about Web services security, orchestration, transactions, and the like, you must first have a complete and clear understanding of XML, SOAP, WSDL, and to a lesser extent UDDI. This book provides you with an in-depth study of these fundamentals so that you will be prepared to implement interoperable Web service clients and services using the J2EE 1.4 platform. This book also provides you with a solid foundation for learning about new and future standards that have recently been introduced, but have yet to prove themselves in production.

Throughout this book, particular care is given to framing most discussions in the context of the Web Services Interoperability Organization’s Basic Profile 1.0. If you
have worked with Web service technologies in the past and have had a chance to study the BP, then you will probably appreciate why it’s so important. To be completely blunt: I do not believe general Web service interoperability with anonymous parties is actually feasible without the guidance of the WS-I Basic Profile 1.0. Experienced Web service developers can tell you numerous interoperability horror stories related to the loose language and broad visions defined by standards like SOAP, WSDL, and UDDI. The WS-I Basic Profile eliminates many, if not all, of the ambiguities inherent in these primary standards, finally making interoperability among disparate Web service platforms (.NET, J2EE, Apache Axis, Perl, etc.) possible.
It’s important that you have a good understanding of XML, XML namespaces, and the W3C XML Schema Language before proceeding, because these technologies are fundamental to understanding SOAP and WSDL, which are the foundation of Web Services as defined by J2EE and the WS-I Basic Profile 1.0.

While most developers understand and have used basic XML, many have not used XML namespaces or the W3C XML Schema Language, so the emphasis of this first part of the book is on these two topics. Chapter 2 includes two major
sections: Section 2.1: XML Primer, which you can skip if you are already familiar with basic XML concepts, and Section 2.2: XML Namespaces, which you should read if XML is new to you.

Chapter 3 covers the W3C XML Schema Language, which you need to understand before you can understand SOAP, WSDL, and UDDI. If you've never worked with the W3C XML Schema Language before, you should read at least the first section of Chapter 3, Section 3.1: XML Schema Basics. The second section in Chapter 3, Section 3.2: Advanced XML Schema, covers more advanced and somewhat trickier features.

The XML schema is very rich and includes a lot of features not covered by Chapter 3—it actually requires an entire book to cover all the features of the W3C XML Schema Language. The bibliography includes a book recommendation for this topic, but the material in Chapter 3 should satisfy most of your needs.

As a supplement to this part of the book, Appendix A covers XML DTDs (document type definitions). The use of DTDs is very common, which is why it's covered in this book. The W3C XML Schema Language is the preferred mechanism for describing and validating the structure of XML documents in Web services, however, so DTDs are not given much attention other than in Appendix A.

In This Part

<table>
<thead>
<tr>
<th>Chapter 2: XML Basics</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 3: The W3C XML Schema Language</td>
<td>33</td>
</tr>
</tbody>
</table>

Related Appendices

| Appendix A: XML DTDs | 739 |
| Appendix B: XML Schema Regular Expressions | 745 |
CHAPTER 6

SERVICE ENDPOINT DESIGN

Excerpted from: Designing Web Services with the J2EE™ 1.4 Platform: JAX-RPC, SOAP, and XML Technologies (Java Series) by Inderjeet Singh, Sean Brydon, Greg Murray, Vijay Ramachandran, Thierry Violleau, Beth Stearns
WEB services interact with clients to receive the clients’ requests and return responses. In between the request and the response, a Web service applies appropriate business logic to process and fulfill a client’s request. Designing an effective Web service starts with understanding the nature of the service to be provided—that is, how the service is going to interact with the client and how it is going to process and fulfill the client’s request—coupled with how users of the service find and make their requests. This chapter examines Web services from the perspective of a service’s interaction and processing functionality.

The chapter describes the key issues you must consider when designing a Web service, then shows how these considerations drive the design and implementation of a service’s Web service interface and functionality. In particular, the chapter examines the interactions between a service and its clients and the business processing that the service performs. It illustrates these considerations by drawing from examples using three typical Web service scenarios.

The chapter covers most of the decisions that must be made when designing and implementing a Web service, including identifying the different possibilities that give rise to different solutions. It describes how to receive requests, delegate requests to business logic, formulate responses, publish a Web service, and handle document-based interactions.

Along the way, the chapter makes recommendations and offers some guidelines for designing a Web service. These recommendations and key points, marked with check boxes, include discussions of justifications and trade-offs. They are illustrated with the example service scenarios. Since Web services basically expose interoperable interfaces for new as well as existing applications, a large segment of the audience of this book may have existing applications for
which they have already implemented the business logic. For that reason, and since the primary interest of most readers is on Web services, this chapter keeps its focus on Web service development and does not delve into the details of designing and implementing business logic.

3.1 Example Scenarios

Let’s revisit the scenarios introduced in “Typical Web Service Scenarios” on page 11—the adventure builder enterprise scenario and the examples illustrating when Web services work well for an enterprise—from the point of view of designing a Web service. This chapter, rather than discussing design issues abstractly, expands these typical scenarios to illustrate important design issues and to keep the discussion in proper perspective.

In this chapter, we focus on three types of Web services:

1. An informational Web service serving data that is more often read than updated—clients read the information much more than they might update it. In our adventure builder example, a good scenario is a Web service that provides interested clients with travel-related information, such as weather forecasts, for a given city.

2. A Web service that concurrently completes client requests while dealing with a high proportion of shared data that is updated frequently and hence requires heavy use of EIS or database transactions. The airline reservation system partner to adventure builder is a good example of this type of Web service. Many clients can simultaneously send details of desired airline reservations, and the Web service concurrently handles and conducts these reservations.

3. A business process Web service whose processing of a client request includes starting a series of long-running business and workflow processes. Adventure builder enterprise’s decision to build a service interface to partner travel agencies is a good example of this type of Web service. Through this service interface, partner agencies can offer their customers the same services offered in adventure builder’s Web site. The partner agencies use adventure builder’s business logic to fulfill their customer orders. A service such as this receives the details of a travel plan request from a partner agency, and then the service initiates a series of processes to reserve airlines, hotels, rental cars, and so forth for the specified dates.
Discussions of Web service design issues in this chapter include references to these examples and scenarios. However, the discussions use only appropriate characteristics of these scenarios as they pertain to a particular design issue, and they are not meant to represent a complete design of a scenario.

### 3.2 Flow of a Web Service Call

In a Web service scenario, a client makes a request to a particular Web service, such as asking for the weather at a certain location, and the service, after processing the request, sends a response to the client to fulfill the request. When both the client and the Web service are implemented in a Java environment, the client makes the call to the service by invoking a Java method, along with setting up and passing the required parameters, and receives as the response the result of the method invocation.

To help you understand the context within which you design Web services, let’s first take a high-level view at what happens beneath the hood in a typical Web services implementation in a Java environment. Figure 3.1 shows how a Java client communicates with a Java Web service on the J2EE 1.4 platform.

**Note:** Figure 3.1 changes when a non-Java client interacts with a Java Web service. In such a case, the right side of the figure, which reflects the actions of the Web service, stays the same as depicted here, but the left side of the figure would reflect the actions of the client platform. When a Java client invokes a Web service that is on a non-Java platform, the right side of the figure changes to reflect the Web service platform and the left side, which reflects the actions of the client, remains as shown in the figure.

Once the client knows how to access the service, the client makes a request to the service by invoking a Java method, which is passed with its parameters to the client-side JAX-RPC runtime. With the method call, the client is actually invoking an operation on the service. These operations represent the different services of interest to clients. The JAX-RPC runtime maps the Java types to standard XML types and forms a SOAP message that encapsulates the method call and parameters. The runtime then passes the SOAP message through the SOAP handlers, if there are any, and then to the server-side service port.

The client’s request reaches the service through a port, since a port provides access over a specific protocol and data format at a network endpoint consisting of a host name and port number.
Before the port passes the request to the endpoint, it ensures that the J2EE container applies its declarative services (such as security checks) to the SOAP request. After that, any developer-written SOAP handlers in place are applied to the request. Note that SOAP handlers, which are optional, let developers apply application-specific processing logic common to all requests and responses that flow through this endpoint. After the handlers operate on the SOAP message, the message is passed to the service endpoint.
The J2EE container extracts the method call invoked by the client along with the parameters for the call, performs any XML-to-Java object mapping necessary, and hands the method to the Web service interface implementation for further processing. A similar set of steps happens when the service sends back its response.

Note: All the details between the method invocation and response just described happen under the hood. The platform shields the developer from these details. Instead, the developer deals only with typical Java programming language semantics, such as Java method calls, Java types, and so forth.

3.3 Key Web Services Design Decisions

Now that you understand what happens in a Web service interaction, let us look further at the issues involved in the design and implementation of a Web service. We first look at what goes into designing a Web service, examining the issues for which decisions are required and, when possible, making recommendations. (Similarly, Chapter 5 examines the issues to consider when designing a Web service client.)

Before doing so, it is worthwhile to repeat this point:

- Web service technologies basically help you expose an interoperable interface for a new or an existing application. That is, you can add a Web service interface to an existing application to make it interoperable with other applications, or you can develop a completely new application that is interoperable from its inception.

It is important to keep in mind that designing Web service capabilities for an application is separate from designing the business logic of the application. In fact, you design the business logic of an application without regard to whether the application has a Web service interface. To put it another way, the application’s business logic design is the same regardless of whether or not the application has a Web service interface. When you design a Web service interface for an application, you must consider those issues that pertain specifically to interoperability and Web services—and not to the business logic—and you make your design decisions based on these issues.

When designing a Web service, consider the logic flow for typical Web services and the issues they address. In general, a Web service:
• Exposes an interface that clients use to make requests to the service
• Makes a service available to partners and interested clients by publishing the service details
• Receives requests from clients
• Delegates received requests to appropriate business logic and processes the requests
• Formulates and sends a response for the request

Given this flow of logic, the following are the typical steps for designing a Web service.

1. Decide on the interface for clients. Decide whether and how to publish this interface.

You as the Web service developer start the design process by deciding on the interface your service makes public to clients. The interface should reflect the type and nature of the calls that clients will make to use the service. You should consider the type of endpoints you want to use—EJB service endpoints or JAX-RPC service endpoints—and when to use them. You must also decide whether you are going to use SOAP handlers. Last, but not least, since one reason for adding a Web service interface is to achieve interoperability, you must ensure that your design decisions do not affect the interoperability of the service as a whole.

Next, you decide whether you want to publish the service interface, and, if so, how to publish it. Publishing a service makes it available to clients. You can restrict the service’s availability to clients you have personally notified about the service, or you can make your service completely public and register it with a public registry. Note that it is not mandatory for you to publish details of your service, especially when you design your service for trusted partners and do not want to let others know about your service. Keep in mind, too, that restricting service details to trusted partners does not by itself automatically ensure security. Effectively, you are making known the details about your service and its access only to partners rather than the general public.

2. Determine how to receive and preprocess requests.

Once you’ve decided on the interface and, if needed, how to make it available, you are ready to consider how to receive requests from clients. You need to design your service to not only receive a call that a client has made, but also to
do any necessary preprocessing to the request—such as translating the request content to an internal format—before applying the service’s business logic.

3. **Determine how to delegate the request to business logic.**

   Once a request has been received and preprocessed, then you are ready to delegate it to the service’s business logic.

4. **Decide how to process the request.**

   Next, the service processes a request. If the service offers a Web service interface to existing business logic, then the work for this step may simply be to determine how the existing business logic interfaces can be used to handle the Web service’s requests.

5. **Determine how to formulate and send the response.**

   Last, you must design how the service formulates and sends a response back to the client. It’s best to keep these operations logically together. There are other considerations to be taken into account before sending the response to the client.

6. **Determine how to report problems.**

   Since Web services are not immune from errors, you must decide how to throw or otherwise handle exceptions or errors that occur. You need to address such issues as whether to throw service-specific exceptions or whether to let the underlying system throw system-specific exceptions. You must also formulate a plan for recovering from exceptions in those situations that require recovery.

   After considering these steps, start designing your Web service by devising suitable answers to these questions:

   - How will clients make use of your services? Consider what sort of calls clients may make and what might be the parameters of those calls.
   - How will your Web service receive client requests? Consider what kind of endpoints you are going to use for your Web service.
   - What kind of common preprocessing, such as transformations, translations, and logging, needs to be done?
   - How will the request be delegated to business logic?
   - How will the response be formed and sent back?
• What kinds of exceptions will the service throw back to the clients, and when will this happen?

• How are you going to let clients know about your Web service? Are you going to publish your service in public registries, in private registries, or some way other than registries?

Before exploring the details of these design issues, let’s look at a service from a high level. Essentially, a service implementation can be seen as having two layers: an interaction and a processing layer. (See Figure 3.2.)

It is helpful to view a service in terms of layers: an interaction layer and a processing layer.

The service interaction layer consists of the endpoint interface that the service exposes to clients and through which it receives client requests. The interaction layer also includes the logic for how the service delegates the requests to business logic and formulates responses. When it receives requests from clients, the interaction layer performs any required preprocessing before delegating requests to the business logic. When the business logic processing completes, the interaction layer sends back the response to the client. The interaction layer may have additional responsibilities for those scenarios where the service expects to receive XML documents from clients but the business logic deals with objects. In these cases, you map the XML documents to equivalent object representations in the interaction layer before delegating the request to the business logic.

The service processing layer holds all business logic used to process client requests. It is also responsible for integrating with EISs and other Web services. In the case of existing applications adding a Web service interface, the existing application itself typically forms the service processing layer.

Figure 3.2  Layered View of a Web Service
Viewing your service implementation in terms of layers helps to:

- Clearly divide responsibilities
- Provide a common or single location for request processing (both pre- and post-processing) logic in the interaction layer
- Expose existing business logic as a Web service

To put this notion of a layered view in the proper context, let’s look at an example such as adventure builder’s business process Web service scenario. In this scenario, a partner travel agency uses adventure builder enterprise’s Web service to build a travel itinerary for its clients. Through the service interface it exposes to these travel agencies, adventure builder enterprise receives business documents (in XML format) containing all required details for travel itinerary requests. Adventure builder uses its existing workflow systems to process and satisfy these partner requests. The interaction layer of adventure builder’s exposed Web service interface validates these incoming business documents, then converts the incoming XML documents to its internal format or maps document content to Java objects. Once the conversion is finished, control passes to the workflow mechanisms in the processing layer where travel requests are completed. The interaction layer generates responses for completed travel requests, converts responses to XML documents or other appropriate formats, and ensures that responses are relayed to the partner agencies.

It is important to clarify the extent of the preprocessing performed at the interaction layer, since it differs from the JAX-RPC runtime processing. Adventure builder’s interaction layer—its exposed Web service interface—applies service-specific preprocessing to requests coming in to the service. This service-specific preprocessing is performed only if required by the service logic, and it includes converting incoming XML documents to a suitable form or mapping the document contents to Java objects. This mapping of incoming XML documents to business objects is not the same as the JAX-RPC runtime mapping between XML documents and Java objects. Although the container performs the JAX-RPC runtime mapping for all requests and responses, the developer chooses the mapping of incoming XML documents to business objects.

Although there are advantages, as noted previously, to viewing a service in terms of interaction and processing layers, a Web service may opt to merge these two layers into a single layer. There are times when multiple layers make
a service unnecessarily complicated and, in these cases, it may be simpler to design the service as one layer. Typically, this happens in scenarios where the logic in either layer is too small to merit a separate layer.

The weather service scenario is one such service that might benefit from merging the interaction and processing layers into a single layer. This type of service does not need to preprocess incoming requests. A client request to the service for weather information simply includes a name or zip code to identify the location. The service looks up the location’s weather information, forms a response containing the information, and returns it to the client. Since incoming requests require no preprocessing, a layered view of the weather service only complicates what otherwise should be a simple service.

3.4 Designing a Service’s Interaction Layer

A service’s interaction layer has several major responsibilities, and chief among them is the design of the interface the service presents to the client. Since clients access the service through it, the interface is the starting point of a client’s interaction with the service. The interaction layer also handles other responsibilities, such as receiving client requests, delegating requests to appropriate business logic, and creating and sending responses. This section examines the responsibilities of the interaction layer and highlights some guidelines for its design.

3.4.1 Designing the Interface

There are some considerations to keep in mind as you design the interface of your Web service, such as issues regarding overloading methods, choosing the endpoint type, and so forth. Before examining these issues, decide on the approach you want to take for developing the service’s interface definition.

Two approaches to developing the interface definition for a Web service are:

1. Java-to-WSDL—Start with a set of Java interfaces for the Web service and from these create the Web Services Description Language (WSDL) description of the service for others to use.

2. WSDL-to-Java—Start with a WSDL document describing the details of the Web service interface and use this information to build the corresponding Java interfaces.
How do these two approaches compare? Starting with Java interfaces and creating a WSDL document is probably the easier of the two approaches. With this approach, you need not know any WSDL details because you use vendor-provided tools to create the WSDL description. While these tools make it easy for you to generate WSDL files from Java interfaces, you do lose some control over the WSDL file creation.

With the Java-to-WSDL approach, keep in mind that the exposed service interface may be too unstable from a service evolution point of view.

With the Java-to-WSDL approach, it may be hard to evolve the service interface without forcing a change in the corresponding WSDL document, and changing the WSDL might require rewriting the service’s clients. These changes, and the accompanying instability, can affect the interoperability of the service itself. Since achieving interoperability is a prime reason to use Web services, the instability of the Java-to-WSDL approach is a major drawback. Also, keep in mind that different tools may use different interpretations for certain Java types (for example, java.util.Date might be interpreted as java.util.Calendar), resulting in different representations in the WSDL file. While not common, these representation variations may result in some semantic surprises.

On the other hand, the WSDL-to-Java approach gives you a powerful way to expose a stable service interface that you can evolve with relative ease. Not only does it give you greater design flexibility, the WSDL-to-Java approach also provides an ideal way for you to finalize all service details—from method call types and fault types to the schemas representing exchanged business documents—before you even start a service or client implementation. Although a good knowledge of WSDL and the WS-I Basic Profile is required to properly describe these Web services details, using available tools helps address these issues.

After you decide on the approach to take, you must still resolve other interface design details, which are described in the next sections.

3.4.1.1 Choice of the Interface Endpoint Type

In the J2EE platform, you have two choices for implementing the Web service interface—you can use a JAX-RPC service endpoint (also referred to as a Web tier endpoint) or an EJB service endpoint (also referred to as an EJB tier endpoint). Using one of these endpoint types makes it possible to embed the endpoint in the same tier as the service implementation. This simplifies the service implementation, because
it obviates the need to place the endpoint in its own tier where the presence of the endpoint is solely to act as a proxy directing requests to other tiers that contain the service’s business logic.

When you develop a new Web service that does not use existing business logic, choosing the endpoint type to use for the Web service interface is straightforward. The endpoint type choice depends on the nature of your business logic—whether the business logic of the service is completely contained within either the Web tier or the EJB tier:

- Use a JAX-RPC service endpoint when the processing layer is within the Web tier.
- Use an EJB service endpoint when the processing layer is only on the EJB tier.

When you add a Web service interface to an existing application or service, you must consider whether the existing application or service preprocesses requests before delegating them to the business logic. If so, then keep the following guideline in mind:

- When you add a Web service interface for an existing application, choose an endpoint type suited for the tier on which the preprocessing logic occurs in the existing application. Use a JAX-RPC service endpoint when the preprocessing occurs on the Web tier of the existing application and an EJB service endpoint when preprocessing occurs on the EJB tier.

If the existing application or service does not require preprocessing of the incoming request, choose the appropriate endpoint that is present in the same tier as the existing business logic. Besides these major considerations for choosing an endpoint type, there are other, more subtle differences between an EJB service endpoint and a JAX-RPC service endpoint. You may find it helpful to keep in mind these additional points when choosing a Web service endpoint type:

- **Multi-threaded access considerations**—An EJB service endpoint, because it is implemented as a stateless session bean, need not worry about multi-threaded access since the EJB container is required to serialize requests to any particular instance of a stateless session bean. For a JAX-RPC service
A JAX-RPC service endpoint has to handle concurrent client access on its own, whereas the EJB container takes care of concurrent client access for an EJB service endpoint.

**Transaction considerations**—The transactional context of the service implementation’s container determines the transactional context in which a service implementation runs. Since a JAX-RPC service endpoint runs in a Web container, its transactional context is unspecified. There is also no declarative means to automatically start the transaction. Thus, you need to use JTA to explicitly demarcate the transaction.

On the other hand, an EJB service endpoint runs in the transaction context of an EJB container. You as the developer need to declaratively demarcate transactions. The service’s business logic thus runs under the transactional context as defined by the EJB’s container-transaction element in the deployment descriptor.

If the Web service’s business logic requires using transactions (and the service has a JAX-RPC service endpoint), you must implement the transaction-handling logic using JTA or some other similar facility. If your service uses an EJB service endpoint, you can use the container’s declarative transaction services. By doing so, the container is responsible for handling transactions according to the setting of the deployment descriptor element container-transaction.

**Considerations for method-level access permissions**—A Web service’s methods can be accessed by an assortment of different clients, and you may want to enforce different access constraints for each method.

When you want to control service access at the individual method level, consider using an EJB service endpoint rather than a JAX-RPC service endpoint.

Enterprise beans permit method-level access permission declaration in the deployment descriptor—you can declare various access permissions for different enterprise bean methods and the container correctly handles access to these methods. This holds true for an EJB service endpoint, since it is a stateless ses-
sion bean. A JAX-RPC service endpoint, on the other hand, does not have a facility for declaring method-level access constraints, requiring you to do this programmatically. See Chapter 7 for more information.

- **HTTP session access considerations**—A JAX-RPC service endpoint, because it runs in the Web container, has complete access to an HttpSession object. Access to an HttpSession object, which can be used to embed cookies and store client state, may help you build session-aware clients. An EJB service endpoint, which runs in the EJB container, has no such access to Web container state. However, generally HTTP session support is appropriate for short duration conversational interactions, whereas Web services often represent business processes with longer durations and hence need additional mechanisms. See “Correlating Messages” on page 359 for one such strategy.

### 3.4.1.2 Granularity of Service

Much of the design of a Web service interface involves designing the service’s operations, or its methods. You first determine the service’s operations, then define the method signature for each operation. That is, you define each operation’s parameters, its return values, and any errors or exceptions it may generate.

- It is important to consider the granularity of the service’s operations when designing your Web service interface.

For those Web services that implement a business process, the nature of the business process itself often dictates the service’s granularity. Business processes that exchange documents, such as purchase orders and invoices, by their nature result in a Web service interface that is coarse grained. With more interactive Web services, you need to carefully choose the granularity of these operations.

You should keep the same considerations in mind when designing the methods for a Web service as when designing the methods of a remote enterprise bean. This is particularly true not only regarding the impact of remote access on performance but also with Web services; it is important with Web services because there is an underlying XML representation requiring parsing and taking bandwidth. Thus, a good rule is to define the Web service’s interface for optimal granularity of its operations; that is, find the right balance between coarse-grained and fine-grained granularity.
Generally, you should consolidate related fine-grained operations into more coarse-grained ones to minimize expensive remote method calls.

More coarse-grained service operations, such as returning catalog entries in sets of categories, keep network overhead lower and improve performance. However, they are sometimes less flexible from a client’s point of view. While finer-grained service operations, such as browsing a catalog by products or items, offer a client greater flexibility, these operations result in greater network overhead and reduced performance.

Keep in mind that too much consolidation leads to inefficiencies.

For example, consolidating logically different operations is inefficient and should be avoided. It is much better to consolidate similar operations or operations that a client is likely to use together, such as querying operations.

When exposing existing stateless session beans as Web service endpoints, ensure that the Web service operations are sufficiently coarse grained.

If you are planning to expose existing stateless session beans as Web service endpoints, remember that such beans may not have been designed with Web services in mind. Hence, they may be too fine grained to be good Web service endpoints. You should consider consolidating related operations into a single Web service operation.

Good design for our airline reservation Web service, for example, is to expect the service’s clients to send all information required for a reservation—destination, preferred departure and arrival times, preferred airline, and so forth—in one invocation to the service, that is, as one large message. This is far more preferable than to have a client invoke a separate method for each piece of information comprising the reservation. To illustrate, it is preferable to have clients use the interface shown in Code Example 3.1.

```java
public interface AirlineTicketsIntf extends Remote {
  public String submitReservationRequest(
      AirReservationDetails details) throws RemoteException;
}
```

**Code Example 3.1** Using Consolidation for Greater Efficiency (Recommended)
Chapter 3  Service Endpoint Design

Code Example 3.1 combines logically-related data into one large message for a more efficient client interaction with the service. This is preferable to receiving the data with individual method calls, as shown in Code Example 3.2.

```java
public interface AirlineTicketsIntf extends Remote {
    public String submitFlightInformation(FlightDetails fltInfo)
        throws RemoteException;
    public String submitPreferredDates(Date depart, Date arrive)
        throws RemoteException;
    // other similar methods
}
```

**Code Example 3.2  Retrieving Data with Separate Method Calls (Not Recommended)**

However, it might not be a good idea to combine in a single service invocation the same reservation with an inquiry method call.

Along with optimal granularity, you should consider data caching issues. Coarse-grained services involve transferring large amounts of data. If you opt for more coarse-grained service operations, it is more efficient to cache data on the client side to reduce the number of round trips between the client and the server.

### 3.4.1.3 Parameter Types for Web Service Operations

A Web service interface exposes a set of method calls to clients. When invoking a service interface method, a client may have to set values for the parameters associated with the call. When you design an interface’s methods, choose carefully the types of these parameters. Keep in mind that a method call and its parameters are sent as a SOAP message between the client and the service. To be part of a SOAP message, parameters must be mapped to XML. When received at the client or service end, the same parameters must be mapped from XML to their proper types or objects. This section describes some guidelines to keep in mind when defining method call parameters and return values.

**Note:** Since each call potentially may return a value, the discussion in this section about parameter values applies equally to return values.
Parameters for Web service method calls may be standard Java objects and types, XML documents, or even nonstandard types. Whether you use the Java-to-WSDL approach or the WSDL-to-Java approach, each type of parameter must be mapped to its XML equivalent in the SOAP message. Figure 3.3 shows how the binding happens for various types of parameters.

### 3.4.1.3.1 Java Objects as Parameters

Parameters for Web service calls can be standard Java types and objects. If you use the Java-to-WSDL approach, you specify the parameter types as part of the arguments of the method calls of your Java interface. If you use the WSDL-to-Java approach, you specify the parameter types as the type or element attributes of the part element of each message in your WSDL. The type of a parameter that you use has a significant effect on the portability and interoperability of your service.

The platform supports the following Java data types. (Refer to the JAX-RPC specification at [http://java.sun.com/xml/jaxrpc/](http://java.sun.com/xml/jaxrpc/) for the equivalent WSDL mappings for these Java data types.)
Chapter 3  Service Endpoint Design

- Java primitive types boolean, byte, short, int, long, float, and double, along with their corresponding wrapper Java classes
- Standard Java classes: String, Date, Calendar, BigInteger, BigDecimal, QName, and URI
- Java arrays with JAX-RPC-supported Java types as members
- JAX-RPC value types—user-defined Java classes, including classes with JavaBeans™ component-like properties

When designing parameters for method calls in a Web service interface, choose parameters that have standard type mappings. (See Figure 3.3.) Always keep in mind that the portability and interoperability of your service is reduced when you use parameter types that by default are not supported.

As Figure 3.3 shows, parameters that have standard type mappings are bound implicitly. However, the developer must do more work when using parameters that do not have standard type mappings. See “Handling Nonstandard Type Parameters” on page 76 for more details on using nonstandard Java types and possible side effects of such use.

Here are some additional helpful points to consider when you use Java objects with standard type mappings as parameters.

1. Many applications and services need to pass lists of objects. However, utilities for handling lists, such as ArrayList and Collection, to name a few, are not supported standard types. Instead, Java arrays provide equivalent functionality, and have a standard mapping provided by the platform.

2. JAX-RPC value types are user-defined Java classes with some restrictions. They have constructors and may have fields that are public, private, protected, static, or transient. JAX-RPC value types may also have methods, including set and get methods for setting and getting Java class fields.

   However, when mapping JAX-RPC value types to and from XML, there is no standard way to retain the order of the parameters to the constructors and other methods. Hence, avoid setting the JAX-RPC value type fields through the constructor. Using the get and set methods to retrieve or set value type fields avoids this mapping problem and ensures portability and interoperability.

3. The J2EE platform supports nested JAX-RPC value types; that is, JAX-RPC value types that reference other JAX-RPC value types within themselves. For
clarity, it is preferable to use this feature and embed value type references rather than to use a single flat, large JAX-RPC value type class.

4. The J2EE platform, because of its support for the SOAP message with attachment protocol, also supports the use of MIME-encoded content. It provides Java mappings for a subset of MIME types. (See Table 3.1.)

**Table 3.1** Mapping of MIME Types

<table>
<thead>
<tr>
<th>MIME Type</th>
<th>Java Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>image/gif</td>
<td>java.awt.Image</td>
</tr>
<tr>
<td>image/jpeg</td>
<td>java.awt.Image</td>
</tr>
<tr>
<td>text/plain</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>multipart/*</td>
<td>javax.mail.internet.MimeMultipart</td>
</tr>
<tr>
<td>text/xml or application/xml</td>
<td>javax.xml.transform.Source</td>
</tr>
</tbody>
</table>

Since the J2EE container automatically handles mappings based on the Java types, using these Java-MIME mappings frees you from the intricacies of sending and retrieving documents and images as part of a service's request and response handling. For example, your service, if it expects to receive a GIF image with a MIME type of image/gif, can expect the client to send a java.awt.Image object. A sample Web service interface that receives an image might look like the one shown in Code Example 3.3:

```java
import java.awt.Image;
public interface WeatherMapService extends Remote {
    public void submitWeatherMap(Image weatherMap)
        throws RemoteException, InvalidMapException;
}
```

**Code Example 3.3** Receiving a java.awt.Image Object

In this example, the Image object lets the container implementation handle the image-passing details. The container provides javax.activation.DataHandler classes, which work with the Java Activation Framework to accomplish the Java-MIME and MIME-Java mappings.
Considering this mapping between Java and MIME types, it is best to send images and XML documents that are in a Web service interface using the Java types shown in Table 3.1. However, you should be careful about the effect on the interoperability of your service. See “Interoperability” on page 86 for more details.

3.4.1.3.2 XML Documents as Parameters

There are scenarios when you want to pass XML documents as parameters. Typically, these occur in business-to-business interactions where there is a need to exchange legally binding business documents, track what is exchanged, and so forth. Exchanging XML documents as part of a Web service is addressed in a separate section—see “Handling XML Documents in a Web Service” on page 105 for guidelines to follow when passing XML documents as parameters.

3.4.1.3.3 Handling Nonstandard Type Parameters

JAX-RPC technology, in addition to providing a rich standard mapping set between XML and Java data types, also provides an extensible type mapping framework. Developers can use this framework to specify pluggable, custom serializers and deserializers that support nonstandard type mappings.

Extensible type mapping frameworks, which developers may use to support nonstandard type mappings, are not yet a standard part of the J2EE platform.

Vendors currently can provide their own solutions to this problem. It must be emphasized that if you implement a service using some vendor’s implementation-specific type mapping framework, then your service is not guaranteed to be portable and interoperable.

Because of portability limitations, you should avoid passing parameters that require the use of vendor-specific serializers or deserializers.

Instead, a better way is to pass these parameters as SOAP document fragments represented as a DOM subtree in the service endpoint interface. (See Figure 3.3.) If so, you should consider binding (either manually or using JAXB) the SOAP fragments to Java objects before passing them to the processing layer to avoid tightly coupling the business logic with the document fragment.
3.4.1.4 Interfaces with Overloaded Methods

In your service interface, you may overload methods and expose them to the service’s clients. Overloaded methods share the same method name but have different parameters and return values. If you do choose to use overloaded methods as part of your service interface, keep in mind that there are some limitations, as follows:

- If you choose the WSDL-to-Java approach, there are limitations to representing overloaded methods in a WSDL description. In the WSDL description, each method call and its response are represented as unique SOAP messages. To represent overloaded methods, the WSDL description would have to support multiple SOAP messages with the same name. WSDL version 1.1 does not have this capability to support multiple messages with the same name.

- If you choose the Java-to-WSDL approach and your service exposes overloaded methods, be sure to check how any vendor-specific tools you are using represent these overloaded methods in the WSDL description. You need to ensure that the WSDL representation of overloaded methods works in the context of your application.

Let’s see how this applies in the weather service scenario. As the provider, you might offer the service to clients, letting them look up weather information by city name or zip code. If you use the Java-to-WSDL approach, you might first define the WeatherService interface as shown in Code Example 3.4.

```java
public interface WeatherService extends Remote {
    public String getWeather(String city) throws RemoteException;
    public String getWeather(int zip) throws RemoteException;
}
```

**Code Example 3.4**  WeatherService Interface for Java-to-WSDL Approach

After you define the interface, you run the vendor-provided tool to create the WSDL from the interface. Each tool has its own way of representing the getWeather overloaded methods in the WSDL, and your WSDL reflects the particular tool you use. For example, if you use the J2EE 1.4 SDK from Sun Microsystems, its wscompile tool creates from the WeatherService interface the WSDL shown in Code Example 3.5.
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="WeatherWebService">
  <types/>
  <message name="WeatherService_getWeather">
    <part name="int_1" type="xsd:int"/>
  </message>
  <message name="WeatherService_getWeatherResponse">
    <part name="result" type="xsd:string"/>
  </message>
  <message name="WeatherService_getWeather2">
    <part name="String_1" type="xsd:string"/>
  </message>
  <message name="WeatherService_getWeather2Response">
    <part name="result" type="xsd:string"/>
  </message>
  ...
</definitions>

Code Example 3.5  Generated WSDL for WeatherService Interface

Notice that the WSDL represents the getWeather overloaded methods as two different SOAP messages, naming one getWeather, which takes an integer for the zip code as its parameter, and the other getWeather2, which takes a string parameter for the city. As a result, a client interested in obtaining weather information using a city name invokes the service by calling getWeather2, as shown in Code Example 3.6.

... Context ic = new InitialContext(); WeatherWebService weatherSvc = (WeatherWebService) ic.lookup("java:comp/env/service/WeatherService"); WeatherServiceIntf port = (WeatherServiceIntf) weatherSvc.getPort(WeatherServiceIntf.class); String returnValue = port.getWeather2("San Francisco"); ...

Code Example 3.6  Using Weather Service Interface with Java-to-WSDL Approach
For example, to obtain the weather information for San Francisco, the client called `port.getWeather2("San Francisco")`. Keep in mind that another tool may very likely generate a WSDL whose representation of overloaded methods is different.

You may want to avoid using overloaded methods in your Java interface altogether if you prefer to have only intuitive method names in the WSDL.

If instead you choose to use the WSDL-to-Java approach, your WSDL description might look as follows. (See Code Example 3.7.)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="WeatherWebService" ...>
  <types/>
  <message name="WeatherService_getWeatherByZip">
    <part name="int_1" type="xsd:int"/>
  </message>
  <message name="WeatherService_getWeatherByZipResponse">
    <part name="result" type="xsd:string"/>
  </message>
  <message name="WeatherService_getWeatherByCity">
    <part name="String_1" type="xsd:string"/>
  </message>
  <message name="WeatherService_getWeatherByCityResponse">
    <part name="result" type="xsd:string"/>
  </message>
  ...
</definitions>
```

**Code Example 3.7**  WSDL for Weather Service with Overloaded Methods Avoided

Since the messages in a WSDL file must have unique names, you must use different message names to represent methods that you would otherwise overload. These different message names actually convert to different method calls in your interface. Notice that the WSDL includes a method `getWeatherByZip`, which takes an integer parameter, and a method `getWeatherByCity`, which takes a string parameter. Thus, a client wishing to obtain weather information by city name from
a WeatherService interface associated with the WSDL in Code Example 3.7 might invoke the service as shown in Code Example 3.8.

```java
...  
Context ic = new InitialContext();  
WeatherWebService weatherSvc = (WeatherWebService)
ic.lookup("java:comp/env/service/WeatherService");  
WeatherServiceIntf port = (WeatherServiceIntf)
weatherSvc.getPort(WeatherServiceIntf.class);  
String returnValue = port.getWeatherByCity("San Francisco");  
...  
```

**Code Example 3.8** Using Weather Service with WSDL-to-Java Approach

### 3.4.1.5 Handling Exceptions

Just like any Java or J2EE application, a Web service application may encounter an error condition while processing a client request. A Web service application needs to properly catch any exceptions thrown by an error condition and propagate these exceptions. For a Java application running in a single virtual machine, you can propagate exceptions up the call stack until reaching a method with an exception handler that handles the type of exception thrown. To put it another way, for non-Web service J2EE and Java applications, you may continue to throw exceptions up the call stack, passing along the entire stack trace, until reaching a method with an exception handler that handles the type of exception thrown. You can also write exceptions that extend or inherit other exceptions.

However, throwing exceptions in Web service applications has additional constraints that impact the design of the service endpoint. When considering how the service endpoint handles error conditions and notifies clients of errors, you must keep in mind these points:

- Similar to requests and responses, exceptions are also sent back to the client as part of the SOAP messages.
- Your Web service application should support clients running on non-Java platforms that may not have the same, or even similar, error-handling mechanisms as the Java exception-handling mechanism.
A Web service application may encounter two types of error conditions. One type of error might be an irrecoverable system error, such as an error due to a network connection problem. When an error such as this occurs, the JAX-RPC runtime on the client throws the client platform’s equivalent of an irrecoverable system exception. For Java clients, this translates to a RuntimeException.

A Web service application may also encounter a recoverable application error condition. This type of error is called a service-specific exception. The error is particular to the specific service. For example, a weather Web service might indicate an error if it cannot find weather information for a specified city.

To illustrate the Web service exception-handling mechanism, let’s examine it in the context of the weather Web service example. When designing the weather service, you want the service to be able to handle a scenario in which the client requests weather information for a nonexistent city. You might design the service to throw a service-specific exception, such as CityNotFoundException, to the client that made the request. You might code the service interface so that the getWeather method throws this exception. (See Code Example 3.9.)

```java
public interface WeatherService extends Remote {
    public String getWeather(String city) throws CityNotFoundException, RemoteException;
}
```

**Code Example 3.9** Throwing a Service-Specific Exception

Service-specific exceptions like CityNotFoundException, which are thrown by the Web service to indicate application-specific error conditions, must be checked exceptions that directly or indirectly extend java.lang.Exception. They cannot be unchecked exceptions. Code Example 3.10 shows a typical implementation of a service-specific exception, such as for CityNotFoundException.

```java
public class CityNotFoundException extends Exception {
    private String message;
    public CityNotFoundException(String message) {
        super(message);
        this.message = message;
    }
    public String getMessage() {
        return message;
    }
}
```
Code Example 3.10  Implementation of a Service-Specific Exception

Code Example 3.11 shows the service implementation for the same weather service interface. This example illustrates how the service might throw CityNotFoundException.

```java
public class WeatherServiceImpl implements WeatherService {
    public String getWeather(String city) throws CityNotFoundException {
        if(!validCity(city))
            throw new CityNotFoundException(city + " not found");
        // Get weather info and return it back
    }
}
```

Code Example 3.11  Example of a Service Throwing a Service-Specific Exception

Chapter 5 describes the details of handling exceptions on the client side. (In particular, refer to “Handling Exceptions” on page 230.) On the service side, keep in mind how to include exceptions in the service interface and how to throw them. Generally, you want to do the following:

- Convert application-specific errors and other Java exceptions into meaningful service-specific exceptions and throw these service-specific exceptions to the clients.

Although they promote interoperability among heterogeneous platforms, Web service standards cannot address every type of exception thrown by different platforms. For example, the standards do not specify how Java exceptions such as `java.io.IOException` and `javax.ejb.EJBException` should be returned to the client. As a consequence, it is important for a Web service—from the service’s interoperability point of view—to not expose Java-specific exceptions (such as those just mentioned) in the Web service interface. Instead, throw a service-specific exception. In addition, keep the following points in mind:
You cannot throw nonserializable exceptions to a client through the Web service endpoint.

When a service throws java or javax exceptions, the exception type and its context information are lost to the client that receives the thrown exception. For example, if your service throws a javax.ejb.FinderException exception to the client, the client may receive an exception named FinderException, but its type information may not be available to the client. Furthermore, the type of the exception to the client may not be the same as the type of the thrown exception. (Depending on the tool used to generate the client-side interfaces, the exception may even belong to some package other than javax.ejb.)

As a result, you should avoid directly throwing java and javax exceptions to clients. Instead, when your service encounters one of these types of exceptions, wrap it within a meaningful service-specific exception and throw this service-specific exception back to the client. For example, suppose your service encounters a javax.ejb.FinderException exception while processing a client request. The service should catch the FinderException exception, and then, rather than throwing this exception as is back to the client, the service should instead throw a service-specific exception that has more meaning for the client. See Code Example 3.12.

```java
... 
try {
    // findByPrimaryKey
    // Do processing
    // return results
} catch (javax.ejb.FinderException fe) {
    throw new InvalidKeyException(
        "Unable to find row with given primary key");
}
```

**Code Example 3.12** Converting an Exception into a Service-Specific Exception

Exception inheritances are lost when you throw a service-specific exception.

You should avoid defining service-specific exceptions that inherit or extend other exceptions. For example, if CityNotFoundException in Code Example 3.10
extends another exception, such as RuntimeException, then when the service throws CityNotFoundException, methods and properties inherited from RuntimeException are not passed to the client.

☐ The exception stack trace is not passed to the client.

The stack trace for an exception is relevant only to the current execution environment and is meaningless on a different system. Hence, when a service throws an exception to the client, the client does not have the stack trace explaining the conditions under which the exception occurred. Thus, you should consider passing additional information in the message for the exception.

Web service standards make it easier for a service to pass error conditions to a client in a platform-independent way. While the following discussion may be of interest, it is not essential that developers know these details about the J2EE platform’s error-handling mechanisms for Web services.

As noted previously, error conditions are included within the SOAP messages that a service returns to clients. The SOAP specification defines a message type, called fault, that enables error conditions to be passed as part of the SOAP message yet still be differentiated from the request or response portion. Similarly, the WSDL specification defines a set of operations that are possible on an endpoint. These operations include input and output operations, which represent the request and response respectively, and an operation called fault.

A SOAP fault defines system-level exceptions, such as RemoteException, which are irrecoverable errors. The WSDL fault denotes service-specific exceptions, such as CityNotFoundException, and these are recoverable application error conditions. Since the WSDL fault denotes a recoverable error condition, the platform can pass it as part of the SOAP response message. Thus, the standards provide a way to exchange fault messages and map these messages to operations on the endpoint.

Code Example 3.13 shows the WSDL code for the same weather Web service example. This example illustrates how service-specific exceptions are mapped just like input and output messages are mapped.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions ...>
...
<message name="WeatherService_getWeather">
  <part name="String_1" type="xsd:string"/>
```
Designing a Service’s Interaction Layer

Code Example 3.13  Mapping a Service-Specific Exception in WSDL

3.4.1.6 Use of Handlers

As discussed in Chapter 2, and as shown in Figure 3.1, JAX-RPC technology enables you to plug in SOAP message handlers, thus allowing processing of SOAP messages that represent requests and responses. Plugging in SOAP message handlers gives you the capability to examine and modify the SOAP requests before they are processed by the Web service and to examine and modify the SOAP responses before they are delivered to the client.

Handlers are particular to a Web service and are associated with the specific port of the service. As a result of this association, the handler’s logic applies to all SOAP requests and responses that pass through a service’s port. Thus, you use these message handlers when your Web service must perform some SOAP message-specific processing common to all its requests and responses. Because handlers are common to all requests and responses that pass through a Web service endpoint, keep the following guideline in mind:
It is not advisable to put in a handler business logic or processing particular to specific requests and responses.

You cannot store client-specific state in a handler: A handler’s logic acts on all requests and responses that pass through an endpoint. However, you may use the handler to store port-specific state, which is state common to all method calls on that service interface. Note also that handlers execute in the context of the component in which they are present.

Do not store client-specific state in a handler.

Also note that handlers work directly on the SOAP message, and this involves XML processing. You can use handlers to pass client-specific state through the message context. (See “Passing Context Information on Web Service Calls” on page 366.)

Use of handlers can result in a significant performance impact for the service as a whole.

Use of handlers could potentially affect the interoperability of your service. See the next section on interoperability. Keep in mind that it takes advanced knowledge of SOAP message manipulation APIs (such as SAAJ) to correctly use handlers. To avoid errors, Web service developers should try to use existing or vendor-supplied handlers. Using handlers makes sense primarily for writing system services such as auditing, logging, and so forth.

3.4.1.7 Interoperability

A major benefit of Web services is interoperability between heterogeneous platforms. To get the maximum benefit, you want to design your Web service to be interoperable with clients on any platform, and, as discussed in Chapter 2, the Web Services Interoperability (WS-I) organization helps in this regard. WS-I promotes a set of generic protocols for the interoperable exchange of messages between Web services. The WS-I Basic Profile promotes interoperability by defining and recommending how a set of core Web services specifications and standards (including SOAP, WSDL, UDDI, and XML) can be used for developing interoperable Web services.
In addition to the WS-I protocols, other groups, such as SOAPBuilders Interoperability group (see http://java.sun.com/wsit/interop/sb/index.html), provide common testing grounds that make it easier to test the interoperability of various SOAP implementations. This has made it possible for various Web services technology vendors to test the interoperability of implementations of their standards. When you implement your service using technologies that adhere to the WS-I Basic Profile specifications, you are assured that such services are interoperable.

Apart from these standards and testing environments, you as the service developer must design and implement your Web service so that maximum interoperability is possible. For maximum interoperability, you should keep these three points in mind:

1. The two messaging styles and bindings supported by WSDL
2. The WS-I support for attachments
3. The most effective way to use handlers

WSDL supports two types of messaging styles: rpc and document. The WSDL style attribute indicates the messaging style. (See Code Example 3.14.) A style attribute set to rpc indicates a RPC-oriented operation, where messages contain parameters and return values, or function signatures. When the style attribute is set to document, it indicates a document-oriented operation, one in which messages contain documents. Each operation style has a different effect on the format of the body of a SOAP message.

Along with operation styles, WSDL supports two types of serialization and deserialization mechanisms: a literal and an encoded mechanism. The WSDL use attribute indicates which mechanism is supported. (See Code Example 3.14.) A literal value for the use attribute indicates that the data is formatted according to the abstract definitions within the WSDL document. The encoded value means data is formatted according to the encodings defined in the URI specified by the encodingStyle attribute. Thus, you can choose between an rpc or document style of message passing and each message can use either a literal or encoded data formatting.

Because the WS-I Basic Profile 1.0, to which J2EE1.4 platform conforms, supports only literal bindings, you should avoid encoded bindings.
Literal bindings cannot represent complex types, such as objects with circular references, in a standard way.

Code Example 3.14 shows a snippet from the WSDL document illustrating how the sample weather service specifies these bindings.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions .......>
  <binding name="WeatherServiceBinding" type="tns:WeatherService">
    <operation name="getWeather">
      <input>
        <soap:body use="literal"
          namespace="urn:WeatherWebService"/>
      </input>
      <output>
        <soap:body use="literal"
          namespace="urn:WeatherWebService"/>
      </output>
      <soap:operation soapAction=""/></operation>
    <soap:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http" />
  </binding>
</service .....>
</definitions>
```

**Code Example 3.14** Specifying WSDL Bindings

It is important to keep in mind these message styles and bindings, particularly when you design the interface using the WSDL-to-Java approach and when you design the WSDL for your service. When you use the Java-to-WSDL approach, you rely on the vendor-provided tools to generate the WSDL for your Java interfaces, and they can be counted on to create WS-I-compliant WSDL for your service. However, note that some vendors may expect you to specify certain options to ensure the creation of a WS-I-compliant WSDL. For example, the J2EE 1.4 SDK from Sun Microsystems provides a \texttt{wscompile} tool, which expects the developer to use the \texttt{-f:wsi} flag to create the WS-I-compliant WSDL for the service. It is also a good idea to check the WSDL document itself to ensure that whatever tool you use created the document correctly.
Regarding the second issue, you should note that the WS-I Basic Profile 1.0 (which is the profile supported by the J2EE 1.4 platform) does not address attachments. The section, “Parameter Types for Web Service Operations” on page 72, which discussed Java-MIME type mappings provided by the J2EE platform, advised that an efficient design is to use these mappings to send images and XML documents within a completely Java environment. Because the WS-I Basic Profile, version 1.0 does not address attachments, a Web service that uses these mappings may not be interoperable with clients on a non-Java platform.

Since the WS-I Basic Profile 1.0 specification does not address attachments, a Web service using the Java-MIME mappings provided by the J2EE platform is not guaranteed to be interoperable.

Since most Web services rely on an exchange of business documents, and interoperability is not always guaranteed, it is important that you properly understand the options for handling XML documents. The section, “Exchanging XML Documents” on page 107, explains the various options available to Web services for exchanging XML documents in an interoperable manner. It should also be noted that the next version of the WS-I Basic Profile specification addresses a standard way to send attachments, and later versions of the J2EE platforms will incorporate this.

Last is the issue of handlers. Handlers, which give you access to SOAP messages, at the same time impose major responsibilities on you.

When using handlers, you must be careful not to change a SOAP message to the degree that the message no longer complies with WS-I specifications, thereby endangering the interoperability of your service.

This ends the discussion of considerations for designing a Web service interface. The next sections examine other responsibilities of the interaction layer, such as receiving and delegating requests and formulating responses.

### 3.4.2 Receiving Requests

The interaction layer, through the endpoint, receives client requests. The platform maps the incoming client requests, which are in the form of SOAP messages, to method calls present in the Web service interface.
Before delegating these incoming client requests to the Web service business logic, you should perform any required security validation, transformation of parameters, and other required preprocessing of parameters.

As noted in “Parameter Types for Web Service Operations” on page 72 and elsewhere, Web service calls are basically method calls whose parameters are passed as either Java objects, XML documents (javax.xml.transform.Source objects), or even SOAP document fragments (javax.xml.soap.SOAPElement objects).

For parameters that are passed as Java objects (such as String, int, JAX-RPC value types, and so forth), do the application-specific parameter validation and map the incoming objects to domain-specific objects in the interaction layer before delegating the request to the processing layer.

You may have to undertake additional steps to handle XML documents that are passed as parameters. These steps, which are best performed in the interaction layer of your service, are as follows:

1. The service endpoint should validate the incoming XML document against its schema. For details and guidelines on how and when to validate incoming XML documents, along with recommended validation techniques, refer to “Validating XML Documents” on page 139.

2. When the service’s processing layer and business logic are designed to deal with XML documents, you should transform the XML document to an internally supported schema, if the schema for the XML document differs from the internal schema, before passing the document to the processing layer.

3. When the processing layer deals with objects but the service interface receives XML documents, then, as part of the interaction layer, map the incoming XML documents to domain objects before delegating the request to the processing layer. For details and guidelines on mapping techniques for incoming XML documents, refer to “Mapping Schemas to the Application Data Model” on page 143.

It is important that these three steps—validation of incoming parameters or XML documents, translation of XML documents to internal supported sche-
mas, and mapping documents to domain objects—be performed as close to the service endpoint as possible, and certainly in the service interaction layer.

A design such as this helps to catch errors early, and thus avoids unnecessary calls and round-trips to the processing layer. Figure 3.4 shows the recommended way to handle requests and responses in the Web service’s interaction layer.

The Web service’s interaction layer handles all incoming requests and delegates them to the business logic exposed in the processing layer. When implemented in this manner, the Web service interaction layer has several advantages, since it gives you a common location for the following tasks:

- Managing the handling of requests so that the service endpoint serves as the initial point of contact
- Invoking security services, including authentication and authorization
- Validating and transforming incoming XML documents and mapping XML documents to domain objects

![Figure 3.4 Web Service Request Processing](image-url)
• Delegating to existing business logic
• Handling errors

It is generally advisable to do all common processing—such as security checks, logging, auditing, input validation, and so forth—for requests at the interaction layer as soon as a request is received and before passing it to the processing layer.

### 3.4.3 Delegating Web Service Requests to Processing Layer

After designing the request preprocessing tasks, the next step is to design how to delegate the request to the processing layer. At this point, consider the kind of processing the request requires, since this helps you decide how to delegate the request to the processing layer. All requests can be categorized into two large categories based on the time it takes to process the request, namely:

• A request that is processed in a short enough time so that a client can afford to block and wait to receive the response before proceeding further. In other words, the client and the service interact in a synchronous manner such that the invoking client blocks until the request is processed completely and the response is received.

• A request that takes a long time to be processed, so much so that it is not a good idea to make the client wait until the processing is completed. In other words, the client and the service interact in an asynchronous manner such that the invoking client need not block and wait until the request is processed completely.

**Note:** When referring to request processing, we use the terms synchronous and asynchronous from the point of view of when the client’s request processing completes fully. Keep in mind that, under the hood, an asynchronous interaction between a client and a service might result in a synchronous invocation over the network, since HTTP is by its nature synchronous. Similarly, SOAP messages sent over HTTP are also synchronous.
The weather information service is a good example of a synchronous interaction between a client and a service. When it receives a client’s request, the weather service must look up the required information and send back a response to the client. This look-up and return of the information can be achieved in a relatively short time, during which the client can be expected to block and wait. The client continues its processing only after it obtains a response from the service. (See Figure 3.5.)
A Web service such as this can be designed using a service endpoint that receives the client’s request and then delegates the request directly to the service’s appropriate logic in the processing layer. The service’s processing layer processes the request and, when the processing completes, the service endpoint returns the response to the client. (See Figure 3.6.)

Code Example 3.15 shows the weather service interface performing some basic parameter validation checks in the interaction layer. The interface also gets required information and passes that information to the client in a synchronous manner:

```java
public class WeatherServiceImpl implements WeatherService, ServiceLifecycle {
    public void init(Object context) throws JAXRPCException {....}

    public String getWeather(String city)
        throws CityNotFoundException {
        /** Validate parameters **/
        if(!validCity(city))
            throw new CityNotFoundException(....);

        /** Get weather info form processing layer and **/
        /**return results **/
        return (getWeatherInfoFromDataSource(city));
    }

    public void destroy() {....}
}
```

**Code Example 3.15** Performing a Synchronous Client Interaction

Now let’s examine an asynchronous interaction between a client and a service. When making a request for this type of service, the client cannot afford to wait for the response because of the significant time it takes for the service to process the request completely. Instead, the client may want to continue with some other processing. Later, when it receives the response, the client resumes whatever processing initiated the service request. Typically in these types of ser-
Designing a Service’s Interaction Layer

vices, the content of the request parameters initiates and determines the processing workflow—the steps to fulfill the request—for the Web service. Often, fulfilling a request requires multiple workflow steps.

The travel agency service is a good example of an asynchronous interaction between a client and a service. A client requests arrangements for a particular trip by sending the travel service all pertinent information (most likely in an XML document). Based on the document’s content, the service performs such steps as verifying the user’s account, checking and getting authorization for the credit card, checking accommodations and transportation availability, building an itinerary, purchasing tickets, and so forth. Since the travel service must perform a series of often time-consuming steps in its normal workflow, the client cannot afford to pause and wait for these steps to complete.

Figure 3.7 shows one recommended approach for asynchronously delegating these types of Web service requests to the processing layer. In this architecture, the client sends a request to the service endpoint. The service endpoint validates the incoming request in the interaction layer and then delegates the client’s request to the appropriate processing layer of the service. It does so by sending the request as a JMS message to a JMS queue or topic specifically designated for this type of request.

- Delegating a request to the processing layer through JMS before validating the request should be avoided.

Validation ensures that a request is correct. Delegating the request before validation may result in passing an invalid request to the processing layer, making error tracking and error handling overly complex. After the request is successfully delegated to the processing layer, the service endpoint may return a correlation identifier to the client. This correlation identifier is for the client’s future reference and may help the client associate a response that corresponds to its previous request. If the business logic is implemented using enterprise beans, message-driven beans in the EJB tier read the request and initiate processing so that a response can ultimately be formulated.
Figure 3.7  Asynchronous Interaction Between Client and Service

Figure 3.8 shows how the travel agency service might implement this interaction, and Code Example 3.16 shows the actual code that might be used.

Figure 3.8  Travel Agency Service Interaction
In Figure 3.8, the vertical lines represent the passage of time, from top to bottom. The vertical rectangular boxes indicate when the entity (client or service) is busy processing the request or waiting for the other entity to complete processing. The half arrow type indicates asynchronous communication and the dashed vertical line indicates that the entity is free to work on other things while a request is being processed.

```java
public class ReservationRequestRcvr {
    public ReservationRequestRcvr() throws RemoteException {....}

    public String receiveRequest(Source reservationDetails) throws RemoteException, InvalidRequestException{
        /** Validate incoming XML document **/
        String xmlDoc = getDocumentAsString(reservationDetails);
        if(!validDocument(xmlDoc))
            throw new InvalidRequestException(...);

        /** Get a JMS Queue and delegate the incoming request **/
        /** to the queue **/
        QueueConnectionFactory queueFactory =
            serviceLocator.getQueueConnectionFactory(....);
        Queue reservationRequestQueue =
            serviceLocator.getQueue(...);
        QueueConnection connection =
            queueFactory.createQueueConnection();
        QueueSession session = connection.createQueueSession(false,
            Session.AUTO_ACKNOWLEDGE);
        QueueSender queueSender = session.createSender(queue);
        TextMessage message = session.createTextMessage();
        message.setText(xmlDoc);
        queueSender.send(message);
        /** Generate and return a correlation identifier **/
        return generateCorrelationID();
    }
}
```

**Code Example 3.16**  Implementing Travel Agency Service Interaction
One question remains: How does the client get the final result of its request? The service may make the result of the client’s request available in one of two ways:

- The client that invoked the service periodically checks the status of the request using the correlation identifier that was provided at the time the request was submitted. This is also known as polling, and it appears as Option 1 in Figure 3.8.

- Or, if the client itself is a Web service peer, the service calls back the client’s service with the result. The client may use the correlation identifier to relate the response with the original request (Option 2 in Figure 3.8).

Often this is decided by the nature of the service itself. For example, if the service runs a business process workflow, the workflow requires the service to take appropriate action after processing the request.

### 3.4.4 Formulating Responses

After you delegate the request to the business logic portion of the application, and the business logic completes its processing, you are ready for the next step: to form the response to the request.

⚠️ You should perform response generation, which is simply constructing the method call return values and output parameters, on the interaction layer, as close as possible to the service endpoint.

This permits having a common location for response assembly and XML document transformations, particularly if the document you return to the caller must conform to a different schema from the internal schema. Keeping this functionality near the endpoint lets you implement data caching and avoid extra trips to the processing layer. (See Figure 3.9.)

Consider response generation from the weather information service’s point-of-view. The weather information service may be used by a variety of client types, from browsers to rich clients to handheld devices. A well-designed weather information service would render its responses in formats suitable for these different client types.
However, it is not good design to have a different implementation of the service’s logic for each client type. Rather, it is better to design a common business logic for all client types. Then, in the interaction layer, transform the results per client type for rendering. It is thus important to consider the above guidelines, especially when your service has a common processing logic but potentially has different response rendering needs to fit its varied client types.

### 3.5 Processing Layer Design

The processing layer is where the business logic is applied to a Web service request. Recall that Web service is an interoperable way to expose new or existing applications. Hence, regardless of the means you use to expose your application’s functionality, the business logic design issues are the same. You must still design the business logic by considering such issues as using enterprise beans, exposing a local or a remote EJB interface model, using container-managed or bean-managed persistence, and so forth.
The issues and considerations for designing an application’s processing or business logic layer, such as whether to perform this logic in the Web or EJB tier, are the same whether or not you use a Web service.

We do not address these business logic design issues here, since much of this discussion has already been covered in the book *Designing Enterprise Applications with the J2EE™ Platform, Second Edition*, and you can refer to that book for general guidelines and recommendations. You should also refer to the BluePrints Web site at http://java.sun.com/blueprints for recommendations on designing an application’s business processing logic.

In addition to these general guidelines, there are some specific issues to keep in mind when designing the processing layer of a Web service.

- **Keep the processing layer independent of the interaction layer.** By keeping the layers independent and loosely coupled, the processing layer remains generic and can support different types of clients, such as Web service clients, classic Web clients, and so forth. To achieve loose coupling between the layers, consider using delegate classes that encapsulate the access to business components.

- **Bind XML documents to Java objects in the interaction layer.** There are times when your Web service expects to receive from a client an XML document containing a complete request, but the service’s business logic has no need to operate on the document. On these occasions, it is recommended that the interaction layer bind the XML document contents to Java objects before passing the request to the processing layer. Since the processing logic does not have to perform the XML-to-Java conversion, a single processing layer can support XML documents that rely on different schemas. This also makes it easy to support multiple versions of an XML schema.

  Keep in mind that your processing logic can operate on the contents of an XML document received from a client. Refer to “Handling XML Documents in a Web Service” on page 105, which highlights issues to consider when you pass XML documents to your business processing logic.

  Depending on your application scenario, your processing layer may be required to work with other Web service peers to complete a client’s request. If so, your processing layer effectively becomes a client of another Web service. Refer to Chapter 5 for guidelines on Web service clients. In other circumstances, your
processing layer may have to interact with EISs. For these cases, refer to Chapter 6 for guidelines.

### 3.6 Publishing a Web Service

Up to now, this chapter has covered guidelines for designing and implementing your Web service. Just as important, your Web service needs to be accessible to its intended clients. Recall that some Web services are intended for use by the general public. Other Web services are intended to be used only between trusted business partners (inter-enterprise), and still others are intended for use just within an enterprise (intra-enterprise).

Regardless of whether a service is to be accessible to the public, other enterprises, or even within a single enterprise, you must first make the details about the Web service—its interface, parameters, where the service is located, and so forth—accessible to clients. You do so by making a description of the Web service available to interested parties. As noted in “Web Services Description Language” on page 36, WSDL is the standard language for describing a service. Making this WSDL description available to clients enables them to use the service.

Once the WSDL is ready, you have the option to publish it in a registry. The next section describes when you might want to publish the WSDL in a registry. If you make the WSDL description of your service available in a public registry, then a Java-based client can use the JAXR APIs to look up the description of your service and then use the service. For that matter, a client can use the same JAXR APIs to look up the description of any Web service with an available WSDL description. This section examines registries from the point of view of a service developer.

#### 3.6.1 Publishing a Service in a Registry

Publishing a service in a registry is one method of making the service available to clients. If you decide to publish your service in a registry, you decide on the type of registry to use based on the likely usage scenarios for your service. Registries run the gamut from public registries to corporate registries available only within a single enterprise.

- You may want to register Web services for general public consumption on a well-known public registry.
When you make your service available through a public registry, you essentially open the service’s accessibility to the widest possible audience. When a service is registered in a public registry, any client, even one with no prior knowledge of the service, may look up and use the service. Keep in mind that the public registry holds the Web service description, which consists not only of the service’s WSDL description but also any XML schemas referenced by the service description. In short, your Web service must publish its public XML schemas and any additional schemas defined in the context of the service. You also must publish on the same public registry XML schemas referred to by the Web service description.

☐ When a Web service is strictly for intra-enterprise use, you may publish a Web service description on a corporate registry within the enterprise.

☐ You do not need to use a registry if all the customers of your Web services are dedicated partners and there is an agreement among the partners on the use of the services. When this is the case, you can publish your Web service description—the WSDL and referenced XML schemas—at a well-known location with the proper access protections.

3.6.2 Understanding Registry Concepts

When considering whether to publish your service via a registry, it is important to understand some of the concepts, such as repositories and taxonomies, that are associated with registries.

Public registries are not repositories. Rather than containing complete details on services, public registries contain only details about what services are available and how to access these services. For example, a service selling adventure packages cannot register its complete catalog of products. A registry can only store the type of service, its location, and information required to access the service. A client interested in a service must first discover the service from the registry and then bind with the service to obtain the service’s complete catalog of products. Once it obtains the service’s catalog, the client can ascertain whether the particular service meets its needs. If not, the client must go back to the registry and repeat the discovery and binding process—the client looks in the registry for some other service that potentially offers what it wants, binds to that service, obtains and assesses its catalog, and so forth. Since this process, which is not insignificant, may have to be repeated several times, it is easy to see that it is important to register a service under its proper taxonomy.
Register a service under the proper taxonomy.

It is important to register your service under the proper taxonomies. When you want to publish your service on a registry, either a public or corporate registry, you must do so against a taxonomy that correctly classifies or categorizes your Web service. It is important to decide on the proper taxonomy, as this affects the ease with which clients can find and use your service. Several well-defined industry standard taxonomies exist today, such as those defined by organizations such as the North American Industry Classification System (NAICS).

Using existing, well-known taxonomies gives clients of your Web service a standard base from which to search for your service, making it easy for clients to find your service. For example, suppose your travel business provides South Sea island-related adventure packages as well as alpine or mountaineering adventures. Rather than create your own taxonomy to categorize your service, clients can more easily find your service if you publish your service description using two different standard taxonomies: one taxonomy for island adventures and another for alpine and mountaineering adventures.

You can publish your Web service in more than one registry. To further help clients find your service, it is also a good idea to publish in as many applicable categories as possible. For example, a travel business selling adventure packages might register using a product category taxonomy as well as a geographical taxonomy. This gives clients a chance to use optimal strategies for locating a service. For example, if multiple instances of a service exist for a particular product, the client might further refine its selection by considering geographical location and choosing a service close to its own location. Using the travel business service as an example, such a service might register under the taxonomies for types of adventure packages (island and mountaineering), as well as under the taxonomies for the locales in which the adventure packages are provided (Mount Kilimanjaro or Tahiti), thus making it as easy as possible for a prospective client to locate its services.

3.6.3 Registry Implementation Scenarios

Once you decide to publish your service and establish the taxonomies that best identify your service, you are ready to implement your decisions. Before doing so, you may find it helpful to examine some of the registry implementation scenarios that you may encounter.
When a registry is used, we have seen that the service provider publishes the Web service description on a registry and clients discover and bind to the Web service to use its services. In general, a client must perform three steps to use a Web service:

1. The client must determine how to access the service’s methods, such as determining the service method parameters, return values, and so forth. This is referred to as discovering the service definition interface.

2. The client must locate the actual Web service; that is, find the service’s address. This is referred to as discovering the service implementation.

3. The client must be bound to the service’s specific location, and this may occur on one of three occasions:
   - When the client is developed (called static binding)
   - When the client is deployed (also called static binding)
   - During runtime (called dynamic binding)

These three steps may produce three scenarios. The particular scenario depends on when the binding occurs and whether the client is implemented solely for a specific service or is a generic client. The following paragraphs describe these scenarios. (See Table 3.2 for a summary.) They also note important points you should consider when designing and implementing a Web service. (Chapter 5 considers these scenarios from the point of view of a client.)

- Scenario 1: The Web service has an agreement with its partners and publishes its WSDL description and referenced XML schemas at a well-known, specified location. It expects its client developers to know this location. When this is the case, the client is implemented with the service’s interface in mind. When it is built, the client is already designed to look up the service interface directly rather than using a registry to find the service.

- Scenario 2: Similar to scenario 1, the Web service publishes its WSDL description and XML schemas at a well-known location, and it expects its partners to either know this location or be able to discover it easily. Or, when the partner is built, it can use a tool to dynamically discover and then include either the service’s specific implementation or the service’s interface definition, along with its specific implementation. In this case, binding is static because the partner is
built when the service interface definition and implementation are already known to it, even though this information was found dynamically.

- Scenario 3: The service implements an interface at a well-known location, or it expects its clients to use tools to find the interface at build time. Since the Web service’s clients are generic clients—they are not clients designed solely to use this Web service—you must design the service so that it can be registered in a registry. Such generic clients dynamically find a service’s specific implementation at runtime using registries. Choose the type of registry for the service—either public, corporate, or private—depending on the types of its clients—either general public or intra-enterprise—its security constraints, and so forth.

### Table 3.2  
Discovery-Binding Scenarios for Clients

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Discover Service Interface Definition</th>
<th>Discover Service Implementation</th>
<th>Binding to Specific Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>None</td>
<td>None</td>
<td>Static</td>
</tr>
<tr>
<td>2</td>
<td>None or dynamic at build time</td>
<td>Dynamic at build time</td>
<td>Static</td>
</tr>
<tr>
<td>3</td>
<td>None or dynamic at build time</td>
<td>Dynamic at runtime</td>
<td>Dynamic at build time</td>
</tr>
</tbody>
</table>

### 3.7 Handling XML Documents in a Web Service

Up to now, this chapter addressed issues applicable to all Web service implementations. There are additional considerations when a Web service implementation expects to receive an XML document containing all the information from a client, and which the service uses to start a business process to handle the request. There are several reasons why it is appropriate to exchange documents:

- Documents, especially business documents, may be very large, and as such, they are often sent as a batch of related information. They may be compressed independently from the SOAP message.

- Documents may be legally binding business documents. At a minimum, their original form needs to be conserved through the exchange and, more than likely, they may need to be archived and kept as evidence in case of disagreement.
For these documents, the complete infoset of the original document should be preserved, including comments and external entity references (as well as the referred entities).

- Some application processing requires the complete document infoset, including comments and external entity references. As with the legally binding documents, it is necessary to preserve the complete infoset, including comments and external entity references, of the original document.

- When sent as attachments, it is possible to handle documents that may conform to schemas expressed in languages not supported by the Web service endpoint or that are prohibited from being present within a SOAP message infoset (such as the Document Type Declaration \( \text{<!DOCTYPE> } \)) for a DTD-based schema.

For example, consider the travel agency Web service, which typically receives a client request as an XML document containing all information needed to arrange a particular trip. The information in the document includes details about the customer’s account, credit card status, desired travel destinations, preferred airlines, class of travel, dates, and so forth. The Web service uses the documents contents to perform such steps as verifying the customer’s account, obtaining authorization for the credit card, checking accommodations and transportation availability, building an itinerary, and purchasing tickets.

In essence, the service, which receives the request with the XML document, starts a business process to perform a series of steps to complete the request. The contents of the XML document are used throughout the business process. Handling this type of scenario effectively requires some considerations in addition to the general ones for all Web services.

- Good design expects XML documents to be received as `javax.xml.transform.Source` objects. See “Exchanging XML Documents” on page 107, which discusses exchanging XML documents as parameters. Keep in mind the effect on interoperability (see “Interoperability” on page 86).

- It is good design to do the validation and any required transformation of the XML documents as close to the endpoint as possible. Validation and transformation should be done before applying any processing logic to the document content. See Figure 3.4 and the discussion on receiving requests in “Receiving Requests” on page 89.
It is important to consider the processing time for a request and whether the client waits for the response. When a service expects an XML document as input and starts a lengthy business process based on the document contents, then clients typically do not want to wait for the response. Good design when processing time may be extensive is to delegate a request to a JMS queue or topic and return a correlation identifier for the client’s future reference. (Recall Figure 3.7 on page 96 and its discussion.)

The following sections discuss other considerations.

3.7.1 Exchanging XML Documents

As noted earlier, there are times when you may have to exchange XML documents as part of your Web service and such documents are received as parameters of a method call. The J2EE platform provides three ways to exchange XML documents.

The first option is to use the Java-MIME mappings provided by the J2EE platform. See Table 3.1 on page 75. With this option, the Web service endpoint receives documents as `javax.xml.transform.Source` objects. (See Code Example 3.3 on page 75.) Along with the document, the service endpoint can also expect to receive other JAX-RPC arguments containing metadata, processing requirements, security information, and so forth. When an XML document is passed as a `Source` object, the container automatically handles the document as an attachment—effectively, the container implementation handles the document-passing details for you. This frees you from the intricacies of sending and retrieving documents as part of the endpoint’s request/response handling.

Passing XML documents as `Source` objects is the most effective option in a completely Java-based environment (one in which all Web service clients are based on Java). However, sending documents as `Source` objects may not be interoperable with non-Java clients. (As already noted in the section “Interoperability” on page 86, standard ways to exchange attachments are currently being formulated. Future versions of the J2EE platform will incorporate these standards once they are final.)

The second option is to design your service endpoint such that it receives documents as `String` types. Code Example 3.17 shows the WSDL description for a service that receives documents as `String` types, illustrating how the WSDL maps the XML document.
<?xml version="1.0" encoding="UTF-8"?>
<definitions ...>
  <types/>
  <message name="PurchaseOrderService_submitPurchaseOrder">
    <part name="PurchaseOrderXMLDoc" type="xsd:string"/>
  </message>
  <message name="PurchaseOrderService_submitPurchaseOrderResponse">
    <part name="result" type="xsd:string"/>
  </message>
  <portType name="PurchaseOrderService">
    <operation name="submitPurchaseOrder" parameterOrder="PurchaseOrderXMLDoc">
      <input message="tns:PurchaseOrderService_submitPurchaseOrder"/>
      <output message="tns:PurchaseOrderService_submitPurchaseOrderResponse"/>
    </operation>
  </portType>
  ...
</definitions>

**Code Example 3.17** Mapping XML Document to `xsd:string`

Code Example 3.18 shows the equivalent Java interface for the WSDL shown in Code Example 3.17.

```java
public interface PurchaseOrderService extends Remote {
    public String submitPurchaseOrder(String poDocument)
        throws RemoteException, InvalidOrderException;
}
```

**Code Example 3.18** Receiving an XML Document as a `String` object

If you are developing your service using the Java-to-WSDL approach, and the service must exchange XML documents and be interoperable with clients on any platform, then passing documents as `String` objects may be your only option.
There may be a performance drawback to sending an XML document as a String object: As the document size grows, the String equivalent size of the document grows as well. As a result, the payload size of the message you send also grows. In addition, the XML document loses its original format since sending a document as a String object sends it in a canonical format.

The third option is to exchange the XML document as a SOAP document fragment. With this option, you map the XML document to xsd:anyType in the service’s WSDL file.

It is recommended that Web services exchange XML documents as SOAP document fragments because passing XML documents in this manner is both portable across J2EE implementations and interoperable with all platforms.

To pass SOAP document fragments, you must implement your service using the WSDL-to-Java approach.

For example, the travel agency service receives an XML document representing a purchase order that contains all details about the customer’s preferred travel plans. To implement this service, you define the WSDL for the service and, in the WSDL, you map the XML document type as xsd:anyType. See Code Example 3.19.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions ...>
  <types/>
  <message name="PurchaseOrderService_submitPurchaseOrder">
    <part name="PurchaseOrderXMLDoc" type="xsd:anyType"/>
  </message>
  <message name="PurchaseOrderService_submitPurchaseOrderResponse">
    <part name="result" type="xsd:string"/>
  </message>
  <portType name="PurchaseOrderService">
    <operation name="submitPurchaseOrder"
      parameterOrder="PurchaseOrderXMLDoc">
      <input message="tns:PurchaseOrderService_submitPurchaseOrder"/>
      <output message=
```
"tns:PurchaseOrderService_submitPurchaseOrderResponse"/>
</operation>
</portType>
...
</definitions>

**Code Example 3.19**  Mapping XML document to xsd:anyType

A WSDL mapping of the XML document type to xsd:anyType requires the platform to map the document parameter as a javax.xml.soap.SOAPElement object. For example, Code Example 3.20 shows the Java interface generated for the WSDL description in Code Example 3.19.

```java
public interface PurchaseOrderService extends Remote {
    public String submitPurchaseOrder(SOAPElement purchaseOrderXMLDoc) throws RemoteException;
}
```

**Code Example 3.20**  Java Interface for WSDL in Code Example 3.19

In this example, the SOAPElement parameter in submitPurchaseOrder represents the SOAP document fragment sent by the client. For the travel agency service, this is the purchase order. The service can parse the received SOAP document fragment using the javax.xml.soap.SOAPElement API. Or, the service can use JAXB to map the document fragment to a Java Object or transform it to another schema. A client of this Web service builds the purchase order document using the client platform-specific API for building SOAP document fragments—on the Java platform, this is the javax.xml.soap.SOAPElement API—and sends the document as one of the Web service’s call parameters.

When using the WSDL-to-Java approach, you can directly map the document to be exchanged to its appropriate schema in the WSDL. The corresponding generated Java interface represents the document as its equivalent Java Object. As a result, the service endpoint never sees the document that is exchanged in its original document form. It also means that the endpoint is tightly coupled to the document’s schema: Any change in the document’s schema requires a corresponding change to the endpoint. If you do not want such tight coupling, consider using xsd:anyType to map the document.
3.7.2 Separating Document Manipulation from Processing Logic

When your service’s business logic operates on the contents of an incoming XML document, the business processing logic must at a minimum read the document, if not modify the document. By separating the document manipulation logic from the processing logic, a developer can switch between various document manipulation mechanisms without affecting the processing logic. In addition, there is a clear division between developer skills.

 الوزن It is a good practice to separate the XML document manipulation logic from the business logic.

The “Abstracting XML Processing from Application Logic” section on page 155 provides more information on how to accomplish this separation and its merits.

3.7.3 Fragmenting XML Documents

When your service’s business logic operates on the contents of an incoming XML document, it is a good idea to break XML documents into logical fragments when appropriate. When the processing logic receives an XML document that contains all information for processing a request, the XML document usually has well-defined segments for different entities, and each segment contains the details about a specific entity.

وزن Rather than pass the entire document to different components handling various stages of the business process, it’s best if the processing logic breaks the document into fragments and passes only the required fragments to other components or services that implement portions of the business process logic.

See “Fragmenting Incoming XML Documents” on page 153 for more details on fragmentation.

3.7.4 Using XML

XML, while it has many benefits, also has performance disadvantages. You should weigh the trade-offs of passing XML documents through the business logic processing stages. The pros and cons of passing XML documents take on greater significance when the business logic implementation spans multiple containers. Refer to
Chapter 5, specifically the section entitled “Use XML Judiciously” on page 194, which provides guidelines on this issue. Following these guidelines may help minimize the performance overhead that comes with passing XML documents through workflow stages.

Also, when deciding on an approach, keep in mind the costs involved for using XML and weigh them along with the recommendations on parsing, validation, and binding documents to Java objects. See Chapter 4 for a discussion of these topics.

3.7.5 Using JAXM and SAAJ Technologies

The J2EE platform provides an array of technologies—including mandatory technologies such as JAX-RPC and SAAJ and optional technologies such as Java API for XML Messaging (JAXM)—that enable message and document exchanges with SOAP. Each of these J2EE technologies offers a different level of support for SOAP-based messaging and communication. (See Chapter 2 for the discussion on JAX-RPC and SAAJ.)

An obvious question that arises is: Why not use JAXM or SAAJ technologies in scenarios where you have to pass XML documents? If you recall:

- SAAJ lets developers deal directly with SOAP messages, and is best suited for point-to-point messaging environments. SAAJ is better for developers who want more control over the SOAP messages being exchanged and for developers using handlers.
- JAXM defines an infrastructure for guaranteed delivery of messages. It provides a way of sending and receiving XML documents and guaranteeing their receipt, and is designed for use cases that involve storing and forwarding XML documents and messages.

SAAJ is considered more useful for advanced developers who thoroughly know the technology and who must deal directly with SOAP messages.

Using JAXM for scenarios that require passing XML documents may be a good choice. Note, though, that JAXM is optional in the J2EE 1.4 platform. As a result, a service developed with JAXM may not be portable. When you control both end points of a Web service, it may make more sense to consider using JAXM.
3.8 Deploying and Packaging a Service Endpoint

Up to now, we have examined Web services on the J2EE platform in terms of design, development, and implementation. Once you complete the Web services implementation, you must write its deployment descriptors, package the service with all its components, and deploy the service.

Developers should, if at all possible, use tools or IDEs to develop a Web service. These Web service development tools and IDEs automatically create the proper deployment descriptors for the service and correctly handle the packaging of the service—steps necessary for a service to operate properly. Furthermore, tools and IDEs hide these details from the developer.

Although you can expect your development tool to perform these tasks for you, it is good to have a conceptual understanding of the J2EE 1.4 platform deployment descriptor and packaging structure, since they determine how a service is deployed on a J2EE server and the service’s availability to clients. This section, which provides a conceptual overview of the deployment and packaging details, is not essential reading. Nonetheless, you may find it worthwhile to see how these details contribute to portable, interoperable Web services.

3.8.1 Service Information in the Deployment Descriptors

To successfully deploy a service, the developer provides the following information.

- Deployment-related details of the service implementation, including the Web service interface, the classes that implement the Web service interface, and so forth.
- Details about the Web services to be deployed, such as the ports and mappings
- Details on the WSDL port-to-port component relationship

More specifically, the deployment descriptor contains information about a service’s port and associated WSDL. Recall from “Web Service Technologies Integrated in J2EE Platform” on page 49:
Chapter 3  Service Endpoint Design

- A port component (also called a port) gives a view of the service to clients such that the client need not worry about how the service has been implemented.
- Each port has an associated WSDL.
- Each port has an associated service endpoint (and its implementation). The endpoint services all requests that pass through the location defined in the WSDL port address.

To begin, the service implementation declares its deployment details in the appropriate module-specific deployment descriptors. For example, a service implementation that uses a JAX-RPC service endpoint declares its details in the WEB-INF/web.xml file using the servlet-class element. (See Code Example 3.21.)

```xml
<web-app ...>
  ...
  <servlet>
    <description>Endpoint for Some Web Service</description>
    <display-name>SomeWebService</display-name>
    <servlet-name>SomeService</servlet-name>
    <servlet-class>com.a.b.c.SomeServiceImpl</servlet-class>
    <load-on-startup>0</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>SomeService</servlet-name>
    <url-pattern>/webservice/SomeService</url-pattern>
  </servlet-mapping>
  ...
</web-app>
```

**Code Example 3.21**  web.xml File for a JAX-RPC Service Endpoint

Note that when you have a service that functions purely as a Web service using JAX-RPC service endpoints, some specifications in the web.xml file, such as <error-page> and <welcome-file-list>, have no effect.

A service implementation that uses an EJB service endpoint declares its deployment details in the file META-INF/ejb-jar.xml using the session element. (See Code Example 3.22.)
<ejb-jar ...>
  <display-name>Some Enterprise Bean</display-name>
  <enterprise-beans>
    <session>
      <ejb-name>SomeBean</ejb-name>
      <service-endpoint>com.a.b.c.SomeIntf</service-endpoint>
      <ejb-class>com.a.b.c.SomeServiceEJB</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  ...
</ejb-jar>

**Code Example 3.22** .ejb-jar.xml File for an EJB Service Endpoint

Next, the details of the port are specified. The Web service deployment descriptor, called webservices.xml, defines and declares the structural details for the port of a Web service. This file contains the following information:

- A logical name for the port that is also unique among all port components (port-component-name element)
- The service endpoint interface for the port (service-endpoint-interface element)
- The name of the class that implements the service interface (service-impl-bean element)
- The WSDL file for the service (wsdl-file element)
- A QName for the port (wsdl-port element)
- A correlation between WSDL definitions and actual Java interfaces and definitions using the mapping file (jaxrpc-mapping-file element)
- Optional details on any handlers

The reference to the service implementation bean, specified using the service-impl-bean element in webservices.xml, is either a servlet-link or an ejb-link depending on whether the endpoint is a JAX-RPC or EJB service end-
point. This link element associates the Web service port to the actual endpoint implementation defined in either the web.xml or ejb-jar.xml file.

The JAX-RPC mapping file, which is specified using the jaxrpc-mapping-file element in webservices.xml, keeps details on the relationships and mappings between WSDL definitions and corresponding Java interfaces and definitions. The information contained in this file, along with information in the WSDL, is used to create stubs and ties for deployed services.

Thus, the Web services deployment descriptor, webservices.xml, links the WSDL port information to a unique port component and from there to the actual implementation classes and Java-to-WSDL mappings. Code Example 3.23 is an example of the Web services deployment descriptor for our sample weather Web service, which uses a JAX-RPC service endpoint.

<webservices ...>
   <description>Web Service Descriptor for weather service</description>
   <webservice-description>
      <webservice-description-name>WeatherWebService</webservice-description-name>
      <wsdl-file>WEB-INF/wsdl/WeatherWebService.wsdl</wsdl-file>
      <jaxrpc-mapping-file>WEB-INF/WeatherWebServiceMapping.xml</jaxrpc-mapping-file>
      <port-component>
         <description>port component description</description>
         <port-component-name>WeatherServicePort</port-component-name>
         <wsdl-port xmlns:weatherns="urn:WeatherWebService" weatherns:WeatherServicePort>
         </wsdl-port>
         <service-endpoint-interface>endpoint.WeatherService</service-endpoint-interface>
         <service-impl-bean>
            <servlet-link>WeatherService</servlet-link>
         </service-impl-bean>
   </port-component>
</webservice-description>
</webservices>
3.8.2 Package Structure

Once the service implementation and deployment descriptors are completed, the following files should be packaged into the appropriate J2EE module:

- The WSDL file
- The service endpoint interface, including its implementation and dependent classes
- The JAX-RPC mapping file, which specifies the package name containing the generated runtime classes and defines the namespace URI for the service. See Code Example 5.21 on page 242.
- The Web service deployment descriptor

The type of endpoint used for the service implementation determines the type of the J2EE module to use.

☐ The appropriate J2EE module for a service with a JAX-RPC service endpoint is a WAR file. A service using an EJB service endpoint must be packaged in an EJB-JAR file.

The package structure is as follows:

- WSDL files are located relative to the root of the module.
- The service interface, the service implementation classes, and the dependent classes are packaged just like any other J2EE component.
- The JAX-RPC mapping file is located relative to the root of the module (typically in the same place as the module’s deployment descriptor).
• The Web service deployment descriptor location depends on the type of service endpoint, as follows:

  • For an EJB service endpoint, the Web service deployment descriptor is packaged in an EJB-JAR in the META-INF directory as `META-INF/webservice.xml`

  • For a JAX-RPC service endpoint, the deployment descriptor is packaged in a WAR file in the WEB-INF directory as `WEB-INF/webservices.xml`.

See Figure 3.10, which shows a typical package structure for a Web service using an EJB endpoint. Figure 3.11 shows the typical structure for a Web service using a JAX-RPC endpoint.

![Figure 3.10 Package Structure for EJB Endpoint](image)
This chapter began with a description of Web service fundamentals. It described the underlying flow of a typical Web service on the Java platform, showing how the various components making up clients and services pass requests and responses among themselves. The chapter also described some example scenarios, which it used to illustrate various concepts. Once the groundwork was set, the chapter discussed the key design decisions that a Web service developer needs to make, principally the design of a service as an interaction and a processing layer. It traced how to go about making design decisions and recommending good design choices for specific scenarios.

The next chapter focuses on developing Web service clients.
CHAPTER 7

BUSINESS TIER DESIGN CONSIDERATIONS AND BAD PRACTICES

Excerpted from: Core J2EE Patterns: Best Practices and Design Strategies, Second Edition by Deepak Alur, Dan Malks, John Crupi
Business Tier Design Considerations and Bad Practices

Topics in This Chapter

- Business Tier Design Considerations
- Business and Integration Tiers Bad Practices
Chapter 3

Business Tier Design Considerations
Business Tier Design Considerations

When you apply the business tier and integration tier patterns in this book, you’ll need to know about related design issues, which we cover in this chapter. These issues cover a variety of topics, and can affect many aspects of a system.

The discussions in this chapter simply describe each issue as a design issue that you should consider when implementing systems based on the J2EE pattern catalog.

Using Session Beans

Session beans are distributed business components with the following characteristics, per the EJB specification:

- A session bean is dedicated to a single client or user.
- A session bean lives only for the duration of the client's session.
- A session bean does not survive container crashes.
- A session bean is not a persistent object.
- A session bean can time out.
- A session bean can be transaction aware.
- A session bean can be used to model stateful or stateless conversations between the client and the business tier-components.

Note: In this section, we use the term workflow in the context of EJB to represent the logic associated with the enterprise beans communication. For example, workflow encompasses how session bean A calls session bean B, then entity bean C.

Session Bean—Stateless Versus Stateful

Session beans come in two flavors—stateless and stateful. A stateless session bean does not hold any conversational state. Hence, once a client’s method invocation on a stateless session beans is completed, the container is free to reuse that session bean instance for another client. This allows the container to maintain a pool of session beans and to reuse session beans among multiple clients. The container pools stateless session beans so that it can reuse them more efficiently by sharing them with multiple clients. The container returns a stateless session bean
to the pool after the client completes its invocation. The container may allocate a
different instance from the pool to subsequent client invocations.

A stateful session bean holds conversational state. A stateful session bean may
be pooled, but since the session bean is holding state on behalf of a client, the
bean cannot simultaneously be shared with and handle requests from another cli-
ent.

The container does not pool stateful session beans in the same manner as it
pools stateless session beans because stateful session beans hold client session
state. Stateful session beans are allocated to a client and remain allocated to the
client as long as the client session is active. Thus, stateful session beans need
more resource overhead than stateless session beans, for the added advantage of
maintaining conversational state.

Many designers believe that using stateless session beans is a more viable ses-
sion bean design strategy for scalable systems. This belief stems from building
distributed object systems with older technologies, because without an inherent
infrastructure to manage component life cycle, such systems rapidly lost scalabil-
ity characteristics as resource demands increased. Scalability loss was due to the
lack of component life cycle, causing the service to continue to consume
resources as the number of clients and objects increased.

An EJB container manages the life cycle of enterprise beans and is responsible
for monitoring system resources to best manage enterprise bean instances. The
container manages a pool of enterprise beans and brings enterprise beans in and
out of memory (called activation and passivation, respectively) to optimize
invocation and resource consumption.

Scalability problems are typically due to the misapplication of stateful and
stateless session beans. The choice of using stateful or stateless session beans
must depend upon the business process being implemented. A business process
that needs only one method call to complete the service is a non-conversational
business process. Such processes are suitably implemented using a stateless ses-
sion bean. A business process that needs multiple method calls to complete the
service is a conversational business process. It is suitably implemented using a
stateful session bean.

However, some designers choose stateless session beans, hoping to increase
scalability, and they may wrongly decide to model all business processes as state-
less session beans. When using stateless session beans for conversational busi-
ness processes, every method invocation requires the state to be passed by the
client to the bean, reconstructed at the business tier, or retrieved from a persistent
store. These techniques could result in reduced scalability due to the associated
overheads in network traffic, reconstruction time, or access time respectively.
Storing State on the Business Tier

Some design considerations for storing state on the Web server are discussed in “Session State in the Presentation Tier” on page 21. Here we continue that discussion to explore when it is appropriate to store state in a stateful session bean instead of in an HttpSession.

One of the considerations is to determine what types of clients access the business services in your system. If the architecture is solely a Web-based application, where all the clients come through a Web server either via a servlet or a JSP, then conversational state may be maintained in an HttpSession in the web tier. This scenario is shown in Figure 3.1.

On the other hand, if your application supports various types of clients, including Web clients, Java applications, other applications, and even other enterprise beans, then conversational state can be maintained in the EJB layer using stateful session beans. This is shown in Figure 3.2.

![Figure 3.1 Storing State in HttpSession](image-url)
We have presented some basic discussion on the subject of state management here and in the previous chapter (see “Session State on the Client” on page 20). A full-scale discussion is outside the scope of this book, since the problem is multi-dimensional and depends very much on the deployment environment, including:

- Hardware
- Traffic management
- Clustering of Web container
- Clustering of EJB container
- Server affinity
- Session replication
- Session persistence

We touch on this issue because it is one that should be considered during development and deployment.

**Using Entity Beans**

Using entity beans appropriately is a question of design heuristics, experience, need, and technology. Entity beans are best suited as coarse-grained business components. Entity beans are distributed objects and have the following characteristics, per the EJB specification:

- Entity beans provide an object view of persistent data.
- Entity beans are transactional.
Chapter 3 Business Tier Design Considerations and Bad Practices

- Entity beans are multiuser.
- Entity beans are long-lived.
- Entity beans survive container crashes. Such crashes are typically transparent to the clients.

Summarizing this definition, the appropriate use of an entity bean is as a distributed, shared, transactional, and persistent object. In addition, EJB containers provide other infrastructure necessary to support such system qualities as scalability, security, performance, clustering, and so forth. All together, this makes for a very reliable and robust platform to implement and deploy applications with distributed business components.

Entity Bean Primary Keys

Entity beans are uniquely identified by their primary keys. A primary key can be a simple key, made up of a single attribute, or it can be a composite key, made up of a group of attributes from the entity bean. For entity beans with a single-field primary key, where the primary key is a primitive type, it is possible to implement the entity bean without defining an explicit primary key class. The deployer can specify the primary key field in the deployment descriptor for the entity bean. However, when the primary key is a composite key, a separate class for the primary key must be specified. This class must be a simple Java class that implements the serializable interface with the attributes that define the composite key for the entity bean. The attribute names and types in the primary key class must match those in the entity bean, and also must be declared public in both the bean implementation class and primary key class.

As a suggested best practice, the primary key class must implement the optional `java.lang.Object` methods, such as `equals` and `hashCode`.

- Override the `equals()` method to properly evaluate the equality of two primary keys by comparing values for each part of the composite key.
- Override the `Object.hashCode()` method to return a unique number representing the hash code for the primary key instance. Ensure that the hash code is indeed unique when you use your primary key attribute values to compute the hash code.

Business Logic in Entity Beans

A common question in entity bean design is what kind of business logic it should contain. Some designers feel that entity beans should contain only persistence logic and simple methods to get and set data values. They feel that entity beans
should not contain business logic, which is often misunderstood to mean that only code related to getting and setting data must be included in the entity bean.

Business logic generally includes any logic associated with providing some service. For this discussion, consider business logic to include all logic related to processing, workflow, business rules, data, and so forth. The following is a list of sample questions to explore the possible results of adding business logic into an entity:

- Will the business logic introduce entity-entity relationships?
- Will the entity bean become responsible for managing workflow of user interaction?
- Will the entity bean take on the responsibilities that should belong in some other business component?

A “yes” answer to any of these questions helps identify whether introducing business logic into the entity bean can have an adverse impact, especially if you use remote entity beans. It is desirable to investigate the design to avoid inter-entity-bean dependencies as much as possible, since such dependences create overheads that may impede overall application performance.

In general, the entity bean should contain business logic that is self-contained to manage its data and its dependent objects’ data. Thus, it may be necessary to identify, extract, and move business logic that introduces entity-bean-to-entity-bean interaction from the entity bean into a session bean by applying the Session Façade pattern. The Composite Entity (394) pattern and some of the refactorings discuss the issues related to entity bean design.

If any workflow associated with multiple entity beans is identified, then such workflow can be suitably implemented in a session bean instead of in an entity bean. Such workflow can be consolidated into a Session Façade (343) or Application Service (359).

- See “Merge Session Beans” on page 97.
- See “Reduce Inter-Entity Bean Communication” on page 99.
- See “Move Business Logic to Session” on page 101.
- See Session Façade (343)
- See Business Object (377)
- See Composite Entity (394)
- See Application Service (359)

For bean-managed persistence in entity beans, data access code is best implemented outside entity beans.
• See “Separate Data Access Code” on page 103.
• See Data Access Object (464).

Caching Enterprise Bean Remote References and Handles

When clients use an enterprise bean, they might need to cache some reference to an enterprise bean for future use. You will encounter this when using business delegates (see *Business Delegate* (304)), where a delegate connects to a session bean and invokes the necessary business methods on the bean on behalf of the client.

When the client uses the business delegate for the first time, the delegate needs to perform a lookup using the EJB Home object to obtain a remote reference to the session bean. For subsequent requests, the business delegate can avoid lookups by caching a remote reference or its handle as necessary.

The EJB Home handle can also be cached to avoid an additional Java Naming and Directory Interface (JNDI) lookup for the enterprise bean home. For more details on using an EJB Handle or the EJB Home Handle, please refer to the current EJB specification.
Business and Integration Tiers Bad Practices

Mapping the Object Model Directly to the Entity Bean Model

Problem Summary
One of the common practices in designing an EJB application is to map the object model directly into entity beans; that is, each class in the object model is transformed into an entity bean. This results in a large number of fine-grained entity beans.

The container and network overhead increases as the number of enterprise beans increases. Such mapping also transforms object relationships into entity-bean-to-entity-bean relationships. This is best avoided, since entity-bean-to-entity-bean relationships introduce severe performance implications for remote entity beans.

Solution Reference
Identify the parent-dependent object relationships in the object model and design them as coarse-grained entity beans. This results in fewer entity beans, where each entity bean composes a group of related objects from the object model.

- **Refactoring**  See “Reduce Inter-Entity Bean Communication” on page 99.
- **Pattern**  See *Composite Entity* (394)

  Consolidate related workflow operations into session beans to provide a uniform coarse-grained service access layer.

- **Refactoring**  See “Merge Session Beans” on page 97.
- **Pattern**  See *Session Façade* (343)

Mapping the Relational Model Directly to the Entity Bean Model

Problem Summary
When designing an EJB model, it is bad practice to model each row in a table as an entity bean. While entity beans are best designed as coarse-grained objects,
this mapping results in a large number of fine-grained entity beans, and it affects scalability.

Such mapping also implements inter-table relationships that is, primary key/foreign key relationships) as entity-bean-to-entity-bean relationships.

Solution Reference

Design your enterprise bean application using an object-oriented approach instead of relying on the preexisting relational database design to produce the EJB model.

- **Bad Practice** See solution reference for “Mapping the Object Model Directly to the Entity Bean Model” on page 53.

Avoid inter-entity relationships by designing coarse-grained business objects by identifying parent-dependent objects.

- **Refactoring** See “Reduce Inter-Entity Bean Communication” on page 99.
- **Refactoring** See “Move Business Logic to Session” on page 101.
- **Pattern** See *Composite Entity* (394)

Mapping Each Use Case to a Session Bean

Problem Summary

Some designers implement each use case with its own unique session bean. This creates fine-grained controllers responsible for servicing only one type of interaction. The drawback of this approach is that it can result in a large number of session beans and significantly increase the complexity of the application.

Solution Reference

Apply the *Session Façade* pattern to aggregate a group of the related interactions into a single session bean. This results in fewer session beans for the application, and leverages the advantages of applying the *Session Façade* pattern.

- **Refactoring** See “Merge Session Beans” on page 97.
- **Pattern** See *Session Façade* (343)
**Exposing All Enterprise Bean Attributes via Getter/Setter Methods**

**Problem Summary**
Exposing each enterprise bean attribute using getter/setter methods is a bad practice. This forces the client to invoke numerous fine-grained remote invocations and creates the potential to introduce a significant amount of network chattiness across the tiers. Each method call is potentially remote and carries with it a certain network overhead that impacts performance and scalability.

**Solution Reference**
Use a value object to transfer aggregate data to and from the client instead of exposing the getters and setters for each attribute.

- **Pattern** See *Transfer Object* (418).

**Embedding Service Lookup in Clients**

**Problem Summary**
Clients and presentation-tier objects frequently need to look up the enterprise beans. In an EJB environment, the container uses JNDI to provide this service.

Putting the burden of locating services on the application client can introduce a proliferation of lookup code in the application code. Any change to the lookup code propagates to all clients that look up the services. Also, embedding lookup code in clients exposes them to the complexity of the underlying implementation and introduces dependency on the lookup code.

**Solution Reference**
Encapsulate implementation details of the lookup mechanisms using a *Service Locator* (317).

- **Pattern** See *Service Locator* (317)

Encapsulate the implementation details of business tier-components, such as session and entity beans, using *Business Delegate* (304). This simplifies client code since they no longer deal with enterprise beans and services. *Business Delegate* (304) can in turn use the *Service Locator* (317).

- **Refactoring** See “Introduce Business Delegate” on page 95.
- **Pattern** See *Business Delegate* (304).
Using Entity Beans as Read-Only Objects

Problem Summary

Any entity bean method is subject to transaction semantics based on its transaction isolation levels specified in the deployment descriptor. Using an entity bean as a read-only object simply wastes expensive resources and results in unnecessary update transactions to the persistent store. This is due to the invocation of the `ejbStore()` methods by the container during the entity bean’s life cycle. Since the container has no way of knowing if the data was changed during a method invocation, it must assume that it has and invoke the `ejbStore()` operation. Thus, the container makes no distinction between read-only and read-write entity beans. However, some containers may provide read-only entity beans, but these are vendor proprietary implementations.

Solution Reference

Encapsulate all access to the data source using Data Access Object (464) pattern. This provides a centralized layer of data access code and also simplifies entity bean code.

- **Pattern** See Data Access Object (464).

  Implement access to read-only functionality using a session bean, typically as a Session Façade that uses a DAO.

- **Pattern** See Session Façade (343)

  You can implement Value List Handler (447) to obtain a list of Transfer Objects (418).

  - **Pattern** See Value List Handler (447).

    You can implement Transfer Objects (418) to obtain a complex data model from the business tier.

  - **Pattern** See Transfer Object Assembler (436).

Using Entity Beans as Fine-Grained Objects

Problem Summary

Entity beans are meant to represent coarse-grained transactional persistent business components. Using a remote entity bean to represent fine-grained objects increases the overall network communication and container overhead. This impacts application performance and scalability.
Think of a fine-grained object as an object that has little meaning without its association to another object (typically a coarse-grained parent object). For example, an item object can be thought of as a fine-grained object because it has little value until it is associated with an order object. In this example, the order object is the coarse-grained object and the item object is the fine-grained (dependent) object.

**Solution Reference**

When designing enterprise beans based on a preexisting RDBMS schema,

- **Bad Practice** See “Mapping the Relational Model Directly to the Entity Bean Model” on page 54.

When designing enterprise beans using an object model,

- **Bad Practice** See “Mapping the Object Model Directly to the Entity Bean Model” on page 53.

Design coarse-grained entity beans and session beans. Apply the following patterns and refactorings that promote coarse-grained enterprise beans design.

- **Pattern** See Composite Entity (394).
- **Pattern** See Session Façade (343).
- **Refactoring** See “Reduce Inter-Entity Bean Communication” on page 99.
- **Refactoring** See “Move Business Logic to Session” on page 101.
- **Refactoring** See “Business Logic in Entity Beans” on page 50.
- **Refactoring** See “Merge Session Beans” on page 97.

**Storing Entire Entity Bean-Dependent Object Graph**

**Problem Summary**

When a complex tree structure of dependent objects is used in an entity bean, performance can degrade rapidly when loading and storing an entire tree of dependent objects. When the container invokes the entity bean’s `ejbLoad()` method, either for the initial load or for reloads to synchronize with the persistent store, loading the entire tree of dependent objects can prove wasteful. Similarly, when the container invokes the entity bean’s `ejbStore()` method at any time, storing the entire tree of objects can be quite expensive and unnecessary.
Solution Reference

Identify the dependent objects that have changed since the previous store operation and store only those objects to the persistent store.

- Pattern See Composite Entity (394) and Store Optimization (Dirty Marker) Strategy (400).

Implement a strategy to load only data that is most accessed and required. Load the remaining dependent objects on demand.

- Pattern See Composite Entity (394) and Lazy Loading Strategy on page 399.

By applying these strategies, it is possible to prevent loading and storing an entire tree of dependent objects.

Exposing EJB-related Exceptions to Non-EJB Clients

Problem Summary

Enterprise beans can throw business application exceptions to clients. When an application throws an application exception, the container simply throws the exception to the client. This allows the client to gracefully handle the exception and possibly take another action. It is reasonable to expect the application developer to understand and handle such application-level exceptions.

However, despite employing such good programming practices as designing and using application exceptions, the clients may still receive EJB-related exceptions, such as a java.rmi.RemoteException. This can happen if the enterprise bean or the container encounters a system failure related to the enterprise bean.

The burden is on the application developer, who may not even be aware of or knowledgeable about EJB exceptions and semantics, to understand the implementation details of the non-application exceptions that may be thrown by business tier-components. In addition, non-application exceptions may not provide relevant information to help the user rectify the problem.

Solution Reference

Decouple the clients from the business tier and hide the business-tier implementation details from clients, using business delegates. Business delegates intercept all service exceptions and may throw an application exception. Business delegates are plain Java objects that are local to the client. Typically, business delegates are developed by the EJB developers and provided to the client developers.
Using Entity Bean Finder Methods to Return a Large Results Set

Problem Summary

Frequently, applications require the ability to search and obtain a list of values. Using an EJB finder method to look up a large collection of entity beans will return a collection of remote references. Consequently, the client has to invoke a method on each remote reference to get the data. This is a remote call and can become very expensive, especially impacting performance, when the caller invokes remote calls on each entity bean reference in the collection.

Solution Reference

Implement queries using session beans and DAOs to obtain a list of Transfer Objects (418) instead of remote references. Use a DAO to perform searches instead of EJB finder methods.

Client Aggregates Data from Business Components

Problem Summary

The application clients (in the client or presentation tier) typically need the data model for the application from the business tier. Since the model is implemented by business components—such as entity beans, session beans, and arbitrary objects in the business tier—the client must locate, interact with, and extract the necessary data from various business components to construct the data model.

These client actions introduce network overhead due to multiple invocations from the client into the business tier. In addition, the client becomes tightly coupled with the application model. In applications where there are various types of clients, this coupling problem multiplies: A change to the model requires changes to all clients that contain code to interact with those model elements comprised of business components.
Solution Reference

Decouple the client from model construction. Implement a business-tier component that is responsible for the construction of the required application model.

- **Pattern** See *Transfer Object Assembler* (436).

Using Enterprise Beans for Long-Lived Transactions

**Problem Summary**

Enterprise beans (pre-EJB 2.0) are suitable for synchronous processing. Furthermore, enterprise beans do well if each method implemented in a bean produces an outcome within a predictable and acceptable time period.

If an enterprise bean method takes a significant amount of time to process a client request, or if it blocks while processing, this also blocks the container resources, such as memory and threads, used by the bean. This can severely impact performance and deplete system resources.

An enterprise bean transaction that takes a long time to complete potentially locks out resources from other enterprise bean instances that need those resources, resulting in performance bottlenecks.

**Solution Reference**

Implement asynchronous processing service using a message-oriented middleware (MOM) with a Java Message Service (JMS) API to facilitate long-lived transactions.

- **Pattern** See “Service Activator” on page 498.

Stateless Session Bean Reconstructs Conversational State for Each Invocation

**Problem Summary**

Some designers choose stateless session beans to increase scalability. They may inadvertently decide to model all business processes as stateless session beans even though the session beans require conversational state. But, since the session bean is stateless, it must rebuild conversational state in every method invocation. The state may have to be rebuilt by retrieving data from a database. This com-
completely defeats the purpose of using stateless session beans to improve performance and scalability and can severely degrade performance.

Solution Reference

Analyze the interaction model before choosing the stateless session bean mode. The choice of stateful or stateless session bean depends on the need for maintaining conversational state across method invocations in stateful session bean versus the cost of rebuilding the state during each invocation in stateless session bean.

- **Pattern** See *Transfer Object Assembler* (436), *Stateless Session Façade Strategy* on page 347, and *Stateful Session Façade Strategy* on page 347.
- **Design** See “Session Bean—Stateless Versus Stateful” on page 46 and “Storing State on the Business Tier” on page 48.
CHAPTER 8

JAVA APPLICATION SERVER PERFORMANCE

Excerpted from: Performance Analysis for Java Websites by Stacy Joines, Ruth Willenborg, Ken Hygh
Chapter 2
JAVA APPLICATION
SERVER PERFORMANCE

In Chapter 1, we discussed how code optimization requires an investment in understanding the code itself. If you’re wondering if this investment is worth the potential dividends, keep this in mind: The Java web application, more than any other single factor, determines web site performance. (Tuning the site hardware or the application server rarely compensates for an underperforming application.) Frequently we find web site teams trying to tune their way around a poorly written or poorly designed web application. If these flaws prove severe, the web site cannot perform or scale until the underlying software receives the attention it needs from the development team.

The next three chapters cover the basics of Java web applications and the Java web site environment. We begin with a brief discussion of Java web application design and an overview of how Java web applications interact with Java web application servers. If you come from a quality assurance (QA) or performance background, this early material gives you an idea of how a Java web application functions based on accepted principles of good design. (Experienced developers might want to just skim for new information.) This overview material does not contain the level of detail you need to design and develop a new Java web application. If you plan to write a Java web application, the bibliography includes some excellent books that provide the detail you need regarding good design and development practices.

Next, we discuss some performance tips that cover many common problem areas in Java web applications, such as logging and poor HTTP session management. Most of these tips apply to any Java web application server implementing the J2EE (Java 2 Platform, Enterprise Edition) standard. (Any tips specific to a given Java web application server implementation receive special note.) If you need more specifics on a particular area of the J2EE standard, we suggest you start by reviewing the materials available on the Sun Java web site: <java.sun.com>.

Remember: The web site is a system. The Java web application plays a critical role in the performance of your web site, but it also interacts with the Java Virtual Machine and other hardware and software components. We suggest that you read this chapter
and the next two as a unit to give you a good sense of how the web site components function together to deliver content. Also, let’s settle on some terminology at this point. Rather than repeating the phrase “Java web application server” throughout the rest of the book, we’re just going to call it an application server. Likewise with “Java web application”: from now on we’ll refer to it as a web application.

Web Content Types

Java web sites serve two types of content: static and dynamic. Before we get into the specifics of the application server, let’s discuss the differences between these two types of content, and what they mean to the performance of your site. Static content refers to elements of your web content that rarely change. Familiar static elements include things like graphic elements (gifs, jpegs, and the like), stable HTML elements like banners and pages, and client-side elements like JavaScript, style sheets, and so on. Usually static content lives in files at your web site. Depending on the type of web site you operate, static content might make up the largest proportion of information your site serves.

Dynamic content refers to web pages or portions of web pages created by a web application. A program actually generates the content returned. For example, if the user requested a search based on a keyword, the web site uses a web application to perform the search and build the list of results to return to the user. Dynamic content requires computing cycles on the application server, and may need to call other programs on other machines in your network, like a database server, search engine, existing program, or something else. As we will see, the mix of static versus dynamic content drives design decisions not only for your applications, but also for the network you set up to support them.

Web Application Basics

The Model-View-Controller Design Pattern

Before we plunge into the various elements of a web application, let’s first discuss a key concept in application design: the Model-View-Controller, or MVC, design pattern. (The J2EE world refers to this pattern as Model 2.) Regardless of what you call it, many web applications use MVC as a basis for their architecture and operation. Therefore, we want to describe briefly how this design pattern works. We’ll be using MVC terminology throughout this chapter to describe the roles of various web application components, so you need to be familiar with this subject before we move ahead.
Architects and developers like MVC because it allows them to divide the application into pieces that can be assigned to different developers with different skill sets. These pieces map cleanly to easily understood subcomponents of the application. Traditionally, the design pattern breaks down as follows:

- **Model**: The model encapsulates business logic and database schemas. Naturally, developers skilled in database technology and business logic develop the model portion of the application. In J2EE applications, model code is often implemented using JavaBeans or *Enterprise Java Beans (EJBs)*. The model remains independent of the application using it; therefore, multiple applications sometimes share the same underlying model.

- **View**: The view focuses on the look and feel of the application. Your developers with skills in human interface design and implementation usually work on the view. These components focus on presenting dynamic information to the user, but they do not gather this dynamic information themselves. In the J2EE model, *JavaServer Pages, or JSPs*, handle the responsibilities of the view.

- **Controller**: The controller handles the flow of the application. Application developers usually handle the components comprising the controller functions of your application. These components guide the flow of the view and instantiate model objects in order to fulfill the application’s requirements. In J2EE, the *servlet* and its associated classes play the role of the controller in the web application.

Because of the multi-tiered nature of web sites, MVC is an excellent choice for web application design. Using MVC yields applications that are flexible, easily maintainable, and extendable. MVC also makes it easy to divide the application among developers with different skill sets. In fact, web sites thrive under the MVC model. Many web sites change their look and feel constantly. By splitting out their presentation (view) components from their application (controller) code, they isolate the presentation of the web site from the logic flow. This allows the design team to make cosmetic changes to the web site without calling the programmers (who aren’t generally known for their artistic abilities anyway). Changing a page’s look not only doesn’t require a programmer if you use MVC, but the update cannot impact the program’s code. Likewise, if another application needs a column added to a database table, the resulting updates to the model do not impact either the controller or the view of other applications that use the same table.

Figure 2.1 shows how MVC works within a Java application server. (If some of the components are unfamiliar to you, we explain them later in this chapter and in the next.) The client, using a browser, makes a request for a URL that invokes a servlet. The network sends the request to the correct machine, and the HTTP server on that machine receives the request. The HTTP server passes it on to the plug-in, which determines that it is a request for the application server to handle. The application server takes the request and maps it to a servlet. The application server’s web container instantiates the servlet, if required, and begins executing it to satisfy the
request. During execution, the servlet calls model objects as needed to read or update data and forwards the resulting data to a JSP. The JSP formats the data into a web page and returns the generated page back to the client.

Many leaders in the field of Java web application design recommend the MVC approach for web site architecture. While the J2EE model does not enforce this design pattern, we expect (and hope) that your web developers used this approach to assemble your web application. Keep in mind that MVC is a design strategy. By itself, it does not guarantee terrific performance for your web application. However, knowing your web application’s overall design structure (even if you’re not a developer) often proves helpful for several reasons:

- Understanding the design of the application allows you to follow its flow more easily. This is frequently useful in tracking down bottlenecks.
- Knowing the breakdown of responsibilities for the various components of the application allows your team to involve the right skills for problem solving.
- Proper application structure often proves an effective indicator of overall application quality.

![Model-View-Controller diagram](image)

**Figure 2.1** Model-View-Controller within a Java web application. From “WebSphere Programming Model: Best Practices for Http Servlets and Java Server Pages,” paper presented by Ken Hygh at Solutions 2001, San Francisco, CA. © IBM Corp. 2001. Reprinted by permission of IBM Corp.

---

Servlets

A servlet is a small Java program. These little programs provide the intelligence behind Java web site applications. Each incoming HTTP request for a dynamic page goes to a servlet. The servlet decides how to satisfy the dynamic request by marshalling data from any number of sources and passing it to the presentation logic to generate the dynamic page. For any given request, the servlet might choose between multiple presentations to return to the user. For example, if the site supports both French and English pages, the servlet looks at the parameters on the incoming HTTP request and picks the correct format for the output. Or a servlet might choose an error page rather than the normally displayed page if the servlet couldn’t communicate with the database or a validation error occurred.

Servlet Scope

Large web sites contain lots of functions for their users. Each function provided maps to a different request sent to the web application providing the intelligence behind the web site. How the web site maps these requests to functionality varies. Some web sites use a new URL for each request, while others generate one or more URLs with a series of parameters attached to differentiate between functions. Regardless of the form your requests take, your web site needs a strategy for dealing with a variety of requests. Specifically, you need a strategy for controlling how you map an incoming request to the function you must provide and to the JSP you eventually use to return the dynamic page to the requestor.

One popular strategy maps the incoming requests into a small set of servlets. Each servlet handles a small set of requests and uses one or two specific JSP pages to return dynamic pages to the requestor. The servlet contains only the logic required to satisfy the small set of requests it handles. This approach also allows you to update and add new function to the web site easily, as the logic for the function resides in discrete servlets and JSPs. Updating or adding functions does not impact other servlets. Another strategy uses a “front controller” to manage incoming requests. The controller potentially handles all the requests for a functional area of your web site (say, the banking or brokerage areas of a finance web site). The controller uses “helper” objects, such as JavaBeans, to pull together data for the JSPs. (See Figure 2.2 for an example of the two common request management strategies.)

3. See Deepak Alur, John Crupi, Dan Malks, Core J2EE Patterns.
Regardless of the request management strategy you use, keep the following performance considerations in mind.

- **Keep code paths short**: Design your control logic to quickly recognize each request and begin meaningful processing to build a response. Some web applications traverse enormous “decision trees” (usually an enormous “if” construct) to determine how to answer the user’s request. The decision trees often prove inefficient, especially if the site handles a lot of different URL/parameter combinations.
Maintain the right granularity as your site grows: As your web site grows, you add web application logic to support new features. For example, if you operate a web banking site, you might grow your web site to include brokerage functions. To support the brokerage functions, you need new web application logic.

How you grow your site really depends on the request management strategy you use (front controllers with helper objects versus a larger group of self-contained servlets). In any case, we recommend building new helper objects or self-contained servlets as you add new function. (Also consider adding a new controller for major new functional areas of your web site.)

Again, the idea is to keep the code paths in your web site short. Often, when developers shove significant new function into existing servlets or helper objects, the code paths within these objects become unnecessarily long. This results in not only suboptimal performance but also in maintenance difficulties for the development team. For example, in our banking web site, we might add new servlets to support our new brokerage functions. (For web sites using the front controller strategy, we might add another front controller and set of helper objects specifically for the brokerage functions.) Likewise you potentially need new JSPs, JavaBeans, and other objects to support this new feature.

As with any design practice, you want to avoid extremes of object creation as well. We sometimes encounter developers who create a servlet for every possible request permutation the web site might receive. Obviously, thousands of little servlets prove just as difficult to maintain and extend as one gigantic servlet. When your site grows, consider refactoring the web application classes if you’re generating lots of classes containing similar logic. (Refactoring simply means reorganizing your application. It often involves centralizing common logic currently distributed throughout your objects.)

Servlet Operation

Now that we better understand what servlets do, exactly how do they work? The application server loads the servlet either as the web container starts or when the servlet receives its first request, depending on administrative settings. (Note: JSPs and servlets share many runtime characteristics, although their roles at the web site differ. We discuss the specifics of loading and running JSPs in more detail later in this chapter.) When the servlet loads into memory, the application server calls the servlet’s init() method. This gives the servlet an opportunity to obtain static resources and perform initialization steps. (As we’ll discuss later in this chapter, good management of static resources often dramatically improves servlet performance.)

---

The `init()` method only runs once, at servlet load time. The servlets we most commonly see inherit from `javax.servlet.http.HttpServlet`. They return HTML or other appropriate formats in response to an HTTP request. As the HTTP requests arrive for the servlet, they trigger execution of the servlet's `doGet()` or `doPost()` methods, depending on the request. The servlet programmer provides the logic inside these methods.

Servlets typically run as multi-threaded entities. That is, the application server creates only one instance of the servlet, and multiple threads may execute code in this instance simultaneously. For example, if your web site receives four simultaneous requests for the same servlet, those requests execute inside the `doGet()` or `doPost()` method on separate threads simultaneously. Because of this multi-threading, you cannot use instance variables inside your servlet to maintain state information.

The servlet specification does provide a single-threaded alternative. By implementing the `javax.servlet.http.SingleThreadModel` interface, your servlet becomes single-threaded. This means the application server creates an instance of your servlet for each simultaneous request. If you’ve never written a multi-threaded application before, you might feel tempted to use the single-threaded model. Don’t do it! The single-threaded model does not perform well, particularly in high-volume web sites nor does it scale well. In this model, each request operates within its own fully instantiated servlets. This requires more memory and related overhead than the multi-threaded model. While the single-threaded model might seem like a shortcut, it is actually a performance dead end. Take the time and learn how to write good multi-threaded servlets instead.

Obviously, threading is an important issue for servlet developers. We discuss threading in more detail in Chapter 4. Other excellent books provide detailed tutorials on managing threading issues inside your servlets. Take the time to learn and master these issues before writing your web application. You will reap returns in improved application performance, fewer application errors, and less rewriting.

**Servlets, Threads, and Queuing**

If the request specifies a page generated by a servlet or JSP, the HTTP server plug-in (see Chapter 3 for a more detailed discussion of HTTP servers) passes the request into the web container. The web container runs inside a J2EE application server running in a Java Virtual Machine (JVM) and has a limited number of threads available to handle servlet requests. Typically, the number of HTTP “listeners” (see Chapter 3) greatly exceeds the number of available servlet threads in the application.

---

server’s web container. (For example, a typical installation may have 150 HTTP server listeners, but only 20–50 servlet threads.) Figure 2.3 shows the different numbers of listeners and threads, including servlet threads, that make a call to a database through a connection pool (we’ll discuss connection pools later in this chapter). The application server assigns the incoming request to a thread, and the requested servlet runs on this thread. After the servlet satisfies the request and returns the data to the user, the thread returns to the available servlet thread pool inside the application server’s web container. If all of the threads in the web container are busy when a request arrives, the servlet request queues. Both queued requests and requests executing inside the servlet engine tie up the HTTP server listener thread originally assigned to them.

At runtime, you want a threading “funnel effect,” as demonstrated in Figure 2.3. You do not want equal numbers of worker threads at every layer in a multi-tiered application. At first, this seems counterintuitive, but keep in mind that the work occurring at one layer does not always involve the next layer. For example, if your HTTP server handles requests for both static and dynamic data, some of the HTTP listeners handle the static requests without involving the application server. Therefore, not all of your HTTP listeners engage the application server simultaneously. Likewise, not all the servlet threads make database calls simultaneously, so you need fewer database connections in the connection pool than configured servlet threads.

Application servers allow you to set the maximum threads allocated for executing servlets and JSPs. You may think the default setting for your application server thread pool is quite low (some vendors set the default as low as 15–20 threads), especially if you plan to handle a lot of user requests per second. Actually, the default settings, despite the small number of threads, probably represent the optimum number of threads for the average web application. This seems counterintuitive: If you want to handle more simultaneous requests through your web site, you should allocate more threads, right? Wrong. Actually, more threads often prove detrimental to web performance. As you allocate more threads, you create more work for the JVM, which manages the threads. Adding more threads generates more overhead as the JVM tries to give each thread its fair share of CPU and other resources. This is known as context switching overhead, and it increases as the number of threads increases, reducing the amount of useful work that the greater number of threads in the JVM can actually accomplish. So, start your performance testing by leaving the servlet/JSP thread pool sizes at their defaults. You might try adjusting these thread pool sizes up (and down) using small increments. If you see improvement, consider moving up a little bit more. However, keep in mind the key to servlet/JSP thread pools: Less is often more.

---

CHAPTER 2  ■  JAVA APPLICATION SERVER PERFORMANCE

The Frozen Web Site Danger

Before we talk about specific servlet performance issues, let's talk about how poor programming practices sometimes literally stall web sites. We call this phenomenon the “frozen web site.” The web site becomes nonresponsive to requests, even requests for static elements like gifs, jpegs, and static HTML pages.

How can failures in servlet programming impact the HTTP server? The answer lies in the queuing between the HTTP server plug-in and the application server engine. We discussed in the previous section how the HTTP server plug-in queues requests for the web application server until a servlet thread becomes available, as shown in Figure 2.3. However, poorly written servlets sometimes stall as a servlet waits indefinitely for a nonresponsive resource—perhaps a remote database or a remote system overloaded with traffic. In any case, the servlet programmer didn't provide an exit strategy for the servlet in this circumstance.

Figure 2.3  Threading/queuing "funnel." From IBM Software Group Services Performance Workshop presented by Stacy Joines, 2001, Hursley, U.K. © IBM Corp. 2001. Reprinted by permission of IBM Corp.

The Frozen Web Site Danger

Before we talk about specific servlet performance issues, let's talk about how poor programming practices sometimes literally stall web sites. We call this phenomenon the “frozen web site.” The web site becomes nonresponsive to requests, even requests for static elements like gifs, jpegs, and static HTML pages.

How can failures in servlet programming impact the HTTP server? The answer lies in the queuing between the HTTP server plug-in and the application server engine. We discussed in the previous section how the HTTP server plug-in queues requests for the web application server until a servlet thread becomes available, as shown in Figure 2.3. However, poorly written servlets sometimes stall as a servlet waits indefinitely for a nonresponsive resource—perhaps a remote database or a remote system overloaded with traffic. In any case, the servlet programmer didn't provide an exit strategy for the servlet in this circumstance.

Figure 2.3  Threading/queuing "funnel." From IBM Software Group Services Performance Workshop presented by Stacy Joines, 2001, Hursley, U.K. © IBM Corp. 2001. Reprinted by permission of IBM Corp.
If the servlet stalls, the thread running the servlet never returns to the application server's thread pool. This reduces the resources available to perform useful work. The nonresponsive remote system may stall every servlet thread in the web site. The stall happens quickly if the servlets call the remote system on almost every servlet request. If the web site uses the remote system infrequently, it may lose threads over the course of hours or days before the stall finally occurs. As new requests arrive they queue to wait for an available servlet thread, occupying the HTTP server threads as they wait. Eventually, the HTTP server runs out of threads to respond to simple static requests. At this point, the web site is “frozen,” as shown in Figure 2.4.

Frozen web sites exhibit some variety in symptoms. As we mentioned above, all of the available servlet/JSP threads might stall immediately if almost every request accesses the nonresponsive back-end system. In these cases, the web site appears to “hang” fairly quickly. That is, the site at first stops responding to dynamic requests, and shortly thereafter stops responding to static requests as well. However, other functions, which monitor the application server without using the HTTP interface, continue to function normally. (For example, depending on the application server vendor, the administrative interface may continue to work, although the application server no longer can serve pages.) If your web site only accesses the nonresponsive system infrequently, it may take hours or days to become nonresponsive. In these cases, the web site often experiences increasingly poor throughput and response time before it completely stops responding to requests.

The “acid test” for a frozen web site is a thread trace. A thread trace gives you details on the activity of the threads in a JVM. In the case of frozen web site, the thread traces shows the statement on which each thread is waiting. (See Chapter 12 for more details on thread traces.)

Avoid the frozen web site by planning for the eventuality of a back-end outage. Set time limits on any requests to a remote resource, and exit if the remote resource fails to respond during the time limit. If the remote resource doesn’t respond, cut your losses and return an error page if you cannot obtain the data you need in a reasonable time. Do not stall your site waiting for resources. Some web containers automatically grow their thread pools in these situations. However, the thread pool cannot grow infinitely. Either performance becomes unacceptable, or the JVM collapses because of the number of running threads. Even with self-adjusting thread pools, then, the web application must prepare for and handle nonresponsive remote systems.

Web sites need short timeout periods. Unlike thick clients, which serve one user, web sites may have dozens or even hundreds of users making requests every second. The servlet often cannot afford to wait 30 seconds for a database or other back-end resource to respond. Keep the timeout values short (a few seconds at most). Place them in
external resource files, such as a resource bundle, and load them at servlet initialization (when the servlet’s `init()` method executes). This allows the administrator to tune the timeout values later, if required, without modifying the code. A good web site keeps moving no matter what happens to remote resources. Plan for outages, and build an error path that allows your servlets to handle outages consistently. (Usually, the web site team sets up an error JSP to return to the user in these cases.) Again, it is far better to return an error page to a single user than to stall the web site and affect every user.

**Servlet Performance Tips**

As with most specifications, the servlet specification gives the programmer little guidance on the best way to write a servlet. We encounter many teams with strong backgrounds in thick client development struggling over the transition to server-based
applications. Some “best practices” from the thick client world make for disaster when applied to a servlet application. A naive code port from a client-server application to a web application typically performs poorly.

Web sites amplify small programming errors and inefficiencies. For example, in the previous section we discussed thread management and thread stalls. In a thick client environment, an overly generous timeout for a back-end resource might annoy the end user but wouldn’t create problems for the system as a whole. On a web site, an overly generous timeout might bring the entire site to its knees. In the client-server paradigm, one client’s operation doesn’t interfere with another’s, but the same is definitely not true in server-side programming for Java web sites.

In this section we discuss some common application performance problems. Some of these issues might not make a tremendous difference in a thick client application. However, because of the volume of requests a large web site executes each day, these minor points become critical to the performance of a web application. Let’s look at some of these best performance practices for your servlets.7

**Use Servlets to Control Application Flow**

Servlets control the flow of your application. Do not use the servlet to generate HTML or other output formats; use JSPs instead. This works even if you’re not returning HTML, as JSPs can build XML or other output datastreams. This also keeps your view logic separate from the control logic of the web site, which makes it easier to change the look and feel of your web site over time.

From a performance perspective, using the servlet strictly as the controller often helps the web application avoid garbage collection issues. The outbound presentation layer usually consists of lots of strings. The JSPs generated by your application server build these strings efficiently, and efficient string handling generally reduces your garbage collection cycles. (See Chapter 4 for more details on string management and garbage collection.)

**Acquire Static Resources at Initialization**

Earlier we discussed the multi-threaded nature of servlets and noted that you cannot store state information in instance variables shared by multiple threads. However, some information fits very well into instance variables. For example, programmers often repeat the Java Name and Directory Interface (JNDI) lookup for a common resource such as a data source unnecessarily. The JNDI context is threadsafe, so if

---

the programmer places it in a class variable, all the servlet threads may share it safely. We recommend performing any common lookups inside the servlet's init() method, and then storing the results in class variables. With this technique, the servlet only looks up the context for a given resource at initialization time. Given the performance overhead associated with JNDI lookups, this technique also significantly speeds up servlet processing.

In Listing 2.1, we use JNDI caching in a servlet example using the IBM WebSphere Application Server. This code gets the data source information once in the init() method and stores it in a class variable called myDataSource. Subsequently, every request that calls the doGet() method obtains a database connection from the data source defined in myDataSource. This spares each request from doing its own JNDI lookup of the data source. The IBM WebSphere Application Server (Version 3.5.2 and above) provides a JNDI lookup cache to speed up applications. However, doing the caching inside the servlet is even faster. (Check with your application server provider for more details about any similar JNDI caching strategies.)

**Listing 2.1** Example of caching JNDI lookups inside a servlet

```java
public class MyServlet extends HttpServlet {
    private static final String dsName = "myDataSource";
    private static Context context = null;
    private static DataSource myDataSource = null;

    public void init(ServletConfig config) {
        Hashtable parms = new Hashtable(2);
        parms.put(Context.INITIAL_CONTEXT_FACTORY,
                   "com.ibm.websphere.naming.WsnInitialContextFactory");
        parms.put(Context.PROVIDER_URL, "iiop:///"); // local machine
        context = new InitialContext(parms);
        myDataSource = (DataSource)context.lookup("jdbc/" + dsName);
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response) {
        acctNumber = request.getParameter("account");
        if (acctNumber == null || ".equals(acctNumber)){
            // do error page
        }
        AccountInfo bean = new AccountInfo(acctNumber, myDataSource);
        // the AccountInfo bean will use the DataSource in order to get
        // its data from the database, by getting a connection, etc.
        //
        request.setAttribute("acctInfo", bean);
        getServletContext().getRequestDispatcher("foo.jsp").forward(
```
request, response);
} // doGet
} // class MyServlet

Manage Servlet Logging

The servlet standard lacks a detailed logging strategy at this point (see the discussion below on using the `servlet.log()` method). Until a standard exists, you might consider several of the free logging frameworks currently available, or you might implement your own custom framework. Regardless of your approach, your application needs a logging strategy. From a performance perspective, logs usually end up at the hard disk at some point. Of course, anything interacting frequently with the hard disk presents a potentially performance expensive operation. Before you choose a logging framework, or write one yourself, consider the following suggestions regarding logging.

- **Support Logging Levels and Use Them** Sometimes you require more information than at others. Allow the application programmers to give a log message a weighted “importance” value. For example, severe application errors always rate a “Level 1” importance, while trace information always rates a “Level 4.” Use code reviews to keep the team in synch as to the importance assigned to various classes of messages.

- **Reduce Production Logging** When the application runs in production, keep logging to a minimum. At this stage, you only want to capture application errors, not trace information (unless you’re trying to find a problem in production, which should be rare!). This greatly reduces the time the system spends interacting with the hard disk. (Remember, hard disk interaction is generally very performance expensive.) It also reduces the size of log files on the system, and avoids potential disk space issues.

- **Make Logging Levels Runtime Configurable** If your system encounters an error, you probably want to record more detailed log information for awhile to capture the trace information around the error. However, turning the logging on for the course of the day might be too expensive in terms of performance. Instead, consider building a special “administration page” as part of your web application to toggle logging levels on the fly. (Actually, custom administration pages often prove useful for other things, such as setting various application parameters. Web site administrators use them to set everything from timeouts to the location of custom log files.)

- **Consider Buffered Writing** Some logging mechanisms prevent disk overflows by setting an absolute maximum file size for the log file. After reaching this size, the log begins writing again at the top of the file. The danger, of course, lies in losing information in the history you might need. However, it does defeat any danger of stalling the system by running out of logging disk space.
- **Take Care with Memory Buffers** Some logging systems write the log to a memory location, and dump to hard disk only on request, or only after the memory buffer fills up. While this is certainly faster than a pure disk I/O system, avoid overcoming your JVM heap with log data. Set a hard maximum for the buffer in memory, and make sure your applications still have enough room to run.

- **Avoid `servlet.log()`** The standard provides a logging interface via the `servlet.log()` method. However, the implementation of this method varies greatly among application servers. Some servers just write the information to `stdout` while others write the information to their administrative database logs. The latter makes for a very, very expensive logging technique, and greatly impacts performance. We recommend avoiding the `servlet.log()` method, particularly if you plan to port your application to multiple application server vendors’ platforms.

- **Avoid `system.out.println()`** Many programmers place these statements in their application code for debugging or logging, and leave them when the code goes into production. This is a serialized resource, and requires file I/O on every call, which impacts performance. If you can’t easily get these out of the code, consider setting your application server’s standard out to `/dev/null` (depending on operating system) so these statements do not trigger file I/O.

- **Use an Available Logging Framework Rather Than Writing Your Own** Rather than reinventing a logging framework, consider using one of the many available (for example, Apache’s Log4j or JTrack’s jLog). We do not recommend writing your own logger, as this tends to introduce errors and performance problems into many projects. Rely on existing technology if at all possible.

### File-Serving Servlets

Many application servers include a servlet that acts as a miniature HTTP server. This servlet is a convenience feature, allowing the application server to serve static content. This permits the web site programmer to test web applications without defining the static content of the application to an external HTTP server. However, the file-serving servlet lacks performance efficiency. Normally, a request for static content stops at the HTTP server, which handles the request. However, if the file-serving servlet handles your static content, the request must travel through the HTTP server, the application server plug-in, and the application server itself before reaching the file-serving servlet. The increase in path length increases the overhead required to return the servlet.

Whenever possible, we strongly recommend using an HTTP server or similar server (such as a caching proxy server) to deliver static content to your users. However, if you plan on using your application server to provide security for static as well as
dynamic content, you may require the application server's web container and the file-serving servlet to properly secure your static content. Of course, the J2EE 1.2 standard complicates this advice through the use of Enterprise Archive (EAR) and Web Archive (WAR) files to consolidate all the components of the application (including static elements). With this approach, the administrator deploys the web application into the application server, which serves both the static and dynamic content. While this approach might work for small web sites, or sites wanting to secure their static content, we still recommend using an HTTP server to serve your static content. Making this work with your application and packaging tools may require some effort, but is well worth the resulting performance gains.

JavaServer Pages (JSPs)

The previous sections discussed JavaServer Pages (JSPs) a great deal without really defining them or what they do. Let's briefly discuss the operation of a JSP, and its role in a Java web application. Early in the development of the servlet standard, the need for a mechanism to separate the logic layers of the web site from the presentation layers became apparent. The JSP developed out of this, and provides the presentation layer with sufficient abstraction to handle dynamic data output while keeping the feel of an HTML page. (Using MVC terminology, JSPs function as the view.)

A JSP exists to return dynamic data in a predefined format (often HTML). The JSP receives the data to display from a servlet and places the data into the format using special JSP tags defined by the specification. You may extend the function of the JSP by writing your own tags and creating your own “tag library,” which is strongly recommended. This allows you to isolate Java code out from the body of your JSPs.

The JSP standard tries to keep the JSP page familiar to a less technical audience: the graphic designers and layout specialists for your site. These folks come with tremendous artistic talent but not a lot of Java programming skills. Good JSPs isolate the design folks from the programming team working on the servlets. As we mentioned earlier in our discussion of servlets, avoid putting presentation logic inside servlet code. Your layout team, the folks high in artistic talent but low in programming skill, cannot modify presentation logic managed inside a servlet. They do not know how to write or modify the servlet’s Java code. Don’t put the look and feel of your web site in the not-so-artistic hands of your servlet developers; use JSPs to build your presentation layer.

Basic JSP Runtime Operation

So how does the JSP work? Operationally, a JSP is just a special form of a servlet. The first time a request requires a JSP, the web application server runs the JSP page
through a parser to turn it into a servlet. The web application server then compiles this intermediate “servlet state” of the JSP and launches it on a thread just like any other servlet. At this point, the web application server makes no differentiation between the JSP and any other runnable servlet. (Figure 2.5 illustrates the differences between servlet loading versus JSP loading.)

Usually the application server checks the file system periodically for a more recent copy of the JSP. If it finds one, it parses, compiles, and starts the newer version. The interval at which the application server checks for new JSPs is usually configurable via an administration setting.

**JSP Performance Tips**

Web applications servers execute JSPs much as they do regular servlets; thus, our previous discussions on good servlet programming techniques also apply to JSPs. Again we want to emphasize that you should not place Java code inside your JSPs. If you need to embed Java logic to control the layout of the page, use a tag library or JavaBean (see below) to hide the code from your presentation team. While JSPs
behave at runtime like any other servlet, they do involve a few unique performance considerations. Let’s review the most important performance tips for your JSPs.

**Reload Intervals**

Application servers check for updates to the JSP and reparse and recompile the JSP if a more recent version exists. Of course, these checks require file I/O to look at timestamps on the JSP files. Again, anything interacting frequently with the hard disk is performance expensive. If your application server permits, avoid auto-reloading JSPs in production (this is also a good practice for site security as well). If you cannot turn the auto-reload function off completely, set the reload interval as high as possible.

**HTTP Session Created by Default**

The J2EE specification supports creating “implicit” objects inside JSPs without requiring a formal declaration by the programmer. To support this feature, some application servers automatically create an HTTP session object for each JSP page if it does not already exist. As you will see, a session object remains in memory until it times out and is removed by the container. The default timeout is often as high as 30 minutes. Obviously, these session objects might take up a lot of heap space in the JVM without the web team even being aware of their existence!

Find out how your web application server manages this portion of the J2EE specification. If you do not use implicit objects and do not want an HTTP session object created in your JSP page, use a tag to prevent the web container from creating one automatically. The tag for turning off the HTTP session default creation is

```html
<%@ page session="false"%>
```

**Precompile or Preclick**

The first time the web site receives a request for a JSP, the web application server parses, compiles, and starts the JSP as a running servlet. The parsing and compiling steps often require several seconds to perform and use significant amounts of CPU. If at all possible, get these parsing and compiling steps out of the way before the web site comes under load.

Some application servers provide JSP precompilers to eliminate the compile overhead at runtime. Make use of this feature if your web application server supports it. (The IBM WebSphere Application Server provides a batch compiler for JSP files.)

If your application server doesn’t provide a JSP precompiler, consider developing a process to request as many JSPs as possible when your web application comes up. You might use an inexpensive load test tool for this, or it might just be a manual process performed quickly by the administrator after restarting the machine. Obviously,
for large web sites, touching every JSP on restart isn’t always feasible. However, the sooner the most frequently used JSPs become compiled, the sooner your web site gets back to normal response time and CPU utilization. (Also, precompiling your new or updated JSPs alerts you to any parsing or compilation errors before your user community encounters them.)

**Using JavaBeans and Custom Tags**

“Regular” JavaBeans make a nice complement to servlet/JSP programming by implementing the Model portion of MVC. Beans frequently play the role of a “contract” between a servlet and a JSP. The Bean defines the data the JSP might expect from the servlet, as well as methods available to access this data. Regular Beans also act as high-level wrappers for Enterprise JavaBeans (EJBs) by defining the data interface without revealing the EJB interactions underneath used to access the data or function.

JSPs use custom tags to camouflage their interactions with JavaBeans. The custom tag looks like just another HTML tag, only with special attributes defined by your development team. Behind the tag, your developers provide logic to interact with a JavaBean to retrieve dynamic data to populate the tag. Custom tags hide the Java implementation details from your design team and eliminate visible Java code in your JSPs.

Servlets pass JavaBeans to JSPs via the HTTP session or the HttpServletRequest object. Again, the Bean contains the dynamic data the JSP will turn into an output page for the requesting user. Your design and programming teams may coordinate JavaBeans and custom tags manually, or they may use some of the tools available to perform “drag-and-drop” creation of JSPs with custom tags and JavaBeans.

**JavaBean and Custom-Tag Performance Tips**

Just as with any other feature, misusing JavaBeans and custom tags negatively impacts your web application’s performance. Keep the following best practices in mind when using JavaBeans throughout your application and custom tags in your JSPs:

- **Avoid bean.instantiate()** JavaBeans come with their own simplified “persistence” mechanism. Beans may place their contents in a file for “reconstitution” at a later time. For this reason, if your code creates a new JavaBean using the bean.instantiate() method, the servlet or JSP code checks the hard disk for any persistent Bean contents to load. Checking the hard disk takes a relatively long time, and greatly impacts the performance of your web application, so avoid using this method when creating Beans. Instead, use the MyBean aBean = new MyBean() technique to create a new JavaBean instance in your web application code.
Use a Database If You Need to Persist Bean Data  If you must persist the contents of your JavaBean, use a database, not the file system. Databases do a much better job of handling simultaneous updates than does a native file system. Again, think of the impact if thousands of servlets every hour tried to dump Bean contents directly to the hard drive. The native file system does not perform well enough to support this kind of burden, and web site performance would tumble. Instead, use a database or EJB (which interacts with the database) to store Bean contents persistently.

Use Custom Tags in Your JSPs  We discussed this earlier in the section, but it bears another brief mention here. Early JSPs depended heavily on embedded Java code to retrieve dynamic data from JavaBeans and to place the dynamic data into the page returned to the user. This usually involved liberal use of the scriptlet tag inside the JSP. It also required involvement by your development team to build this embedded Java and to maintain it as the look and feel of the web site changed over time.

The custom tag feature of the J2EE standard allows your developers to build a library of custom HTML tags to hide the Java required to retrieve dynamic data. These tags allow your designers to work with a familiar interface to place dynamic data in the JSPs’ layout.

While custom tags provide no direct benefit to performance (we can’t say a custom tag runs faster than a lump of code in a scriptlet tag), they do simplify the web application and make it more maintainable by the folks with the correct skill sets. Simplification and organization often go a long way in making your code run faster. Likewise, using custom tags might prevent your designers from making ad hoc updates to the Java code used to retrieve dynamic elements. Scriptlet tags make it much easier for folks outside the development team to add underperforming code to your web site.

JSPs and XML/XSL

Previous sections discussed JSPs returning HTML pages. Recently, however, many web sites are using XML (Extensible Markup Language) and XSL (Extensible Style Language) to generate dynamic pages for the user requests. For example, the JSPs might generate XML instead of full HTML pages. The XML contains only the dynamic data, while the XSL stylesheet defines how to translate this data into a format usable by the requesting client.

The XML/XSL concepts often proves beneficial for web sites supporting several user devices. With XSL, your web site might use one stylesheet to generate WML (Wireless Markup Language) for wireless devices such as mobile phones or personal digital assistants (PDAs), while another stylesheet provides the format for traditional
HTML pages. While XML/XSL allows you to readily support multiple device types, this flexibility comes at a cost. The XML/XSL approach requires repeated parsing of the XML to pull together the final output. Repeated string parsing in almost any language quickly becomes a performance drain. Java often incurs an even higher overhead because of the extra garbage collections sometimes required to clean up after string processing. (See Chapter 4 for more details on string management in Java.)

Some client devices and browsers support XSLT (XSL transformation). If all your clients support XSLT, consider offloading the page assembly to the client’s machine. This pushes the parsing of the XML to the client machine, distributes the parsing penalty, and reduces the processing burden at your web site. However, many devices and browsers do not support XSLT. In these cases, you must parse the XML and build the output at the server.

**Assorted Application Server Tuning Tips**

Before we move ahead with our performance discussions, let’s cover a few remaining performance topics. These tips mostly concern some performance settings available to your web application objects. Consult your documentation on how to manipulate these parameters for your application server.

**Load at Startup**

You choose when a servlet starts. If you so specify, the web container starts the servlet when the web container itself starts. Otherwise, the web container only starts the servlet if it receives a request for the servlet. Starting the servlet at web container startup often proves beneficial if the servlet’s `init()` method performs significant initialization. This also proves beneficial if the servlet’s `init()` method performs initialization that affects other areas of the web application. However, it comes at the expense of slowing the overall application server start-up time. Check your application server documentation on how to set the “load at startup” parameter for your servlets.

**Web Container Thread Pool Size**

The number of threads available in the web container to execute servlets, as we have noted, affects the throughput and performance of your web site. As we discussed earlier, less is often more when allocating threads because of the context switching overhead. Adjust the thread pool size for your web container in small increments to find the optimum size for your particular application and operating environment. Finding the optimal setting is an iterative process best addressed during your performance testing.
Servlet Reload

The J2EE specification requires the web container to detect updated servlets, and to start executing the new version automatically. Often, development teams use this feature during the development phase to quickly produce and test servlet updates. However, the automatic reload function is rarely desirable in production. Some points to consider include the following:

- **Performance:** Enabling this function causes the web container to check the timestamp on the Java .class file that contains the servlet, and compare it against the last known timestamp. This check either happens at regular intervals (see below), or every time the servlet receives a request. Either way, a file check takes processing time away from your application.

- **Seamless Operation:** Most application servers implement servlet reloading through the use of Java’s custom class loaders. Some application servers will destroy all HTTP session objects when reloading any servlet to prevent mismatches in class definitions. Obviously, dropping all of the HTTP session objects on a production web site leads to high customer dissatisfaction.

- **Security:** Automatic reload allows the replacement of running code without stopping the application server. Whether innocently or maliciously, this gives people in your organization the opportunity to easily move unauthorized code into the production web environment.

  Turn off the automatic reload in production, if at all possible. However, if you choose to use automatic reload, consider increasing the reload interval. Some application servers set the interval quite low (as low as a few seconds in some cases). Increase the reload interval to several minutes, if possible, to minimize the application server’s interaction with the file system, and improve performance.

  **Note:** Your application server may differentiate between servlet and JSP reload intervals and provide individual settings for servlet and JSP reload settings.

**Beyond the Basics**

In addition to the basic features we’ve discussed already, the J2EE standard provides additional features for the web application. In this section, we cover the best practices for using these features.

**HTTP Sessions**

We discussed earlier the multi-threaded servlet model, and some of the implications this model has for your servlets. To recap, the connectionless nature of HTTP and
servlet multi-threading means the servlet cannot maintain state information for the user through normal programming techniques. For example, the servlet cannot store the user's name or account ID in an instance variable because each user does not obtain a unique instance of the servlet. Rather, multiple threads engage the same instance simultaneously.

Because the usual technique for keeping state information (instance variables) doesn't work with servlets, the servlet specification provides a different mechanism for maintaining this information. This mechanism is the `javax.servlet.http.HttpSession` class. A user's web site visit usually spans multiple, discrete requests, and sometimes lasts for an extended period of time (up to a day, in some cases). The `HttpSession` class allows the web application to keep a user's state information for the duration of the user's visit. The web application, through a servlet or JSP, requests the creation of an HTTP session for a visiting user. While the servlets and JSPs treat the HTTP session as a hash table, the application server manages the session and associates its contents repeatedly with the same user.

Application servers employ several techniques to maintain the association between users and their HTTP session data. In one commonly used approach, the application server gives the visiting user a “session ID”, and returns it in a cookie to the user's browser. On subsequent visits, the browser passes the cookie back to the web site, and the application server pulls the session ID from the cookie to find the user's HTTP session data. (Also, using this technique, the cookie and HTTP session management remains invisible to the programmer. The servlet programmer merely requests the current HTTP session object without directly manipulating the cookie.) See Figure 2.6 for an example of HTTP session management using cookies. Some specialized web sites also support users who cannot or will not accept cookies. In these situations, the web sites use URL-rewriting or embed state information as hidden form fields within the returned HTML. Unlike cookie-based HTTP session support, these two methods are not transparent to the developer. Developers must add code to their servlets/JSPs in order to support URL-encoding or embed hidden state information in outbound data.

While HTTP sessions solve a significant problem for your web application (how to keep state information about a user over the course of her visit), they introduce new considerations for your web site. HTTP session misuse often leads to web site performance and scalability issues.

**HTTP Session Memory Usage**

Most web application servers keep the HTTP sessions in memory. This allows the web application fast access to the user's state information. However, this also means
that the HTTP sessions share memory with the running web site applications inside
the JVM heap. The J2EE specification provides no mechanism for controlling the
size of an individual HTTP session. Some application servers, such as the IBM Web-
Sphere Application Server, allow the administrator to limit the number of HTTP
sessions held in memory, but they do not limit the memory allocated for each of
these HTTP sessions.

We discussed previously how small indiscretions that might go unnoticed in a thick
client application amplify to become serious issues on a high-volume web site. The
HTTP session size often falls into this category. For example, thick client program-
mers routinely trade memory for performance. The developer might cache signifi-
cant chunks of data in the client’s memory to avoid repeated trips to a remote data
source for retrieval. Because thick clients generally contain lots of dedicated memory
(125MB or more, routinely), this approach makes sense.
However, this same paradigm does not work in server-side programming. We rou-
tinely encounter web applications attempting to store 10MB of data or more into
each HTTP session object. 10MB of application data on a thick client presents no
real problem, but 10MB spells disaster on a high-volume web site. Let's look at the
math. If the HTTP session contains 10MB of data, and we have 10,000 users arriving
on the site during the HTTP session timeout period, these users would require the
following memory to hold their HTTP session data:

\[ 10,000 \text{ users} \times 10\text{MB/user} = 100\text{GB} \]

Since our optimum JVM heapsize often lies between 256MB and 512MB, one JVM
cannot support this web site alone. In fact, if we assume a 512MB maximum heap size
for each JVM, we need almost 200 JVMs to support this application!

Also, keep in mind that the objects you store in your HTTP session often contain
other objects. Your actual HTTP session size depends on every object you store,
including deeply nested objects. Make sure you understand the full extent of every-
thing you place in the HTTP session. HTTP session bloat clearly demonstrates why
you need a web site performance test before you enter production. This problem fre-
quently goes undetected in both the programming phase as well as the functional test
phase of your application's development. HTTP session bloat impacts your web site
only under load, because it deprives the web site of memory resource as the user bur-
den increases. Therefore, you need to introduce load from many unique users to
your web site before you enter production to flush out problems of this nature.

**HTTP Session Memory-Management Tips**

So how do you defeat the engorged HTTP session problem? A few techniques are
explained below.

*Preserve Data That Must Persist*  If you are using the class HttpSession to store
things that will later go into a database, consider putting them into a database in the
first place. For example, some programmers put all of a user's shopping cart data into
HTTP session and then store this information into a database as well, “just in case.”
For these situations, consider just keeping keys in HTTP session and looking up the
rest of the state information in the database if and when needed. (This technique is
epecially handy if the state information is referenced infrequently.)

*Perform Code Reviews*  Review the web application code for excessive HTTP session
storage. Also, go over this problem with your web site team, especially programmers
new to the web application space. Ideally, use the HTTP session to hold a few thou-
sand bytes of data (under 2KB is ideal for most applications). If data is rarely used,
and inexpensive to look up, then just leave it in the database and load it as needed.
Use Code-Proﬁling Tools Many tools are available for proﬁling Java applications. Consider using these tools to get a handle on how much data the HTTP session uses. (See Appendix C for a list of popular code-proﬁling tools available for Java.)

Serialize the HTTP Session to a Stream Many web sites use a specialized servlet to write a user’s HTTP session data to a stream and then measure the size of the stream to determine the size of the HTTP session. This allows you to check the HTTP session size easily in a production environment without introducing specialized tools into the environment. It also allows you to check the contents of the HTTP session to be sure everything it holds implements the serializable interface. (If an HTTP session contains non-serializable elements, this restricts your ability to share it in a clustered environment.)

In addition to reducing the data kept in the average HTTP session, we also need to monitor how many HTTP sessions your web application keeps in memory at any given time. Optimizing this HTTP session cache reduces the total memory required by your HTTP sessions. Let’s discuss a few techniques for managing the HTTP session cache.

Reduce HTTP Session Timeout Most application servers allow the administrator to reduce the HTTP session timeout interval, which speciﬁes how long the HTTP session exists after the user’s last interaction with the web site. This interval gives the user an opportunity to read pages returned from the web site without losing their state information.

The interval also allows the web application server to recognize stale HTTP sessions. If the user does not return to the web site before the interval expires, the web application considers the user inactive and purges his corresponding HTTP session. Of course, the longer the timeout interval, the more unused HTTP sessions your site contains at any given point.

Usually the application server sets the timeout interval to a reasonably generous period of time. (The IBM WebSphere Application Server sets HTTP session timeout to 30 minutes by default.) Reducing the timeout might help the HTTP session pressure in your JVM, but avoid setting the timeout too low. You’ll anger users if you drop their HTTP sessions while they’re still using your web site. Also, setting the timeout very low sometimes negatively impacts performance. An extremely low timeout period may cause the application server’s HTTP session management routines to run frequently. Often these routines incur a signiﬁcant overhead, thus diminishing any beneﬁt of frequently cleaning the HTTP session cache.

Before you change the timeout interval, however, check with the web application’s development team. The J2EE speciﬁcation also provides a method on the HttpSession class for modifying its timeout interval. The timeout set via this
method overrides any value set administratively, so be aware of any timeout intervals set by the programming staff, and, of course, encourage them to externalize these interval values for tuning purposes.

Support a Logout Function  Consider supporting a logout function for your web site as either a button or menu item on your web pages. This lets users indicate when they’ve completed their visit, and allows your web application to immediately invalidate the user’s corresponding HTTP session. (The web application invalidates the HTTP session via a method. Invalidation destroys the HTTP session right away rather than waiting for the timeout interval to elapse.) One caution regarding HTTP session invalidation: Earlier versions of the servlet standard allow web applications to share HTTP sessions. Don’t remove HTTP session objects potentially in use by other web applications, if your application server supports the earlier spec level and your application uses this capability.

While we recommend a logout feature, we don’t really expect it to solve HTTP session management for your web site. Sadly, most users never touch the logout buttons on web sites they visit. Most users just move to the next web site without formally logging out of yours. So, while it provides some benefits, the logout function is only a part of an overall HTTP session management strategy.

If your web application handles sensitive data (such as financial information) and may be accessed from a shared workstation or kiosk, consider a logout function to be a requirement. Logging out prevents subsequent users from obtaining a previous visitor’s information via an existing HTTP session. Naturally, these applications generally support a very short HTTP session timeout interval and may force a logout after completing certain tasks.

Keep the HTTP Session Working Set in Memory  The working set of HTTP sessions belongs to the users currently engaged in using our web site. We want to keep these HTTP sessions available for fast access as the active users make requests. These HTTP sessions, therefore, need to remain in memory.

In the next section, we discuss in detail some techniques for sharing HTTP sessions and keeping them in persistent storage. At this point, however, let’s discuss the most common improper use of a persistent HTTP session store. Sometimes web sites desperate for memory reduce their HTTP session cache size and off-load most of their HTTP session data to persistent storage. This works like an operating system paging scheme: As the web application needs an HTTP session, the application server retrieves it from the persistent store. After use, the application server stores the updates to the HTTP session back to the database.

While this sounds like a good idea at first, in practice it does not perform well. Accessing remote HTTP sessions takes time and increases network traffic. Particularly for web sites serving high request volumes or supporting large HTTP sessions,
the remote HTTP session “retrieve on demand” approach quickly degenerates into abysmal performance. (Of course, these web sites are usually the most eager to try this technique.)

Always keep your working set of HTTP sessions in the memory cache. Use any persistence or sharing mechanism only to store/share HTTP sessions as an outage precaution. Do not try to use these mechanisms as an extension of the in-memory HTTP session cache. You may need to configure multiple application server instances (see Multiple Instances: Clones later in this chapter) to get enough total heap size to hold all the HTTP session data.

**Sharing HTTP Sessions**

Many application servers support sharing HTTP session data among many instances of the application server functioning in a cluster (see Clones below in this chapter, and Web Site Topologies in Chapter 3). Strategies for sharing session data vary. Sharing HTTP session data allows the cluster to failover a visiting user. If the web application server handling the user’s requests fails, the web site may route the user’s request to another application server in the cluster. If the user’s HTTP session data resides in a common store, this application server pulls in the user’s state data and continues the user’s visit to the site without interruption. Figure 2.7 demonstrates HTTP session failover using database persistence.

Some vendors share the sessions through a shared network update, either to all other servers in the cluster or from a “primary” server to a “secondary” server. In this strategy, the session data resides in the memory of one or more application servers in the cluster. Another technique places all the HTTP sessions in the cluster in a shared, persistent datastore (usually a relational database). Yet another sharing method involves using flat files on a shared file system to store the session data.

Each strategy generates its own performance issues. Clustered network updates potentially generate lots of network traffic and spread the memory burden of the HTTP sessions throughout the cluster. Likewise, persistent session storage requires transfer of data between the application server instances and the datastore. Serializing HTTP session data and writing it to disk is also slow.

**HTTP Session Sharing Tips**

If you share HTTP session data within your application server cluster, keep the following performance pointers in mind.

*Keep HTTP Sessions Small* Moving HTTP session data across the network increases the need for small HTTP sessions. Small HTTP sessions reduce the database burden for persistent session storage and keep the clusterwide memory
requirements manageable for a networked HTTP session sharing system. Small HTTP sessions also use less network bandwidth during their transfers.

**Keep HTTP Session Data Serializable** Objects placed in the HTTP session should implement the `java.io.Serializable` interface. (This includes the objects themselves, as well as anything they include or from which they inherit.) The application server serializes the HTTP session data onto the network when transferring the data. If the data does not implement the `Serializable` interface, the web application server throws an exception when it tries to transfer it to a shared database or other store.
These serialization exceptions often take the web site team by surprise. HTTP session sharing, in theory, requires no programming changes to implement. (The application server administrator determines whether to enable this feature.) However, if the application developers did not prepare their code for HTTP session sharing, the first attempt by the administrator to use this feature often fails.

While enabling HTTP session sharing may seem as easy as “flipping a switch” in load testing or production, objects placed in the HTTP session actually determine whether HTTP session sharing works. Check the content of your HTTP sessions during code reviews to avoid encountering this problem in testing or production.

**Avoid Nontransferable Data** Sometimes HTTP session data does not transfer in a meaningful way to another web application server instance. For example, if your application programmers stuff the HTTP session with things like thread handles, this data provides no value to another web application server instance on failover. (Remember, each server instance lives in its own Java Virtual Machine.)

Mark nontransferable variables in your HTTP session objects with the `transient` keyword, so they won’t be serialized. Of course, you must write your code to handle failover by recreating the transient data within the context of the new web application server instance, as needed.

This leads to a programming issue for your web application. Avoid using information specific to a machine or web application server instance whenever possible. In addition to thread handles and the like, avoid depending on a machine’s IP address and similar data that make it difficult for your application to failover. Again, rigorous code reviews and programmer education provide your best protection for these scenarios.

**Enterprise JavaBeans (EJBs)**

Enterprise JavaBeans (EJBs) allow your web site applications and other traditional applications to gain access to centralized business logic contained in distributed components. They provide a layer of abstraction between your applications and your database. In fact, your EJBs may consist of data assembled from various datastores. This section covers the EJB basics so that we can discuss some of their performance characteristics. If you need more information about EJBs, we list several excellent books in the Bibliography.

EJBs exist inside an *EJB container*, which controls the life cycle of the Beans under its care. This management includes instantiating and destroying EJBs, managing EJB pools, and even interacting directly with databases on behalf of certain types of EJBs.
EJBs come in two basic flavors: Session Beans and Entity Beans. Session Beans live a transient existence inside the container. They contain business logic and may contain state for a specific client. Session Beans are either Stateful or Stateless.

- **Stateless Session Beans**: As their name indicates, Stateless Session Beans do not maintain state from one method call to the next. After executing a method, they return to their pool to await the next request, which might originate from any client.
- **Stateful Session Beans**: Again, as their name implies, Stateful Session Beans maintain state information. A client application or web application may reference the same instance of a Stateful Session Bean for multiple calls. Stateful Session Beans, while not as transient as the Stateless Session Beans, do not persist. If the Bean times out, or if the container experiences a failure, the Bean and its state go away. (Note: Some application servers actually support failover for Stateful Session Beans, so check your vendor’s documentation.)

Entity Beans represent a persistent entity. The data contained by an Entity Bean generally originates from a single database row, although it could consist of an assemblage of data. Likewise, EJBs most commonly store persistent data in a database, although other persistence mechanisms also exist. The J2EE specification defines two types of Entity Beans: Container-managed Persistence (CMP), or Bean-managed Persistence (BMP).

- **Container-managed Entity Beans**: These Entity Beans require little or no coding to control their data retrieval and persistence. For example, when using a relational database as the persistent store, the CMP Bean requires little or no custom SQL to retrieve or store its contents from/to the database. The container manages the interaction with the persistent storage mechanism, and updates the contents of the EJB from the persistent store as required.
- **Bean-managed Entity Beans**, on the other hand, require custom-coded persistence interaction logic. These Beans contain their own logic for retrieval and storage with regard to their persistent storage mechanism. In the case of relational database interaction, these Beans require custom SQL provided by application developers to interact with the relational database.

**EJB Performance Tips**

EJBs give you excellent support for sharing your business logic between your web applications and thick client applications. EJBs also perform and scale very well in web applications if you plan ahead for performance and scalability. Let’s discuss some of the best practices to ensure good EJB performance.
Use Stateless Session Beans as Façades  Servlet or JavaBean programmers often interact directly with Entity Beans. On a typical user request, the servlet makes several calls to Entity Beans to gather the data required to satisfy the request. This, however, is not the best strategy for high performance. Direct interaction with Entity Beans from a servlet usually results in several remote method calls. These calls tend to be performance expensive, so reducing the number of remote calls in your code path makes sense.

Direct interaction also stresses the transactional boundary between the servlet and the EJB. Entity Beans synchronize their state after each transaction, often needing two database calls for each method called. When a servlet accesses the Bean directly, each “getter” method becomes a transaction. This usually means a database call to get the current data from the database and then another to update the database at the end of the method, as shown in Figure 2.8.

To circumvent these problems, use a Session Bean as a façade to Entity Beans. The Session Bean contains the logic to gather data from one or more Entity Beans and return this information to the calling application, which yields several advantages. The programming interface becomes much simpler: The client calls only one method on a single EJB to perform complex interactions. This reduces the remote method calls for these complex interactions (the Session Bean doesn’t require a...
remote call to Entity Beans in the same EJB container as itself). Likewise, if the Session Bean shares an EJB container with the Entity Beans, it also controls the transactional boundary for the entire interaction. The Session Bean controls the transaction for all other EJBs involved in implementing the function. The Entity Bean only synchronizes its state when the Session Bean reaches a transactional boundary (such as the completion of the Stateless Session Bean’s method). Figure 2.9 shows an example of transactional boundaries using façade Beans.

Also, façade Beans prove useful when coordinating transactions across multiple Entity Beans, as well as other Session Beans performing direct JDBC calls. This requires the Entity Beans involved to implement the correct transaction setting, usually TX_SUPPORTED or TX_REQUIRED.

Use Direct JDBC Calls If you manipulate multiple rows of your database when retrieving an Entity Bean, consider using direct JDBC calls from within a Stateless Session Bean to perform a “read” of the Bean’s data. Later, if you need to update or delete the data (or create new data), use an Entity Bean to control these tasks. This approach sometimes saves time by reducing expensive “finder” overhead. It also provides performance benefits to applications that perform lots of read activity, but few updates, on a complex database table structure.

Avoid Fine-Grained EJB Data Models Don’t overdo it with the Entity Beans. In some cases, we’ve seen a simple application request use literally hundreds of different

---

Figure 2.9  Enterprise JavaBean transactions using a façade Bean
Entity Beans. Obviously, your web site cannot handle high volumes if you load hundreds of EJBs on each call. Simplify the model and consolidate your data into larger Beans. Also, reduce the Beans involved by letting your SQL and the database do more work for you. Use database joins to consolidate your data at the database.

**Mark Methods as Read-Only** This setting reduces the database interactions required to execute your EJB methods and provides performance benefits to your application. Check with your application server vendor for the implementation of this J2EE feature.

**Consider Scalability Issues of Stateful Session and Entity Beans** Since these Beans have identity and state, each instance only exists in one EJB container in your web site. Stateless Beans, on the other hand, exist in any convenient container (often, the EJB container running in the same JVM as the web container), or in the container with the most processing power available. This flexibility often makes Stateless Session Beans more scalable than other EJB choices.

**Cache Bean Contents in Custom Finders** Custom finders returning an Entity Bean collection make \( n + 1 \) database calls: one to perform the “find” and one call per **Entity Bean instance** to instantiate the Entity Bean’s contents. Depending on the number of Entity Beans returned, their size, and the frequency with which you access them, you may decide to cache the results of a custom finder.

Before you implement a custom-finder cache, consider how much memory the cache requires. Again, caching at your server only works to your advantage if it does not consume excessive amounts of memory. Likewise, caching only works if the data retrieved does not change while cached.

**Avoid EJB Passivation** We discussed earlier the inefficiencies of the file system in high-volume web sites. EJB containers passivate inactive Stateful Session Beans if the container begins to run out of room when creating new Beans. Passivation dumps the state of the Bean to a file. If a subsequent request needs the passivated Bean, the container must interact with the file system to reload it.

To avoid passivation, remove unneeded Stateful Session Beans as soon as possible. Of course, eventually the container times out and removes old Stateful Session Beans, but if your web site moves high volumes, the container may be forced to passivate the Bean before the time-out. Use the EJB remove() method to remove unneeded Stateful Session Beans from the container. Also, consider shortening the timeout period for the Stateless Session Beans.

**Cache Homes for EJBs** As stated earlier in the discussion on servlets, avoid expensive JNDI lookups whenever possible. As we discussed in the servlet section, you should use caching techniques to reduce or eliminate repeated JNDI lookups. Consider broker
objects or other design patterns to manage JNDI lookups, as well as any other cacheable operations. (Caching EJB homes is always a good idea.)

**Tuning EJB Containers**

EJB containers from different vendors often include some unique settings that affect their performance, but some settings span all vendor implementations. Let’s take a quick look at some of the more important EJB container settings. Of course, you should check your application server vendor’s documentation for other tuning settings.

**Pool Size**

Containers maintain pools of EJB instances. You may configure the maximum size for these pools, which determines how many instances your EJB container controls at any point in time. (Some containers support a “per pool” setting, while others use a grosser measurement for the total EJB instances managed by the container.) Configure your pool size to contain the working set of EJBs your web application requires. Your pool should at least accommodate the EJB instances required to satisfy your maximum simultaneous requests. (Keep in mind that these requests may originate from multiple web containers in a large, distributed web site.) At the same time, you must prevent the EJB container from exhausting its JVM heap with excessive EJB instances.

Thus, the size of the EJB pool really depends on two factors: demand and data size. Use your performance test to help define an optimal setting for your EJB pool. Understanding how many EJB instances you need at peak loading, as well as the size of these instances, also gives you significant insight in setting an optimal EJB pool size. (Also, as with web containers, some EJB containers permit unbounded pool growth, which you should avoid because it eventually leads to memory exhaustion. Instead, enforce a hard maximum on the EJB pool size based on your testing. Different vendors support setting this hard maximum in different ways, so check the vendor’s documentation for details.)

**Cleanup Interval**

Some containers allow you to specify how frequently the container sweeps its pool for old objects. Reducing the interval between sweeps reduces the number of old objects left in your pool and gives you more room to create new objects as needed. However, running the sweeping routine too frequently may degrade performance, depending on the “weight” of the sweeping routine. Consider adjusting this interval during your performance testing to find an adequate setting for your application.
Transaction Isolation Level

The transaction isolation level controls the visibility of uncommitted changes within the underlying database. You may specify the application level at the EJB or EJB method level. (Additional rules apply if your application calls multiple methods with differing transaction isolation levels.)

For best performance, use as liberal an isolation level as possible. However, this really depends on the application as well as the other objects and applications sharing the underlying database. Too low a setting may result in unexpected side-effects such as multiple, simultaneous updates to the same data row. (Obviously, performance considerations take a back seat to data correctness issues.) Likewise, too high a setting results in application deadlocks.

Local versus Remote Interfaces

As more vendors implement the EJB 2.0 specification, the features of this specification level become more important. For example, you may decide to call your EJBs via local interfaces, rather than remote interfaces. Local interfaces work when the calling client (such as a servlet or JavaBean) shares the same JVM as the called EJB. By avoiding a remote method call, this protocol provides significant performance benefits.

However, you cannot use local interfaces with widely distributed web sites. If your EJBs reside in a remote JVM, you must use remote interfaces to access them. (See the discussion on clones later in this chapter.) Relying on local interfaces for performance limits your web site scalability. If you anticipate scaling your web site in the future, develop a performance strategy that assumes the possibility of remote interfaces.

Message-Oriented Middleware (MOMs)

The EJB 2.0 specification also introduces message-driven Beans. This feature enables Message-Oriented Middleware (MOMs) like IBM's MQ-Series, to call functions implemented within EJBs. Because this technology is so new, very few performance guidelines exist regarding it.

However, the reverse practice, where EJBs access a MOM for data, is quite common. The best practice for accessing MOMs via an EJB, or from any other object in web application (servlet, JavaBean, etc.), for that matter, is the following. First, be aware that MOMs operate asynchronously, while web applications operate synchronously. Many web sites try to mask the asynchronous nature of a MOM within the synchronous operation of their web application by placing the web application in a wait state until the MOM completes the requested task. However, MOMs do not guarantee delivery within a particular time frame; they only guarantee eventual delivery. Therefore, you cannot count on the responsiveness of a MOM. Be very careful accessing an
asynchronous MOM inside your synchronous web application. As noted earlier, indefinite wait within a web application often leads to disaster. Plan ahead for long response times and outages within the MOM system. Just because your MOM normally responds quickly, don’t assume that it always will.

Database Connection Pool Management

Database connection management in the servlet world proves difficult for former thick client developers. Thick client best practices encourage the programmer to obtain a database connection once and cache it inside the client for all future requests. This works well inside a thick client because it costs a lot of time to obtain a new connection; however, it is not practical on a web site. Your web site may support thousands of logged-in users at any given time and may run dozens of different servlets over the course of a minute. Obviously, assigning a database connection to a given user, or even to a given servlet, isn’t practical.

Instead, most web applications use pooled database connections. The most common pooling mechanisms are the `javax.sql.ConnectionPoolDataSource` or the `javax.sql.XADataSource`. The JDBC specification defines the `DataSource` class, which is in turn implemented by the vendors. Many early application servers also supported their own implementations of data sources, or similar pooling constructs, prior to the JDBC standard. The data source allows the administrator to define the number of database connections available to the running web applications. An example flow using a data source goes as follows:

1. A servlet obtains a database connection when required for database interaction.
2. The servlet uses the database connection to interact with the database and processes any returned data.
3. *Immediately* after completing the database interaction, the servlet returns the connection to the data source connection pool by issuing a `connection.close()` method.

By “immediately,” we mean within the method that gets the connection. Don’t pass a result set around to various layers (JSPs, other servlets, and so on) in order to manipulate the data from the database. Loop through the result set, pulling the data out into your own collection, and then close the result set, statement, and connection. (Failing to close the statement object before returning the connection to the pool sometimes results in an error for the next application reusing the connection.)
Database Connection Management Tips

The data source and its database connection pool make managing database connections trivial. However, many programmers fail to use the database connection pool properly, if at all. Let’s discuss some of the best practices for database connection management.

Return the Database Connection to Its Pool Quickly

Many servlet writers return the database connection to the connection pool as an afterthought when all the processing inside the servlet completes. This makes the connection unavailable to other servlets for significant periods of time (a half second or more, in some cases). Return the connection as soon as possible to the connection pool. Also, consider optimizing your code to use database connections more efficiently.

Return Connections No Matter What Happens

Make sure you close everything within `finally` clauses in your code. Otherwise, if an exception occurs, you may “leak” a database connection or leave a statement open. While the connection pool may eventually “reap” the lost connection (discard it and create a new one), this process only starts after a generous timeout interval. Too many lost connections may consume all of the connections in your database connection pool, and this may result in a frozen web site. Try to return connections to their pools even in the event of an exception. Also, try to close statements to avoid returning a connection to the pool in an unusable state.

Use the Connection Pooling Feature

A few servlet writers refuse to use the data source at all. They continually obtain and release database connections inside their servlets. The performance overhead of this approach is breathtaking. Use a connection pool, even if you need to make some changes in your existing code. The extra effort almost always pays for itself in enormous performance benefits.

“Custom” Connection Pool Managers versus Data Sources

Sometimes we may encounter shops using custom connection pools. Often, these shops wrote their own pooling logic prior to the adoption of the `DataSource` object in the JDBC standard. Connection pools contain surprisingly complex logic for things such as recognizing and reaping dead connections, growing and shrinking the pool in accordance with demand, and refreshing connections after a database outage. Often home-grown data sources require considerable tuning, and they may not scale as the site continues to grow. Take advantage of the tuning and expertise built into your vendor’s implementation and convert to the JDBC standard data source.
Web Services

Another relative newcomer to the web application space is the concept of web services. Regrettably, the web services technology lacks a body of performance expertise at this point. (Performance guidelines usually follow some number of practical experiences with a new technology.) Nonetheless, based on what we do know about web services in general, we tentatively offer some rough advice. Again, these touch web services performance at a high level. Use solid performance testing of your implementation to better understand your specific performance profile.

Use UDDI (Universal Description, Discovery and Integration) with Care
Repeatedly locating a web service, parsing the interface description, and mapping parameters to the interface seriously degrades the performance of your web application. For better performance, consider selecting the services you plan to use during the development of your web application. Directly code interactions with these services rather than repeatedly using UDDI at runtime. Of course, hard-coding services may result in some flexibility restrictions on your applications, but it may make sense as performance trade-off for heavily used web application functions.

Minimize SOAP Data Transfers  SOAP (Simple Object Access Protocol) is relatively slow. Web services applications often spend most of their time parsing and generating the XML used by SOAP rather than executing logic. Reduce the parsing overhead by keeping the data transferred via SOAP to a minimum.

Monitor Changing Web Site Usage Patterns  Providing web services functions often changes the usage patterns of your web site. Most web site designs focus on direct interactions with a web user. However, web sites providing web services functions often find themselves playing the role of “middleware” for other web applications using their functions. For example, a search engine or sports site often handles more requests for information actually displayed by other web sites than direct requests from web users.

This new role as middleware often shifts the priorities of the web site providing the web services function. For example, key features such as the web site’s home page become less important. However, other features, such as returning the high school basketball scores for all teams in a particular zip code become heavily used through the web services function.

If your web site provides web services functions, monitor the usage patterns of your web site. You may discover a need for additional resources, such as database connections, to support increased demand for certain web application functions. Also, this increased demand may require performance improvements in previously underperforming (but seldom-accessed) functions now receiving considerably more traffic.

Other Features

Your web application server might use other features not defined in the J2EE specification. We’ve included below some performance pointers for a few of the more common of these features.

Built-in HTTP Servers

Many web application servers include their own, built-in HTTP servers. By using the built-in HTTP server, the web site avoids the performance overhead of the plug-in processing normally associated with a stand-alone HTTP server. These built-in HTTP servers often run “out of the box” with little setup, which makes them particularly appealing to web site teams hard-pressed for time.

While the built-in HTTP servers have a lot going for them, we do not recommend them for high-volume, production web sites. Many high-volume web sites place the HTTP server inside a DMZ, and run the web application server behind the DMZ. Built-in HTTP servers cannot support this configuration. Also, stand-alone, commercial HTTP servers provide more features and support high volumes and static content requests more efficiently. (See the next chapter for more on HTTP servers and their performance.)

General Object Pool Management

The JDBC data source standard we discussed earlier applies only to database connections. Sometimes you need a custom pool manager in your web application to control sharing of other objects, such as connections to a non-JDBC-compliant data repository or other sharable objects. For example, if your web application launches threads, you need a thread pool. The thread pool reduces overhead from repeatedly creating and destroying threads, and prevents some runaway conditions.
General Object Pool Management Tips

If you find yourself writing a custom pool manager, consider the following best practices.

Use Configurable Limits   Load the parameters for your thread pool from an external source, such as an XML file or resource bundle, during initialization. This allows the administrator to fine-tune the pools in your application during performance testing or deployment without making a code change. Key parameters include the minimum and maximum pool size, timeout values, growth values, and so on. (Some web applications build their own custom administration function to update these values on the fly.)

Never Wait Indefinitely Inside the Pool   The servlet requests a thread from the pool, but none are available. The pool manager decides to grow the pool by another five threads (the pool has not reached its maximum size yet). What happens to the servlet request while the pool grows? Do not allow it to wait indefinitely. Time out the request and/or throw a “soft” exception if the pool cannot obtain new resources in a very short period of time (milliseconds, not seconds). It is better to return empty-handed than not return at all from a resource request. Avoid stalling the web site by leaving servlet threads in an indefinite wait state. (See the discussion on “frozen web sites” earlier in this chapter.)

Set Pool Maximums   To avoid runaway thread creation, set hard maximums for your thread pool. If your threads interact with a back-end system, they might stall if that system becomes nonresponsive. In this case, your thread pool might fill up with stalled threads waiting for a dead remote system. Creating more threads only makes the situation worse, and may lead to a system crash if you allow thread pool to create hundreds of threads. Instead, return an exception to the servlets requesting threads. Program the servlets to catch this exception and return the appropriate error page, if necessary. This is another example of making outages a normal part of your site’s operation.

Set a Maximum Wait for Thread Execution   After retrieving the thread, take care when launching it. Again, you cannot wait indefinitely for a thread to return. Always manage the amount of time your servlet waits for a thread before continuing, particularly if your thread tries to access remote resources. Use a timeout on your threads. Instead of

myThread.join();

use the join with a timeout parameter:

myThread.join(500);
Again, whenever possible, make the timeout value externally configurable. Also, if you set a timeout, you may abandon threads over time. This is another reason to set an absolute maximum for your thread pool to avoid runaway thread acquisition.

**Multiple Instances: Clones**

Many larger Java web application server vendors provide an easy way to create and manage multiple application server instances (each running in a separate JVM) on your web site. These multiple JVMs may execute servlet applications, or EJB containers, or both. The IBM WebSphere Application Server, for example, calls these multiple JVMs *clones*; other products call them *instances*. Clones may share the same machine, or install across multiple machines, or both. Figure 2.10 shows multiple clones within a server cluster.

Clones play a useful role in the performance of your Java web site. For very large multiprocessor servers, clones allow you to take better advantage of the CPU and
memory available on the server. Clones assist with large-server vertical scaling. As discussed in Chapter 1, a single JVM cannot take full advantage of the resources available on an extremely large server. For example, if you use a 24-way, 64GB RAM server machine, you cannot expect to get full utilization of this box with a single JVM. If the single JVM uses 50% of the CPU available, adding another cloned JVM might raise that to 80% or 90%.

Cloning also allows the administrator to manage application server instances spread across multiple machines. This gives the administrator flexibility to better allocate resources, as well as centralized control over the web applications executing on the server machines.

**Cloning Performance Tips**

As with any other feature, cloning requires careful use to avoid performance pitfalls. Let’s discuss some cloning performance best practices. (See Web Site Topologies in Chapter 3 for more on clones and the best practices for their deployment.)

**Too Many Clones, Too Little Resource** Cloning does not solve problems for overloaded servers. If your server machine uses 100% of the CPU capacity (or even close to 100%), *do not add clones to your server*. The CPU is saturated. Instead, add CPU capacity to your system, or tune your application. Do not increase the CPU burden by adding clones to the server.

**Scale Your Resources as You Add Clones** If you add clones to your servers or web site, don’t forget to add resources to support them. Each clone requires database connections, heap space, and network capacity. If you add clones, add the resources the clone needs to operate.

**Summary**

This chapter covered many aspects of Java application servers, including settings on the various containers, coding practices, and performance issues with various J2EE features. However, your web site is more than your Java web application. The next chapter discusses some of the other key “players” such as the HTTP server and the firewall. Again, your web site is a system. Tuning your web application while ignoring other web site components usually leads to an underperforming production system.
Chapter 6
DEVELOPING A
PERFORMANCE TEST PLAN

Performance tests require a test plan. Before writing the first test script or starting the first test run, you need to develop test goals and a plan for obtaining the desired measurements. Well-designed tests tell you if the site achieves key goals, as well as how the site needs to grow in the future as traffic increases. A poorly designed test usually provides misleading information.

In Chapter 5, we discussed the different performance issues and priorities of various web sites. Keep these differences in mind as you begin your test planning. While understanding throughput and response time is important, some web sites require additional testing to determine the maximum logged-on users they support. Likewise, each site needs realistic test scenarios in order to accurately stress key components such as caches, databases, and application servers. A good test plan manages all of these requirements.

Beyond the measures, we need goals before entering the performance test. Performance tests, as more thoroughly described in Chapter 11, cycle through an iterative process of testing, resolving bottlenecks, and retesting. Without goals, the cycle of testing and resolution often continues indefinitely. You must identify the point of diminishing returns for your testing before you start.

This chapter discusses how to interpret rough performance requirements to develop test goals. Also, we discuss the basics of response time measurements, as well as testing for maximum logged-on users. Beyond setting goals, the chapter covers strategies for test implementation. For very large sites, generating the full, peak user load proves impossible. We cover strategies for validating web site performance by testing small portions of the site. Finally, we discuss how to use the performance test for planning future site growth. Properly designed tests uncover the weakest points in the web site and teach us how to better manage these weaknesses as the site grows.
Test Goals

First, before starting performance testing, complete a rigorous function and integration test of your web applications. This ensures that the applications function properly (at least in the single-user case) before you begin load testing. Remember: You cannot performance test broken applications. They behave erratically and may exhibit much better or much worse performance than a correct version. After completing function and integration testing, it’s time to make sure that your application and supporting infrastructure supports the expected traffic volumes.

Start your performance test planning by setting performance goals. Without clearly defined goals, performance tests tend to continue indefinitely. Any web site contains an infinite number of bottlenecks. Solving one bottleneck merely exposes the next one. Without clear targets, your test team never knows if and when the web site reaches a sufficient level of performance to support its expected traffic. Setting goals lets you know when the web site is ready for production.

In Chapter 1, we discussed some of the basic performance concepts, such as peak load, throughput, response time, and scalability. In this chapter, we discuss how to develop performance goals for each of these concepts.

Peak Load

The performance test must simulate the web site’s peak load. As we discussed in Chapter 1, if the web site does not perform well at peak loading, it does not really perform at all. We use the performance test in conjunction with the anticipated peak load to accomplish the following:

- Determine if the web site currently supports this goal.
- Tune an underperforming web site so that it supports peak loading.
- Determine the hardware capacity required by the web site at peak.

Surprisingly, many performance tests overlook the importance of peak loading. These tests frequently use average load numbers as their performance test targets. However, as Figure 6.1 demonstrates, average load often differs dramatically from the peak. Merely testing for average load sometimes means your web site cannot support key periods over the course of a year or even a day.

Also, use the numbers from your web site’s peak usage period. As we discussed in Chapter 5, many e-Commerce web sites experience their busiest days during the Christmas shopping season. Take your peak daily load from the busiest day of your busiest season.
While determining the peak load sounds simple enough, in practice it often proves challenging. If the site currently exists in a production environment, use data from the existing site to gauge traffic patterns and peak usage. Having said this, we should say as well that we encounter many web masters and web site teams who have no idea how many users access the site per hour, per day, or even per year! If the web site team lacks this data, many tools exist to pull site usage statistics from HTTP access logs.

An existing thick client application also presents another avenue for predicting peak load. For example, reviewing the host database transactions generated by the thick client often provides insight into peak loading. (To generalize a bit, DBAs tend to keep much better records than web site teams anyway.) Of course, this technique works best when the web site under test is the planned replacement for the thick client application.

Even data from similar web sites frequently proves useful. If these sites serve the same target audience as your web site, or provide a similar class of functionality, their traffic patterns might help you set expectations for your site. The key word here is similar. Arbitrarily selecting a web site as a pattern for a completely different site rarely works, as we discussed in Chapter 5. For example, assume your company supports both an e-Commerce site selling retail clothing and a site supporting financial interactions. These two web sites receive very different patterns and peak load volume; you cannot use the data from either site to predict the peak load of the other. However, the traffic patterns from the “shoe store” portion of the e-Commerce site probably apply quite well to a new “handbag store” function on this same site.

If you performance test a completely new web site or function (or target the web site to a new user market), estimating peak load becomes more difficult. In these cases
you rely heavily on projections based on any available market research. This includes any public data available on similar web sites operated by other companies. However, in the end, you end up with a best guess in these situations. We suggest generously buffering any peak load estimate you develop for a completely new web site.

As you gather these numbers, don’t forget about growth data as well as current usage data. Predicting future usage is also tricky, but important. By the time you finish a performance test, today’s peak load no longer applies. Analyze past growth, and discuss your early estimates with marketing and development to generate your growth targets. In particular, check with marketing for any plans that might dramatically increase load. Promotional mailings, television advertising, or new products all stand to dramatically impact web site loads. Again, generously buffer any load-growth estimates you develop.

Even though the performance teams we advise often begin their test planning without good peak loading estimates, these same teams usually develop reasonable and relevant estimates after consulting the available data. Teams without loading estimates often release web sites without sufficient traffic capacity. At the other extreme, we sometimes find teams building enormous web site infrastructures to support tiny user loads. Good loading estimates lead to a properly sized web site.

Peak loading also provides the basis for throughput estimates. In the next section, we discuss how to use rough load data to develop some estimates for both peak loading and throughput.

**Throughput Estimates**

As you begin your search for load and throughput estimates, expect an influx of ill-defined goals. For example, we often hear requirements such as, “The site needs to support 10,000 hits a day,” or “We expect 10 million users per year,” or (the most ill-defined) “We anticipate 100,000 users a day.” You must turn this data into reasonable estimates for building a performance test. In this section, we demonstrate some of the formulas we use for turning rough guesses into the estimates you need. Appendix A also presents these calculations in spreadsheet format to help you better organize your data.

*Important!* After developing rough loading and throughput estimates, feed them back to the other teams involved with the web site (such as marketing or management) for validation. These numbers only work if they reflect reality. All of those involved with the web site must agree with the targets you develop for the performance test and, ultimately, for the production web site itself.
Hits per Day Estimates

So the marketing team tells you, “Expect 10,000 hits per day for the site.” Before you take this estimate at face value, we suggest asking a few questions. As mentioned in Chapter 1, Internet terminology often varies between companies or even between departments in the same company. Let’s use the marketing team’s statement to explore how terminology usage impacts performance estimates.

What Is a Hit?

A “hit” potentially means a couple of different things. For an HTTP server specialist, a hit means any request sent to the HTTP server. Because HTML pages usually contain embedded elements such as gifs or jpegs, one HTML page might result in multiple HTTP “hits” as the browser retrieves all of the elements from the server to build a page. Figure 6.2 shows a web page constructed from multiple HTTP requests.

If your web master tracks site activity via the HTTP access logs, a hit might actually mean just an HTTP request rather than a full page. For example, if the average page on the site contains four embedded elements, our marketing team’s “10,000 hits per day” number translates to become as few as 2,000 page requests per day:

\[
\text{Requests to display a full page} = 1 \text{ page request} + 4 \text{ embedded element requests} = 5 \text{ requests}
\]

\[
10,000 \text{ requests per day} / 5 \text{ requests per page} = 2,000 \text{ pages per day}
\]
This assumes every request for a page also requires the browser to retrieve all the embedded elements. This isn’t always the case, as most browsers cache static page elements so they do not have to request them repeatedly. If a web site’s pages contain many of the same embedded static elements on every page (for example, a banner common to all pages), the user’s browser may not retrieve this element again after the first page of the user’s visit.

These considerations become important as we try to find out how much dynamic content our web site serves. We need these estimates to develop some key capacity planning ratios for our web site. We want to know how many HTTP servers we require as compared to application servers. This also allows us to predict the value of specialized components, such as caching proxy servers, within our web site.

In the example above, let’s assume our page request is a dynamic page request requiring service by the application server. Let’s also assume the static elements do not require interaction with the application server (they’re handled by the HTTP server, not the application server). In this case, our dynamic request to static request ratio is 1:4. Establishing this ratio helps us to better tune our web site (for example, configuring the ratio between HTTP server listeners and application server threads). It also helps us to plan capacity for the application servers if we understand how much traffic they receive.

Returning to our example of 10,000 hits per day, we know this number resolves to 2,000 page hits (representing dynamic content) if the browser requests all the embedded static elements for each page. If the browser caches static elements, the 10,000 hits per day measurement contains a higher percentage of dynamic page requests versus static elements. Consider the impact of caching on your request ratios.

Regrettably, the rest of the world uses the term hit in very ambiguous ways. We’ve seen hit used to refer to an entire page, including embedded elements or even entire framesets. Also, many companies routinely use hit to mean an entire site visit by a given user. A site visit usually encompasses many pages, not to mention the embedded elements and frames included in those pages.

This leads us to an important point: Use performance terminology consistently. However you define a hit, define it explicitly, and stick with the definition uniformly throughout your planning and testing. Terminology misunderstandings make for unsuccessful performance tests.

What Is a Day?

Beyond the concept of a hit, what does the marketing team mean by the term day? In the world of web sites, even the concept of a day lies open for interpretation. A day
usually defines the primary period of activity for a web site. Many, many web site teams wrongly assume their site will be fully populated by happy users 24 hours a day. With few exceptions (such as worldwide sporting event sites), the average web site does not receive a great deal of traffic on a round-the-clock basis. Instead, most users arrive during the period of a business day. Defining the day becomes very important for determining the peak load on the web site. As shown in Figure 6.3, overestimating the length of the day conversely underestimates the peak pressure on the site. Determining the period of time when most users arrive on the system becomes critical for estimating peak load.

**Determining Peak Throughput from Hits per Day**

Given 10,000 hits per day, how can we turn this number into a peak throughput estimate? The ideal estimate comes from the traffic patterns of an existing web site or application, as mentioned above. For example, if the web master tells us to expect 30% of the traffic during the peak hour,\(^1\) expect 3,000 hits per hour during the peak period.

---

1. Your web site may track a smaller or larger peak time interval. Use the peak interval your team can describe with the most confidence.
period. This number gives us 0.8 hits/sec during the peak hour. Here’s the math to get these numbers:

\[
10,000 \text{ hits/day} \times 30\% \text{ of traffic at peak hour} = 3,000 \text{ hits during peak hour}
\]

\[
3,000 \text{ hits per hour} / 60 \text{ minutes per hour} / 60 \text{ seconds per minute} = 0.83 \text{ hits/second}
\]

Again, keep in mind, depending on the definition of hit, this may resolve to more traffic. For example, if a hit really represents a whole page containing 30 embedded elements, this number resolves to as many as 24 HTTP requests per second. Understanding the meaning behind the number makes the difference in these estimates.

Frequently, however, you receive a daily or yearly traffic rate but no guidance on how much of the traffic arrives during the peak period. In these cases, applying some calculations to the available information yields a rough peak estimate. This algorithm first spreads the traffic evenly over the website’s workday. For this example, let’s assume the website receives most of its traffic during an eight-hour period:

\[
10,000 \text{ hits per day} / 8 \text{ hours per day} = 1,250 \text{ hits/hour}
\]

Notice that this gives us an even distribution with no consideration for peak loading. To estimate the peak, we normally assume three to five times the weight of the daily average. For safety, in this case, let’s assume the peak is five times the average.

\[
1,250 \text{ hits/hour} \times 5 = 6,250 \text{ hits/hour}
\]

This gives us the following value for the per second hit rate at peak:

\[
6,250 \text{ hits per hour} / 60 \text{ minutes per hour} / 60 \text{ seconds per minute} = 1.7 \text{ hits/second}
\]

After arriving at this step, we proceed as follows.

- Convert the number to peak HTTP requests. For example, if our numbers represent page requests instead of HTTP requests, we do more work. If each page request translates to seven HTTP requests, the peak HTTP request rate becomes

\[
6,250 \text{ hits per hour} \times 7 \text{ HTTP requests per hit} = 43,750 \text{ HTTP requests per hour}
\]

\[
43,750 \text{ HTTP requests per hour} \Rightarrow 12.15 \text{ HTTP requests per second}
\]
After converting to HTTP requests, apply the dynamic to static ratio. From an earlier example, we know each dynamic page request generates four additional static requests. This means that, of our 12.15 requests/second:

12.15 hits per second * 1/5 = 2.43 dynamic hits per second
12.15 hits per second * 4/5 = 9.72 static hits per second

Validate these calculations with others involved in the site, such as marketing or management. Always review the calculations in light of the real site.

**Users per Day Estimates**

Regrettably, the *hit* does not reign as the most ill-defined term in web site performance. The term *user* leads the pack as the most misunderstood performance metric. Often, the web site team receives guidelines for site performance such as “the site needs to support 100,000 users per day.” Of course, this doesn’t give us any information about what these 100,000 users will be doing on the web site.

So, once again, we must dig into the number to determine how the users interact with the site. Likewise, we need to develop some idea of how the site is constructed to understand the burden the typical user interaction places on the web site. Finally, we use all of this information to develop peak load estimates.

**What Is a User?**

*User* usually means one of three things in terms of web site usage:

1. One of potentially many “visits” the user will make to the web site over the course of a day
2. A complete set of activities accomplished by the user on the site for the whole day
3. A hit

**Users as Hits**

Let’s start with the last definition: a hit. Many web masters and web site teams use *hit* and *user* interchangeably. So when they say “user,” they really mean “user request.” So, in this case, they expect 100,000 user requests per day. Usually, this translates best to pages, but, as with the discussion above, hit can mean a variety of things.
All Day Visitors

The second definition is often used in B2B, portal, and some other classes of websites where the users log on to the site for the entire day. In these cases, a user represents a complete set of activities accomplished by the user for the whole day. These users may or may not be very active. If they infrequently make requests throughout the day, the primary website burden may well be the “footprint” of the user on the system.

This user “footprint” includes any resources required to keep the user active on the system all day. For example, the system sometimes uses memory to keep HTTP session data about the user. Also, the website may save the user’s session to a persistent HTTP session database. The performance team must design a test for these systems simulating the burden of logged-on, but largely dormant, concurrent users.

User Visits

The term visit defines the most common usage of user. The website receives many visitors over the course of the day. These visitors interact with the website, performing one or more tasks that usually involve navigating through multiple pages. The user stays logged on to the site until shortly after his activity finishes (usually 15 or 30 minutes after the last request the user makes). After this timeout period, the website removes the user’s transient information (the HTTP session data) from the site. If the user wants to interact with the site later, he must log on to the site again.

User Scenarios

Regardless of the loading goal (users, hits, or transactions), eventually you must determine the load’s distribution, or how the users interact with the website. Usually, this requires developing several scenarios and determining the percentage of traffic executing each scenario at any given time.

For example, an e-Commerce site might receive users who want to browse, purchase, or check the status of an order. A financial site might receive visitors who want to check their portfolio value, trade stocks, or do research on a particular company. We then develop these usage scenarios as test scripts for each of these tasks, and estimate their relative execution frequency.

Scenarios come from several sources. Usability testing or analysis of site logs gives the most reliable data about usage patterns. For new websites without the benefit of usability testing, we often look to use cases from the analysis/design phase of the web application to provide scenarios. However, the best scenarios involve real data on how the users interact with the website. (See Chapter 7 for more details on developing test scripts.)
On the basis of these scenarios, we develop an idea of how many pages the users access for each scenario. However, we also need an estimate for the duration of the average visit. A good choice for this number is how long it takes us to run the scenario manually (see the caveats applying to this in Chapter 7). Even better, we might obtain some data from early usability testing. However, most sites do not perform rigorous usability testing, so a rough guess from walking through the scenarios is a good first start.

The frequency of each scenario is important as well. We must determine how many users execute each scenario over the course of the day. The scenario utilization becomes very important if some scenarios require “heavier” activities, such as database interactions, or a large number of page visits per scenario. Usually, we derive the scenario mix from the existing web site or receive this weighting from the marketing team. Scenarios tell what our traffic is doing. For “users per day” estimates, we also need the scenarios to tell us how much traffic our users generate during peak periods.

**User Visit Projections**

So what do user arrival numbers mean for web site load and throughput projections? Let’s assume the following information from the marketing team:

- 100,000 user visits per day
- 30% of traffic arrives during peak hour

After some clever and dedicated investigation work, you uncover the following information about the web site:

- Average user visit: 10 minutes
- Session timeout: 15 minutes
- Average pages per visit: 5

During the peak hour, the web site has the following new user arrival rate:

\[ 100,000 \text{ users} \times 30\% \text{ peak users} = 30,000 \text{ peak users} \]

\[ 30,000 \text{ peak users/hour} \Rightarrow 8.33 \text{ new users/second} \]

Over a ten-minute visit, at a minimum, the web site contains the following concurrent users:

\[ 8.33 \text{ new users/second} \times 10 \text{ minutes} \times 60 \text{ seconds/minute} \]
\[ = 4,998 \text{ concurrent users (approx. 5,000)} \]
These users typically stay active for ten minutes and visit five pages during that time. This gives us the following page rate:

\[
5000 \text{ users} \times 5 \text{ pages per user} / 10 \text{ minutes} / 60 \text{ seconds per minute} = 41.667 \text{ pages/second}
\]

(approx. 42 pages/sec)

If each page contains an average of five embedded static elements per page, the overall throughput (in requests per second) is as follows:

\[
((1 \text{ initial request}) + (5 \text{ static requests})) \times 42 \text{ pages/second} = 252 \text{ requests/second}
\]

Obviously, browser caching might reduce this request burden, but it serves as an upper bound for testing purposes.

User arrival rates and active log-on rates also help determine the HTTP session pressure of these users. If the Java web site uses HTTP sessions, each user may use memory on one or more application servers. This memory only frees after the user’s HTTP session times out (or, in some cases, when the user logs out of the web site).

Let’s assume each concurrent user requires about 2KB of memory. Since the average user visit is 10 minutes and, in this case, the HTTP session timeout is 15 minutes, the average session stays in memory a total of 25 minutes. Therefore, the server requires the following memory to hold these sessions:

\[
8.33 \text{ users/second} \times (25 \text{ minutes}) \times 60 \text{ seconds/minute} = 12,495 \text{ users (let’s say 12,500)}
\]

\[
12,500 \text{ users} \times 2\text{KB/user} = 25\text{MB of memory}
\]

Of course, this equation does not take into account any users still logged on from previous arrivals outside the 15-minute window. (Remember, the 10-minute visit time is an average.)

**Response Time Measurements**

The web site requirements consist not only of load, but also responsiveness to the load. Response time requirements may be very general, such as “no more than five-second response time for any page at peak load.” Some teams receive requirements broken out by pages. For example, a financial site might require two-second response time on stock quotes, but accept ten seconds for research pages.
Response time requirements change as the industry changes. The goal is to keep your response time consistent or better than the customer’s experience at other web sites. Usually your QA or marketing teams provide response time targets based on current industry “standards.”

Defining the Test Scope

After determining the peak loading and throughput, we start designing the performance test. A good performance test, as we mentioned earlier, tells us how to build the web site to support peak loading. It also tells us how to prepare the web site for future growth. To make these preparations, we need detailed information on how key components of the site, such as the servers, databases, and networks, perform under increasing load.

The test requires a starting point. Even for the most complex web sites, the best tests begin with a single server. Determining the capacity of just one machine tells us the following:

- The accuracy of capacity estimates on a per server basis
- The database and other back-end resources required by one server
- The network capacity consumed by this server
- Front-end resources such as HTTP server and router capacity consumed to support this server

For example, if we estimate we need ten servers to support the 100,000 user day from the previous section, let’s validate this estimate early in the performance test. Begin the validation process by determining how much load each server can support. (If you want to learn how to actually estimate the equipment required by your web site, see Chapter 15.)

The estimates calculated previously from this loading estimate show a peak load of 5,000 users over a ten-minute period. These users request 42 pages/second at peak, which translates to 252 requests/second to the web site. The web site requires 50% headroom (double the capacity it needs for the projected peak). This gives the site a generous margin for error in case the initial capacity estimates are wrong, and it also gives the site capacity in case of a pathological surge in traffic caused by some unforeseen event.
Given this, we estimate each server handles 8.4 pages/second or 50.4 HTTP requests/second:

\[
\frac{(42 \text{ pages/second} \times 2 \text{ headroom cushion})}{10 \text{ servers}} = 8.4 \text{ pages/second}
\]

\[
\frac{(252 \text{ requests/second} \times 2 \text{ headroom cushion})}{10 \text{ servers}} = 50.4 \text{ requests/second}
\]

Likewise, this transaction rate represents the workload generated by a fraction of the peak users. Using a similar calculation, we get an estimate for the number of users supported by a single server. In this case, the test for a single machine simulates the workload generated by 1,000 users:

\[
\frac{(5,000 \text{ peak users} \times 2 \text{ headroom cushion})}{10 \text{ servers}} = 1,000 \text{ concurrent users/server}
\]

This number represents the maximum number of concurrent users expected per server during peak loading.

**Building the Test**

After some analysis to develop peak loading estimates, we build the actual test to simulate peak conditions. We want to begin the test with a single server and determine how many requests per second and users the server can support. Based on preliminary pen-and-paper estimates, we expect to get 1,000 users and 50.4 requests/second with this server. Testing is the only way to determine if these estimates are valid.

**Setup**

In order to simplify this example, let’s map each concurrent user to one virtual user. This means we’ll need 1,000 virtual users to perform our testing. (Later in the book, we’ll discuss some techniques for reducing the number of virtual users required for a test.) Next, let’s establish the scenarios these virtual user will execute, as well as the proportion of users assigned each scenario. For our example, let’s assume we’re testing a financial web site, and we want the following test scenarios and weighting:

- Check portfolio—50% of users
- Perform trade—30% of users
- Perform market research—10% of users
- Setup new account—10% of users
The next step, of course, is to turn these scenarios into test scripts. At this point, we also need to answer some key questions about the scripts, such as whether they should include think time, cookies, or browser caching. After completing scenario preparation and validating that the performance test scripts execute correctly and without generating errors on the server, the test actually begins against our single server.

**Simple-to-Complex Strategy**

If at all possible, start with a very simple test configuration, and work outward to more complex scenarios. Consider the test configuration shown in Figure 6.4.

Begin with a simple configuration of one application server, one HTTP server, and a database. If possible, start the tests without firewalls and SSL. After resolving bottlenecks in this environment and doing initial tuning on key components, begin increasing complexity by adding components to the test environment incrementally. After adding each new component, retest to determine its impact on performance, tuning as necessary, before adding another component.

![Figure 6.4 An example test environment](image-url)
By working from the inside of the web site outward, and reducing the complexity of initial tests, we learn

- How the fundamental components of the system perform. If key components such as the web application suffer from severe performance problems or require substantial tuning, we need to find and address these problems as early as possible in the testing. Web applications tend to have the greatest impact on site performance, and they merit careful performance analysis.
- The impact of each component on performance. By knowing the capacity of the base systems, we can measure the impact of additional components, such as a firewall, on the test environment's performance. This also proves very useful in planning for site growth, and for determining problems in production.

The performance test provides validation for capacity planning estimates. It also helps us identify underperforming components in test environment.

**Scalability Testing**

So far, we've discussed testing with a subset of the production web site. We begin by testing and tuning this subset of equipment, and continue until we reach our performance goals for this portion of the web site. However, we cannot assume the performance we achieve for the subset applies to the web site as whole. To achieve this confidence, we need a scalability test. During this test, we add capacity to our test environment and determine whether the web application actually takes advantage of this capacity. That is, does the web application scale to use additional resources?

Scalability testing usually doubles capacity at each step of the test. We start with a subset of our web site that supports some amount of user load. If we double the subset's capacity, we also expect to double the traffic the subset supports while maintaining the same response time. We call this linear scaling: As you multiply capacity, the throughput and users handled by the web site multiply as well, while maintaining the same responsiveness.

Scalability determines the behavior of the web site as a system, including its growth characteristics and limiting factors. These tests tell us how adding capacity at one point in the web site impacts the capacity of other components. We need this knowledge as we build our web site, as well as when we grow it in the production environment.

**Building the Performance Team**

When we consider building a performance test, we often overlook the people who make the testing successful. Testing requires skills from every component used by
your web site throughout all phases of the testing cycle (planning, execution, measuring, analysis, and tuning).

For new web applications with links to existing data or function, it’s fairly common to focus exclusively on the newly created function and to ignore what’s happening inside the existing components. This myopic approach to testing frequently proves disastrous. Web applications work differently than existing thick client applications: They exert more simultaneous requests, and often use the existing applications differently. In short, legacy applications often require tuning for the web environment.

For example, consider a legacy application used by in-house stock brokers. Customers call the brokers, who then use a thick client application to create orders, retrieve pricing, or check the customer’s account. The company decides to build a web application giving customers direct access to these functions over the Internet. However, your new web application uses the same business logic at the mainframe as the old thick client application.

Not surprisingly, this web application generates many more transactions per day than the thick client application. Along with more transactions comes the need for more connections to your back-end database—which implies potential licensing and performance impacts not only on the new application, but also on those who continue to use the old system.

Obviously, you need to get the team supporting the mainframe business logic involved with your testing. Involving them early in the test planning phase gives them time to understand your performance goals and to do some tuning in preparation for your testing. As you continue the testing, they assist by capturing data at the mainframe and performing additional tuning.

Beyond this one example, consider the peripheral systems involved in your test, and pull together the right people to make the test successful. Below is a list of folks we often find useful as part of the larger test team. Review this list and consider the relationships you need for a thorough and accurate test.

**Quality Assurance People**

We consider quality assurance people first because they’re often overlooked. Some enterprises relegate performance testing to something outside their standard QA processes and only assign the application programmers to the task. This is a mistake.

Performance testing really operates as an extension of QA, assuming that your QA team is in the business of assuring that applications behave satisfactorily when put into production. Many QA groups actually own the performance test. This includes creating the test scripts based on either the design or current customer usage patterns.
Even if QA does not own the test, they often make a good resource for test scenarios, as mentioned above. They also often provide many of the performance targets (such as maximum response time) used in the testing.

**Web Application Programmers**

Use the application programmers to troubleshoot the application during testing. Problem analysis usually becomes one of the biggest headaches in deploying a distributed application. The test team needs someone intimately familiar with the application’s internals to fix problems as they arise.

**Web Application Architect**

You need the application architect’s input to build a good test. For large applications, use the architect's guidance to identify the subset of function and infrastructure required to accurately model the entire application's behavior. The architect also understands how users should interact with the web site. Use this information when designing your test scripts. However, always try to back up this information with observation of actual users interacting with the system.

Finally, the architect knows all the systems, such as databases and content servers, that are interacting with the web application. Use this information both in designing your test system and in resolving any performance problems you encounter during the testing.

**System Administrators**

System administrators set up and configure the test machines. During the test, they monitor the resource consumption on all the machines in order to identify machines and processes that are acting as bottlenecks.

**Network Administrators**

Just as your system administrators install, configure, and monitor the servers and clients, the network administrators do the same for the network. They validate your capacity estimates, set up an isolated network (hopefully), and monitor the network during testing.

**Systems Analysts**

Systems analysts monitor the legacy systems used by your web site. As we discussed earlier, new web applications often drive legacy systems much harder than existing thick client applications. You need a systems analyst to monitor these legacy components during your performance testing. They also provide important information about planned outages and backups of the systems, which you need for scheduling
performance tests. Also, they manage the legacy resources and help you schedule your testing so that you do not interfere with existing production applications.

**Database Administrators**

Almost all web applications connect to databases. Large web sites may connect to a variety of databases on multiple systems, including mid-tier and host systems. Each database used by your web site often has a different DBA or support team. Find these folks, and include them on your team before the testing begins.

Work with the DBAs to configure the database for the traffic you’re planning to send its way. In particular, coordinate available database connections to match the size of your web site’s database connection pools. During the testing, use the DBAs to monitor the databases for potential performance problems.

The DBAs are very important players on the web site performance team. These folks, along with the systems analyst, have skills you cannot reproduce on your team in the short term. Call on them early to help you understand the issues regarding their systems before a problem occurs.

**Legacy Application Programmers**

Often web applications use legacy applications in ways unforeseen in the legacy applications’ original designs. Having a legacy application programmer assigned as a point of contact and assistance is important. It’s important to build teamwork and a sense of ownership for the performance test within this broad group of support personnel. Obtain commitments from this varied group of folks and their management to make your testing successful.

**Summary**

Performance testing requires planning. Responsible performance targets, accurate test scenarios, and a reasonable testing strategy all lead to accurate test results. In this chapter, we covered some of the highlights of performance test planning. Everyone uses terms like *hit, user, and day*, but regrettably we often use them to describe a variety of different concepts. It’s important to get agreement ahead of time on the definition of these terms as they are used in your planning.

We also provided some examples of developing performance estimates for your web site. Starting with interpreting data from your existing web site or working with estimates provided by your marketing team, we showed how to create rough estimates for your planning and testing. We also discussed the difference between peak and
average load, and emphasized the importance of planning for peak load usage. Appendix A contains worksheets for most of these estimates to help you better organize your data and develop a test plan. We suggest you use this plan to drive discussion with the various parties involved with the web site (marketing, management, development, etc.) in order to achieve consensus on your performance goals before beginning the test.

Also, we discussed the basics of scenario development and scalability testing. Upcoming chapters explore these topics in greater detail, complete with a case study applying best practices. However, as you learn more in these chapters about executing a performance test, keep in mind the importance of a good test plan. If your fundamental goals and assumptions prove erroneous, your test results are worthless.
In the span of just a few years, the Internet has transformed the way information is both provided and consumed throughout the world. Its hardware and software technologies have made it possible for anyone not only to be an information consumer, but also for nearly anyone to be an information provider. Although the Internet—specifically the World Wide Web (the Web)—has been treated seriously as a platform for information sharing among the mass public for only a short time, many organizations have managed to create useful Web applications that provide significant value to consumers.

These Web applications allow consumers to buy books and compact discs online. They enable businesses to use the Internet for secure data transactions. Workers use Web applications to find jobs; employers use them to find employees; stocks are bought and sold using online applications provided by brokerages; and travelers book flight and hotel reservations using Web applications. The list goes on and on. Obviously, many useful Web applications are available on the public Internet as well as within countless corporate intranets today.

This book describes general techniques for building high-performance and scalable enterprise Web applications. Generally speaking, this means building applications that are reasonably and consistently fast and have a strong, gradual tolerance for rising user and request demands. Although we will spend a lot of time considering this topic in general, the core of our discussion will be phrased in terms of a solution built around the Java 2 Enterprise Edition (J2EE) specification. Now, before we dive into the details of building these kinds of applications, it is important to identify and understand the overall problem. More specifically, it is important to define Web applications and scalability.
Basic Definitions

In this book, Web application has a very general definition—client/server software that is connected by Internet technologies to route the data it processes. By “Internet technologies,” I mean the collection of hardware and software that comprises the network infrastructure between consumers and providers of information. Web applications can be made accessible by specific client software or by one or more related Web pages that are logically grouped for a specific productive purpose. That purpose can be one of any number of things, for example, to buy books, to process stock orders, or to simply exist as content to be read by the consumer.

Notice that our discussion is about Web applications, not just “Web sites.” In truth, the difference between the two is essential to understanding one of the key themes of this book. Most nonengineers do not make a distinction between a Web site and a Web application. Regardless of the term, it’s the thing that allows them to buy their books online, to make plane reservations, to purchase tickets, and so forth.

If you’re an engineer, however, there is a difference. For you, it’s likely that when someone talks about, say, the performance of a Web site, you start thinking of back-end details. And so do I. Your mind begins to consider if it’s running an Apache or IIS and whether it works using Java servlets, PHP, or CGI-bin Perl scripts. This difference in thinking between engineers and nonengineers could be confusing. Engineers, by habit, tend to associate “Web site” with the server side. However, as we all know, there is more to a Web application than just the server side; there’s the network and the client. So, based on just that, a Web site (server) is not the same thing as a Web application (the client, network, and server).

While this book emphasizes server-side solutions, it is also concerned with client-side and networking topics because they have a fundamental impact on how end users perceive Web applications. That is, we will be concerned with the end-to-end interaction with a Web site, which simply means from client to server and back to client. This is a reasonable focus. After all, most people who use the Web are concerned with its end-to-end behavior. If it takes them a while to buy concert tickets online, it doesn’t matter if the problem is caused by a slow modem, an overtaxed server, or network congestion. Whatever the reason(s), the effect is the same—a slow application that’s eating up time. As engineers, we are concerned not only that such applications might be slow for one user, but also that the system becomes slower as more users access it.

Now that we have a better fix on the scope of a Web application, let us review its core components. These are the major pieces of any online application and each represents an opportunity—a problem or a challenge, depending on how you look at it. Although you’re probably familiar with the components, it doesn’t hurt to make sure everyone is on the same page, especially since these terms appear throughout the book. Let’s start with the client side.
We will say that Web applications are used by consumers via client software (i.e., Web browsers or applications that use the Web to retrieve or process data) running on client hardware (i.e., PCs, PDAs). Application data is provided and processing is handled by producers via server software (i.e., Web server, server-side component software, database) running on server hardware (i.e., high-end multiprocessor systems, clusters, etc.). Connecting the client to the server (from the modem or network port on the client device to the networking equipment on the server side) is the networking infrastructure. Figure 1–1 shows the client/server relationship graphically. Notice that the server side is bigger; in general, we assume that the server side has more resources at its disposal.

At this point, it is important to distinguish one piece of server software, the Web server, because it nearly always plays a central role in brokering communication (HTTP traffic) between client and server. In this book, when I refer to the “server side,” I am nearly always including the Web server. When it is necessary to distinguish it from the rest of the software on the server side, I will do so explicitly.

**The Nature of the Web and Its Challenges**

Although Web applications have rapidly made the Internet a productive medium, the nature of the Internet poses many engineering puzzles. Even the most basic of challenges—engineering how a provider can quickly and reliably deliver information to all who want it—is neither simple nor well understood. Like other challenges, this problem’s complexity has to do with the nature of the medium. The Internet is
different from the information-sharing paradigms of radio, television, and newspapers for several reasons. Perhaps two of the most important reasons are its incredibly wide audience (unpredictable number of customers) and the potential at any time for that wide audience to request information from any given provider (unpredictable work demands).

Unlike in other media, Internet information providers simply do not have the ability to know their audience in advance. Newspapers, for example, know their circulation before they print each edition. They also have the advantage of being able to control their growth, making sure they have enough employees to deliver the paper daily, and have the resources and time to go from deadline on the previous night to delivery on the next morning. Furthermore, newspapers do not have to deal with sudden jumps in circulation. Compared to the Internet, the growth of big newspapers in metropolitan areas seems far more gradual. For example, when the Washington Post was founded in 1877, it had a circulation of 10,000. By 1998, that circulation had reached nearly 800,000 for its daily edition and more than that for its Sunday edition.* That’s an average growth rate of just over 6,500 subscribers per year, or 17 per day.

Deployers of Web applications have a love/hate relationship with their growth rates. In one sense, they would love the gradual growth of 17 new users per day. How nice life would be if you had to worry about scaling at that rate! You could finally go home at 5 P.M., not 9:30 P.M. At the same time, such growth rates are the reason that people are so excited about Web applications—because you can potentially reach the whole world in a matter of seconds. Your growth rate out of the gate could be hundreds of thousands of users. Although this bodes well for the business side of the things, it creates a tremendous challenge in terms of dealing with such demands.

On the Internet, circulation is akin to page hits, that is, the number of requests for a given document. Page hits can jump wildly overnight. A favorite example in the Web-caching community is the popularity of the online distribution of the Starr report. As most Americans know, this report was put together by the Office of the Independent Counsel during the Clinton administration. Let us just say that, while it was not flattering by any means, it was eagerly awaited by both the American public and the international press corps.

When the Starr report was released online in the summer of 1998 at government Web sites, tens of thousands of people tried to download it. A representative for Sprint, Inc., one of the Internet’s backbone providers, reported a surge in bandwidth demand that ranged between 10 and 20 percent above normal; a representative of AOL reported an “immediate 30 percent spike”; and NetRatings, a Nielsen-like Internet content popularity company, estimated that at one point, more than

one in five Web users was requesting the report or news about it. CNET.COM ran a number of stories about the event and its ramifications for Internet scalability in the Fall of 1998.*

The conclusion among network administrators and engineers was universal. There were simply too many requests to be handled at once, and the distribution mechanisms were unable to scale to demand. It was a real test of the scalability of the Internet itself. Not only were the Web servers that provided this information overloaded, but the networking infrastructure connecting consumers to providers became heavily congested and severely inefficient. The effect was much like that of a traffic jam on a freeway.

This phenomenon was unique because it demonstrated the effects of sudden popularity as well as the short-lived nature of that popularity. For example, it is unlikely that you or anyone else remembers the URL(s) where the report was first available. And it is unlikely that you have it bookmarked. Thus, even had those sites been able to accommodate the demands of the time by buying bigger and faster machines, it would likely have been money wasted because the need for those resources dropped dramatically after the public lost interest in the report.

Other media, such as radio and television, are broadcast and do not need to worry about the size of their audience affecting their ability to deliver information. Consider television or radio programs, such as the national and local news. Their programmers know in advance when they are scheduled to broadcast. They have the luxury of being able to prepare ahead of time. Even when live radio or television beckons, the fact that both media are broadcast means that there is only one audience to address. Cable companies and good old TV antennae are already in place to facilitate the transport of that information. If we all watch or listen to the same channel, we all see or hear the same program. This is not the case with Internet audiences, where it is usually impossible to prepare for every request, where every consumer of information requires a unique response, and where there is a continual need for new virtual links (HTTP connections) between consumer and provider to be both created and then destroyed.

**Performance and Scalability**

Have you ever gone to a Web site, clicked on a link, and really waited for a response? Of course you have; we all have. It’s annoying and frustrating. Worst are those content-laden sites that are meant to be read like newspapers. You want to jump from link to link, but every time you click, you have to wait seconds (not milliseconds) for

the page and the ads and the embedded applets to download. You almost begin to hate clicking on a link because you know you will have to wait. You’ve learned to classify this kind of site as slow.

Then there are sites that are suspiciously slow. In these cases, you have reason to believe that bazillions of people are trying to connect, and this mass, not the technology, is responsible for the slowness. Say you’re ordering a book at a site that has just announced a 50%-off sale. Or suppose tickets for a really hot concert have just gone on sale. When you’re able to connect, the site seems unresponsive. When it does respond, it crawls. You guess that the site is buckling under the demand caused by the event. You’ve learned to classify this kind of site as not scalable.

As users, we have learned what poor performance and scalability are because we have experienced them. As engineers, we would like to understand these faults better so that our own users don’t experience them. Because that is the focus of this book, let’s start our discussion of performance and scalability by defining our terms.

**Performance**

Performance can be described simply as the raw speed of your application in terms of a single user. How long does a single application-level operation take? How long does it take to search for a book? How long does it take to confirm an online registration once we click Confirm? How long does it take to check out and pay at an online store? Notice that some of these examples describe atomic operations and some don’t. When describing performance, we have to be clear if we are talking about one application operation or an entire session.

Consider the user interaction required to buy an airline ticket in Figure 1–2: In this session, there are three application operations, each consisting of a roundtrip between client and server. The operations are listed in Table 1–1 with their code names.

When we are talking about the performance of an operation, such as selection, we are interested in the end-to-end time required to complete that operation. In other words, the clock starts ticking when the user clicks the button and stops ticking when the user sees the information delivered. Why all this focus on end-to-end performance?

<table>
<thead>
<tr>
<th><strong>Table 1–1: Application Operations</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Code Name</strong></td>
</tr>
<tr>
<td>Search</td>
</tr>
<tr>
<td>Selection</td>
</tr>
<tr>
<td>Confirmation</td>
</tr>
</tbody>
</table>
We could, of course, judge performance by measuring the speed of the Web server's response, of the network, of our database retrievals, and so on. But we know that all of these performance marks are irrelevant when compared to the overall time for a logical operation. Although unit performance numbers make us happy or proud (especially if we designed that piece of the application!), end-to-end performance is the one that really counts—this is the metric that either scares users off or wins their loyalty. And thus, this is the one that can spell life or death for your application.

Addressing end-to-end performance means making operations faster for the user. To do that, we can improve the unit performance of some of the components involved in the operation(s). For example, we can improve the performance of the Web server, the database, or the application servers. The exact solution (e.g., better algorithms, more efficient queries, etc.) depends on the unit being tuned. The point is that measuring performance should be a top-down process: Start with the user, move to the components, and then to parts in the components. Look for trends and ask if a single instance of poor performance can be traced to a larger, general problem.

**Scalability**

Informally, engineers describe the challenge of dealing with large audiences and high demand as a problem of **scalability**. More specifically, we say that a Web application can scale if it continues to be available and functional at consistent speeds as the number of users and requests continues to grow, even to very high numbers. A
provider’s inability to deliver a document, such as the Starr report, because of server overload was thus a problem of scalability. Note that this definition has nothing to do with performance. As long as a slow application continues to provide consistent performance in the wake of rising demand, it is classified as scalable!

Now, although scalability is commonly defined strictly as a measurement of resiliency under ever-increasing user load, nobody expects a single instance of an application server on a single machine to accommodate millions of users. Often people consider how well an application can “scale up” by describing how effective it is to add resources, such as more CPUs, more memory, or more disks. An application is considered to scale up well if it requires additional resources at a low rate. For example, if we need to add 300MB RAM per 10 concurrent users on our system, we are in trouble. As I discuss later, this scale-up attribute is often represented as a cost, for example, a cost per concurrent transaction.

Generally, three techniques can be employed to improve scalability:

- Increase the resources (bigger machines, more disk, more memory).
- Improve the software.
- Increase the resources and improve the software.

Although the long-term answer is the third technique, our bias is toward the second. Good design at the beginning of a project is the most cost-effective way to improve scalability. No doubt you will need greater resources to deal with higher demands, but this is never the whole story. Although it can take the purchaser part of the distance, throwing money at the problem cannot ensure scalability. I don’t deny the need to spend money at certain points in the process. Rather, I suggest strategic places to spend and strategic opportunities during the design that can give application designers the biggest bang for their buck, thereby reducing their need to purchase more resources than necessary.

The Internet Medium

Six attributes of the Internet as a medium compound the challenge of delivering performance and scalability. The better we understand and appreciate these attributes, the more strategic we can be in meeting the challenge to build Web applications that perform and scale well.

First, as mentioned earlier, there is potentially a wide audience for Web application providers to manage—wider than in any other medium. Second, the Web is an interactive medium: Consumers not only receive information, they also submit it. Third,
the Internet is *dynamic* in the sense that a given user request does not always result in the same server-side response. Fourth, the Internet as a utility is *always on* and providers have no guarantees about when and how often their information will be accessed. Fifth, providing information over the Internet is an *integrated* process that often depends on the coordination of multiple provider subsystems to deliver information. And sixth, providers *lack complete control* in terms of the delivery of information to consumers: There are many networking elements that exist between provider and consumer, most of which are not controlled by the provider.

Some of these attributes may seem obvious; some may not. In either case, thinking about the details and their implications will prepare you for the solutions part of this book.

**Wide Audience**

I’m not going to beat you over the head with the fact that millions of people use the Internet every day. That is obvious and the increasing numbers are the primary reason that application architects worry about things like scalability in the first place. However, I will inform you of a few things that you may not know—or just may not appreciate, yet.

One is that there is another Internet “audience” to consider, one that is not often addressed. This quieter, hidden, but rapidly growing group of Web clients are better known as “bots.” If you are familiar with search engine technology, you already know that search engines use automated softbots to “spider” (recursively traverse) the Web and update search engine indices. This process has been going on since search engines were first deployed; bots are a simple example of one type of *information agent*.

Today’s bots are just the tip of the iceberg. More sophisticated information agents are just around the corner that will allow users to monitor multiple sites continuously and automatically. For example, instead of using the Web interactively to watch and participate in online auctions (like those at eBay and Yahoo), users will configure information agents to watch continuously and bid automatically. This is an inevitable and obvious future direction for the Web: People want to do more than sit around watching their monitors all day, manually hunting for information.

Bots and information agents are particularly fond of things like data feeds, which are information sources that continually change and require monitoring. When the Internet was first being commercialized, it was popular to connect real-time data feeds (such as the newswire services) and build access methods to them via Web applications. This trend shows no sign of slowing; in fact, it threatens to become much greater as Web applications gradually become data feeds in themselves.
I've avoided boring, albeit frightening, statistics about the growing number of human Internet users. Instead, I've reminded you that there are and will be new types of application clients, not just those with two eyes. An increasing number of information agents will automate Web querying and a growing trend will be to treat Web applications like data feeds. In short, the Web's audience is definitely growing, not to mention changing, and so are its demands. What's more, this newer audience is persistent and regular, and does not mind testing the 24x7 feature of the Web and its applications!

**Interactive**

On the Internet, consumers query providers for information. Unlike in other media, information is not distributed at the whim of the provider. Instead, consumers request information via queries, which consist of a series of interactions between the client and server.

In addition to querying, consumer requests can contain submitted information that must be processed. This submission mechanism can be explicit or implicit. Explicit submission is the user's deliberate transmission of information to the provider, such as a completed HTML form. In contrast, implicit submission is the provision of data through the user's Web session. Cookies are a good example of this, in that they consist of data that is chosen by either the provider (e.g., for page tracking) or the consumer (e.g., for personalization).

Regardless of how the information is submitted, the application's processing must often be based on this information. Thus, the Internet is not simply a library where clients request items that exist on shelves; rather, requests involve calculations or processing, sometimes leading to a unique result. Furthermore, the interactive nature of the Web means that a request cannot be fulfilled in advance—instead, the application must respond at the time the request is made, even though substantial processing may be associated with that request.

**Dynamic**

Web applications present information that depends on data associated with the user or session. As far as the user goes, countless demographic and historical session attributes can make a difference in how an application responds. The response may also depend on things unrelated to the user, such as a temporal variable (e.g., the season or the day or the week) or some other external real-time data (e.g., the current number of houses for sale). In any case, the data being generated by a Web application is often dynamic and a function based on user and/or session information.
The main problem that a dynamic Web application creates for the designer is the inability to use the results of prior work. For example, if you use a Web application to search for a house online, searching with the same criteria one week from the date of the first search may very well return different results. Of course, this is not always the case. If you conduct the same house search 10 minutes after the first one, you will very likely get the same results both times. Obviously, the designer must know when it is safe to reuse results and when it is not.

There is a subtle relationship between interactive and dynamic application behavior. To avoid confusion, keep the following in mind: Interactivity has to do with the Web application executing in response to a user, whereas dynamism has to do with the response being a product of the user, her response, or some temporal or external variable. Thus, dynamic behavior is the more general notion: An application response is the product of a set of variables, some user-specified, some not. Interactivity is simply one means to achieve a dynamic response. Put another way, interactivity describes a cause; dynamism describes an effect.

**Always On**

This Internet is never supposed to sleep. Banks advertise Web banking 24 hours a day, 7 days a week. This 24x7 mentality is part of what makes the Internet so enticing for users. People naturally assume that, at any time, it exists as an available resource. However nice this feature is for users, it is equally daunting for Web application designers. A good example of what can happen when an application is not available 24x7 is the trouble users had with eBay, the online auctioneer, in late 1999 and 2000.

During various system or software upgrades over that time, eBay suffered intermittent problems that made it unavailable to users. In June of 1999, it was unavailable for 22 hours. Since the purpose of eBay’s service is to manage millions of time-limited auctions, its core business was directly affected. Instead of selling to the highest bidder, some sellers were forced to sell to the “only bidder.” Users complained, demanding a reduction in fees. The problems made the news, and the company was forced to issue apologies in addition to refunding some fees. This is not to say that eBay is not a scalable service or that the system is always unstable; indeed, eBay is one of the most trafficked sites on the Internet, and except in rare instances, has done a tremendous amount of successful 24x7 processing.

However, this simple example does underscore the importance of 24x7 when it comes to Web applications. Nobody will write news stories about how well you perform 24x7 service, but they will definitely take you to task for glitches when you don’t. These problems can affect your whole company, especially if part of its revenue comes via the Web.
Observant readers might argue that failure to provide 24x7 service is not a question of scalability but of reliability. True, the inability to provide service because of a system failure is a question of reliability and robustness. From the practical standpoint of the user, however, it does not matter. Whether the application is unavailable because of a power problem with the site’s Internet service provider (as was the case in one of eBay’s outages) or because the system can’t handle a million simultaneous users, the result is the same: The application is unavailable.

**Integrated**

When consumers request information, providers often refer to multiple local and remote sources to integrate several pieces of information in their responses. For example, if you use the Internet to make an airline reservation, it is common for multiple systems (some of which are not directly connected to the Internet) to be indirectly involved in the processing of your reservation. The “confirmation code” you receive when the reservation is made comes only after all steps of the transaction have been completed successfully.

Integration on the server side is common for most Web applications. To some extent, this is a medium-term problem. The Web is a young technology and most of its important processing still involves some legacy or intermediate proprietary systems. These systems have proved reliable and have seemed scalable. Certainly, they are still part of the loop because organizations believe in their ability to handle workloads, but the question is whether these systems are ready for Internet-level scale.

Consider an airline that migrates its ticketing to the Web. To do so, server-side processing is required to connect to a remote, proprietary invoice database for each request. In the past, hundreds of phone-based human agents had no trouble using such a system to do processing. But it may be the case that, for example, there are some hard limits to the number of concurrent connections to this database. When there were never more than a few hundred agents, these limits were never exposed. However, putting such a system in the server-side mix may turn out to be the bottleneck in a Web application.

**Lack of Complete Control**

To a provider of information, one of the most frustrating aspects about the Web is the fact that, no matter how much money is thrown at improving application scalability, it does not mean that the application will become scalable. The culprit here is the Internet itself. While its topology of interconnected networks enables information to be delivered from anywhere to anywhere, it delivers very few quality of service (QoS)
guarantees. No matter how much time you spend tuning the client and server sides of a Web application, no authority is going to ensure that data will travel from your server to your clients at quality or priority any better than that of a student downloading MP3 files all night. And despite your best efforts, an important client that relies on a sketchy ISP with intermittent outages may deem your application slow or unreliable, though no fault of your own.

In short, the problem is decentralization. For critical Web applications, designers want complete control of the problem, but the reality is that they can almost never have it unless they circumvent the Web. This is another reminder that the solution to scalable Web applications consists of more than writing speedy server-side code. Sure, that can help, but it is by no means the whole picture.

When we talk about the lack of control over the network, we are more precisely referring to the inability to reserve bandwidth and the lack of knowledge or control over the networking elements that make up the path from client to server. Without being able to reserve bandwidth between a server and all its clients, we cannot schedule a big event that will bring in many HTTP requests and be guaranteed that they can get through. Although we can do much to widen the path in certain areas (from the server side to the ISP), we cannot widen it everywhere.

In terms of lack of knowledge about networking elements, we have to consider how clients reach servers. On the Internet, the mechanism for reaching a server from a client involves querying a series of routing tables. Without access or control over those tables, there is no way that designers can ensure high quality of service.

Techniques like Web caching and content distribution allow us to influence QoS somewhat, but they don’t provide guarantees. As it turns out, the lack of control over the underlying network represents the biggest question mark in terms of consistent application performance. We simply cannot understand or address the inefficiencies of every path by which a client connects to our application. The best we can do is design and deploy for efficiency and limit our use of the network, and thus limit performance variability, when possible.

**Measuring Performance and Scalability**

Thus far, I have defined the problem of performance and scalability in the context of Web applications, but I have not said much about their measurement. The measurement of performance and scalability is a weighty subject, and is different from the focus of this book. However, as you apply the various techniques that we cover here to your systems, you will want some simple measurements of the success of your efforts. In this section, we’ll cover a few metrics that will tell you if your application is fast and scalable.
Measuring Performance

It's fairly easy to measure performance. We can use the application being tested or we can design an automatic benchmark and observe the original speed of the application against it. Then we can make changes to the software or hardware and determine if the execution time has improved. This is a very simple approach, but by far the most common metric we will use in our study.

It is important that, when measuring performance in this way, we identify the complete path of particular application operation. That is, we have to decompose it into its parts and assign values to each. Let us return to an earlier example, that of buying airline tickets online, and imagine that we’re analyzing the performance of the “confirmation” process, which takes 2.8 seconds. Table 1–2 shows one possible set of results.

The way to read this table is to consider that completing the operation in the first (far left) column occurs at some point in time offset by the user’s click (shown in the second column) and thus some percentage of time (shown in the third column) of the end-to-end execution. Some of this requires interpretation. For example, “Web server gets request” does not mean that the single act of getting of the request is responsible for over 6 percent of the execution time. It means that 6 percent of the execution time is spent between the initial user’s click and the Web server’s getting the request; thus, 6 percent was essentially required for one-way network communication. Building these kinds of tables is useful because it allows you to focus your efforts on the bottlenecks that count. For example, in Table 1–2, we can clearly see that the database query is the bottleneck.

To build accurate tables requires two important features. One is that your system be instrumented as much as possible; that is, all components should have logging.

<table>
<thead>
<tr>
<th>Unit Action</th>
<th>Elapsed Time of Action (ms)</th>
<th>End-to-End Time (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>User clicks</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>Web server gets request</td>
<td>170</td>
<td>6.07</td>
</tr>
<tr>
<td>Servlet gets request</td>
<td>178</td>
<td>0.29</td>
</tr>
<tr>
<td>EJB server gets request</td>
<td>1.68</td>
<td></td>
</tr>
<tr>
<td>Database query starts</td>
<td>440</td>
<td>7.68</td>
</tr>
<tr>
<td>Database query ends</td>
<td>2250</td>
<td>64.64</td>
</tr>
<tr>
<td>EJB server replies</td>
<td>2280</td>
<td>1.07</td>
</tr>
<tr>
<td>Servlet replies</td>
<td>2360</td>
<td>2.86</td>
</tr>
<tr>
<td>User gets information</td>
<td>2800</td>
<td>15.71</td>
</tr>
</tbody>
</table>
features that allow them to be debugged or benchmarked. Web servers, become familiar with how these systems allow logging to be turned on and off. Make sure that you turn on logging for benchmark testing but turn it off when resuming deployment; if it’s on, logging will slow down your application. Also, your code is actually the least likely place to be instrumented. Thus, it can be good to place some well-chosen logging statements in your code. For example, if an application server makes three queries (as part of a single transaction) before replying, it would be useful to put logging statements before each query.

The second important requirement is clock synchronization. The components being measured may be on different machines and without synchronizing your clocks, you can mistakenly assess too little or too much blame to an action that is actually much faster than you thought. Exact synchronization of clocks is a bit unrealistic, but as long as you know the clocks’ relative drifts, you should be able to compensate in your calculations. Don’t overdo synchronization or calibration—for example, being off by less than a hundred milliseconds for an entire operation is not a big deal because it won’t be perceptible.

**Beyond Benchmarking**

In addition to benchmarking, there are other types of performance measurements that are well-suited to certain classes of problems. For example, suppose your Web applications are very CPU bound. To improve performance, you can add multiple processors to your system or process the problem over a cluster of workstations. Both approaches assume that it is possible to either automatically parallelize your computations or leverage explicit parallelization (i.e., thread use) and allocate parallel blocks of instructions to different CPUs/workstations. Whichever solution you choose, you will need to measure its net effect. If you don’t, then you’re shooting in the dark.

When trying to assess improvement in pure computational performance, we can measure the speedup associated with that computation. Speedup is generally defined as:

\[
\text{Speedup} = \frac{T_{\text{old}}}{T_{\text{new}}}
\]

where \(T_{\text{old}}\) is the execution time under the previous computational scenario and \(T_{\text{new}}\) is the execution time under the new scenario.

The term *scenario* is general because there are two general ways to investigate speedup: at the software level and at the hardware level. At the software level, this means changing the program code: If a program takes 10 seconds to run with the old code and 5 seconds to run with the new code, the speedup is obviously 2. At the hardware level, this means adding processors or cluster nodes. Correspondingly, for multiprocessor or cluster-based systems, the speedup metric is commonly redefined as:

\[
\text{Speedup} = \frac{T}{T_p}
\]
where $T_1$ is the execution time with one processor and $T_p$ is the execution time when the program is run on $p$ processors.

Ideally, speedup increases linearly, as processors are added to a system. In reality, however, this is never the case. All sorts of issues—processor-to-processor communication cost, program data hazards, and the like—contribute to an overall overhead of computing something on $p$ processors instead of one.

**Measuring Scalability**

Scalability is almost as easy to measure as performance is. We know that scalability refers to an application’s ability to accommodate rising resource demand gracefully, without a noticeable loss in QoS. To measure scalability, it would seem that we need to calculate how well increasing demand is handled. But how exactly do we do this?

Let’s consider a simple example. Suppose that we deploy an online banking application. One type of request that clients can make is to view recent bank transactions. Suppose that when a single client connects to the system, it takes a speedy 10 ms of server-side time to process this request. Note that network latency and other client or network issues affecting the delivery of the response will increase the end-to-end response time; for example, maybe end-to-end response time will be 1,000 ms for a single client. But, to keep our example simple, let’s consider just server-side time.

Next, suppose that 50 users simultaneously want to view their recent transactions, and that it takes an average of 500 ms of server-side time to process each of these 50 concurrent requests. Obviously, our server-side response time has slowed because of the concurrency of demands. That is to be expected.

Our next question might be: How well does our application scale? To answer this, we need some scalability metrics, such as the following:

- **Throughput**—the rate at which transactions are processed by the system
- **Resource usage**—the usage levels for the various resources involved (CPU, memory, disk, bandwidth)
- **Cost**—the price per transaction

A more detailed discussion of these and other metrics can be found in *Scaling for E-Business: Technologies, Models, Performance, and Capacity Planning* (Menasce and Almeida, 2000). Measuring resource use is fairly easy; measuring throughput and cost requires a bit more explanation.

What is the throughput in both of the cases described, with one user and with 50 users? To calculate this, we can take advantage of something called Little’s law, a simple but very useful measure that can be applied very broadly. Consider the simple
black box shown in Figure 1–3. Little’s law says that if this box contains an average of \( N \) users, and the average user spends \( R \) seconds in that box, then the throughput \( X \) of that box is roughly

\[
X = \frac{N}{R}.
\]

Little’s law can be applied to almost any device: a server, a disk, a system, or a Web application. Indeed, any system that employs a notion of input and output and that can be considered a black box is a candidate for this kind of analysis.

Armed with this knowledge, we can now apply it to our example. Specifically, we can calculate application throughput for different numbers of concurrent users. Our \( N \) will be transactions, and since \( R \) is in seconds, we will measure throughput in terms of transactions per second (tps). At the same time, let’s add some data to our banking example. Table 1–3 summarizes what we might observe, along with throughputs calculated using Little’s law. Again, keep in mind that this is just an example; I pulled these response times from thin air. Even so, they are not unreasonable.

Based on these numbers, how well does our application scale? It’s still hard to say. We can quote numbers, but do they mean anything? Not really. The problem here is that we need a comparison—something to hold up against our mythical application so we can judge how well or how poorly our example scales.

<table>
<thead>
<tr>
<th>Concurrent Users</th>
<th>Average Response Time (ms)</th>
<th>Throughput (tps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>50</td>
<td>500</td>
<td>100</td>
</tr>
<tr>
<td>100</td>
<td>1200</td>
<td>83.333</td>
</tr>
<tr>
<td>150</td>
<td>2200</td>
<td>68.182</td>
</tr>
<tr>
<td>200</td>
<td>4000</td>
<td>50</td>
</tr>
</tbody>
</table>
One good comparison is against a “linearly scalable” version of our application, by which I mean an application that continues to do exactly the same amount of work per second no matter how many clients use it. This is not to say the average response time will remain constant—no way. In fact, it will increase, but in a perfectly predictable manner. However, our throughput will remain constant. Linearly scalable applications are perfectly scalable in that their performance degrades at a constant rate directly proportional to their demands.

If our application is indeed linearly scalable, we’ll see the numbers shown in Table 1–4. Notice that our performance degrades in a constant manner: The average response time is ten times the number of concurrent users. However, our throughput is constant at 100 tps.

To understand this data better, and how we can use it in a comparison with our original mythical application results, let’s view their trends in graph form. Figure 1–4 illustrates average response time as a function of the number of concurrent users; Figure 1–5 shows throughput as a function of the number of users. These graphs also compare our results with results for an idealized system whose response time increases linearly with the number of concurrent users.

Figure 1–4 shows that our application starts to deviate from linear scalability after about 50 concurrent users. With a higher number of concurrent sessions, the line migrates toward an exponential trend. Notice that I’m drawing attention to the nature of the line, not the numbers to which the line corresponds. As we discussed earlier, scalability analysis is not the same as performance analysis; (that is, a slow application is not necessarily unable to scale). While we are interested in the average time per request from a performance standpoint, we are more interested in performance trends with higher concurrent demand, or how well an application deals with increased load, when it comes to scalability.

Figure 1–5 shows that a theoretical application should maintain a constant number of transactions per second. This makes sense: Even though our average response

<table>
<thead>
<tr>
<th>Concurrent Users</th>
<th>Average Response Time (ms)</th>
<th>Throughput (tps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>50</td>
<td>500</td>
<td>100</td>
</tr>
<tr>
<td>100</td>
<td>1000</td>
<td>100</td>
</tr>
<tr>
<td>150</td>
<td>1500</td>
<td>100</td>
</tr>
<tr>
<td>200</td>
<td>2000</td>
<td>100</td>
</tr>
</tbody>
</table>
time may increase, the amount of work done per unit time remains the same. (Think of a kitchen faucet: It is reasonable that even though it takes longer to wash 100 dishes than to wash one, the number of dishes per second should remain constant.) Notice that our mythical application becomes less productive after 50 concurrent
users. In this sense, it would be better to replicate our application and limit the number of concurrent users to 50 if we want to achieve maximum throughput.

Analyzing response time and throughput trends, as we have done here, is important for gauging the scalability of your system. Figures 1–4 and 1–5 show how to compare an application and its theoretical potential. Figure 1–4 illustrates the efficiency from the client’s point of view, where the focus is on latency; Figure 1–5 shows application efficiency from the server’s point of view, where the focus is on productivity (work done per time unit).

**Throughput and Price/Performance**

In measuring throughput, we have ignored the cost of the systems we are analyzing. If a system costing $100 can handle 1,000 transactions per second and a system costing $500,000 can handle 1,200 transactions per second, the latter obviously has better throughput—but it’s gained at a much higher cost. The idea of measuring throughput and its relationship to price is something that has been popularized by the Transaction Processing Council (TPC), which has created database benchmarks, better known as the TPC-style benchmarks.

There are three TPC benchmarks: TPC-A, TPC-B, and TPC-C. The most recently developed (as of this writing) is the TPC-C. It measures database transaction processing in terms of how efficiently it supports a mythical ordering system. Specifically, it measures how many “new order” transactions can be handled while the system is busy handling four other types of order-related transactions (payment, status, etc.). While the TPC specification is meant to measure database throughput, you can use the same principle with your systems. After all, Web application transactions are at their core a set of database transactions.

Although it is unlikely that you will benchmark your system against another, you can measure how well your system is improving or lagging over its own evolution. For example, if release 1 of your application requires $100,000 worth of hardware and software and nets 10,000 transactions per second, you can calculate a price/performance index by dividing the price by the performance:

\[
\frac{100,000}{10,000} = $10 \text{ per transaction.}
\]

This doesn’t mean that it costs $10 to execute a transaction on your system. It is simply a measure of throughput as it relates to the overall cost of the system. Suppose that a year later, release 2 of your application requires $150,000 worth of hardware and handles 40,000 transactions per second. The release 2 price/performance index would be:

\[
\frac{150,000}{40,000} = $3.75 \text{ per transaction.}
\]
Obviously, release 2 is more efficient than release 1 by evidence of its lower price/performance figure.

My interest in price/performance in this section is a reminder of the more general bias throughout this book: *Favor architectural strategies over resources when developing your application.* Once the application is deployed, you can always buy more resources to meet demand. On the other hand, rewriting code, changing designs, or re-architecting your application after deployment comes at a much higher cost. The best solution is obviously good design at the outset for scalability and performance. Not only does this eliminate the need for massive design changes after deployment, but it also typically leads to more cost-efficient resource acquisitions. CPUs, memory, disk, and other resources are purchased less frequently for applications that are inherently fast and scalable. In short, well-designed systems adapt and evolve much better than poorly designed ones do.

**Scalability and Performance Hints**

Nearly all of the chapters in this book include a section on hints for scalability and performance. The idea is to provide some conclusions or suggestions that have to do with the material presented in the chapter. Since we’ve just started our journey, there is nothing terribly complicated to conclude. However, we can remind ourselves of a few useful things covered earlier.

**Think End-to-End**

If nothing else, this chapter should have made clear that scalability and performance are end-to-end challenges. Don’t just focus on the server; consider client and network issues as well. Spending all your time optimizing your server and database is not going to help if one part of your solution doesn’t scale. You will always be hampered by your weakest link, so spend more time thinking about all of the parts involved in an application session, not just the ones you suspect or the ones you read articles about. Keep an open mind: While many applications face similar dilemmas, not all have the same clients, the same growth rate, or the same 24x7 demands.

**Scalability Doesn’t Equal Performance**

Another thing you should have gotten out of this chapter is that scalability is not the same as performance. The two have different metrics and measure distinct things.

Performance has to do with the raw speed of the application, perhaps in a vacuum where only one user is using it. When we talk about performance, we mean response time—it’s as simple as that. Optimizing performance has to do with improving the performance for that one user. If we measure average response time of 100 concurrent
users, our performance challenge is to improve the average response time of the same 100 concurrent users.

Scalability, on the other hand, has to do with the ability to accommodate increasing demand. A primary metric for scalability is throughput, which measures transactions or users per second. There is no such thing as infinite scalability—the ability to handle arbitrary demand. Every application has its limits. In fact, for many deployments it is satisfying to achieve just linear scalability, although the optimizer in all of us wants to achieve much better than that. Not unexpectedly, the most successful examples of scalability are those that simply minimize the rate at which new resources are required.

### Measure Scalability by Comparison

Scalability is difficult to ensure because its metrics don’t allow you to compare it easily to an average (nonlinearly scalable) baseline and make some conclusions. One thing you can do, however, is measure how the scalability of your application evolves. First, define what kind of throughput is reasonable: Create (or buy) an automated stress-testing system that identifies whether your current system achieves that goal for a reasonable number of users. Then, as the application evolves, periodically retest and determine if it’s improving relative to past scalability—this is without a doubt something that even your end users will notice.

Another strategy is to measure throughput as the number of users increases and identify important trends. For example, measure the throughput of your applications with 100 concurrent transactions, then with 1,000, and then with 10,000 transactions. Look at how your throughput changes and see how it compares with linear scalability. This comparison will likely give you a better sense for whether your application architecture is inherently scalable.

### Summary

In this first chapter, we focused on defining Web applications and the nature of their deployment on the Internet. We also defined and discussed performance and scalability—two important concepts that will remain our focus throughout this book—and described their related metrics.

One very important subtheme of this chapter was the focus on the entire application, not just its parts. Although it may be academically interesting to optimize our bandwidth or CPU use, the end user does not care about such things. Instead, he or she thinks only in terms of time, that is, whether the application is fast. And he or she wants that same response time regardless of how many other users are on the system at the same time. Now that we are focused on the goal of end-to-end performance and scalability, let’s move on to talk in more detail about application architectures and the specific challenges that lie ahead.
Throughout this book, we’ll discuss techniques for addressing scalability and performance in all phases of an application. Our discussion will range from the HTTP protocol to J2EE technologies, such as EJBs and Java servlets, to relational databases. Although some techniques will be relevant to only one type of technology, a few general scalability and performance strategies will permeate most, if not all, of them.

Many of these techniques were originally developed for the then-revolutionary distributed systems designed a couple of decades ago. However, they continue to be relevant today and will likely remain so for years to come. They include:

- Caching/replication
- Parallelism
- Redundancy
- Asynchrony
- Resource pooling

We’ll discuss why each is generally useful and provide real examples that demonstrate its benefits.

Caching and Replication

A cache is a structure that contains a copy of information that originates at some source. Generally speaking, a cache consists of a table of key/value pairs. For each pair, a key represents a question and the corresponding value represents an answer.
Table 4–1: Sample California City/County Cache

<table>
<thead>
<tr>
<th>City</th>
<th>County</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pleasanton</td>
<td>Alameda</td>
</tr>
<tr>
<td>Santa Clara</td>
<td>Santa Clara</td>
</tr>
<tr>
<td>Livermore</td>
<td>Alameda</td>
</tr>
<tr>
<td>Palo Alto</td>
<td>Santa Clara</td>
</tr>
<tr>
<td>Marina del Rey</td>
<td>Los Angeles</td>
</tr>
<tr>
<td>Garden Grove</td>
<td>Orange</td>
</tr>
</tbody>
</table>

For example, to look up the California county that contains the city of Marina del Rey, we ask a question—What county is Marina del Rey located in?—and obtain an answer—Los Angeles. Table 4–1 shows this information in a sample cache of cities and counties.

The keys and values in a cache can be more complex objects and are not limited to strings; the only requirement is that the key object support some sort of equality test so that we can successfully test its membership when we query the cache.

Caches typically hold much less information than that in the originating source. For example, there may be thousands (or hundreds of thousands) of city/county pairs, even though only a fraction of them are contained in the cache. There are a variety of techniques to determine what information should be retained in the cache and what should be eliminated or “flushed.”

Generally speaking, when information not in the cache is requested, the system fetches it from its original source, returns it to the requestor, and adds it to the cache. Of course, a cache has limited size and eventually it must be decided which information to flush. Obviously, it’s desirable to keep those pieces of information that will be the most frequently queried because that will provide the best system speedup. Keep in mind, however, that I just said “will be the most frequently queried.” Since there’s no way to predict the future with 100 percent accuracy, part of the challenge in designing a caching system is identifying a good cache flush policy.

One example of such a policy is least recently used (LRU), which keeps only the most recently queried information. For example, consider the sample city cache and assume that it can store only six key/value pairs, as shown. Also, assume that the table shows the order of information access—thus, Garden Grove was the last city queried. Now, if the county for San Ramon is queried, the system fetches the answer (Alameda) from the originating source and replaces the least-recently-used pair—Pleasanton / Alameda—from the cache.
Although there are many cases where LRU ends up being the best or close-enough-to-the-best policy, this is not always the case. For example, suppose that we also keep track of how many times information is accessed. Our table might be amended as shown in Table 4–2.

As you can see, even though Pleasanton was queried 100 times, it wasn’t the object of the last five queries. Then, when San Ramon was queried, LRU forced out Pleasanton because it hadn’t been requested recently. Since the cost of accessing information from its originating source can be very high, it can be better in the long term if things like access counts are taken into consideration.

Using caches to duplicate information can improve performance if the cache is more physically proximate, naturally quicker to access, or both. The speed of light guarantees that a more proximate structure is always quicker to contact than a remote source. That’s just a rule of physics, and there is no getting around it. Network optimizations aside, if I’m in San Francisco and I have to access my e-mail, it’s going to take longer if the e-mail database is in New York than if it’s in San Francisco.

On your computer, a CPU maintains a cache so that a memory request doesn’t need to travel along the system bus to the actual static RAM chips. Similarly, Web browsers maintain a cache so that they don’t have to send a request over a network and withstand network latencies for a reply from the originating source. Figure 4–1 illustrates this similarity. In both cases, the reasoning is the same: It’s physically faster to query a more proximate source.

There’s another compelling reason for caching: better access time. If the access time of a cache is quicker than that for the same information on the originating source, the cache may be more efficient, even if it’s more remote. You see this all the time on the Web. Suppose there’s a great Web site hosted in your city but its server is slow or the network infrastructure connecting it to an Internet backbone is lightweight. A content distributor like Akamai or a publicly available cache like Google may

<table>
<thead>
<tr>
<th>City</th>
<th>County</th>
<th>Accessed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pleasanton</td>
<td>Alameda</td>
<td>100</td>
</tr>
<tr>
<td>Santa Clara</td>
<td>Santa Clara</td>
<td>1</td>
</tr>
<tr>
<td>Livermore</td>
<td>Alameda</td>
<td>1</td>
</tr>
<tr>
<td>Palo Alto</td>
<td>Santa Clara</td>
<td>1</td>
</tr>
<tr>
<td>Marina del Rey</td>
<td>Los Angeles</td>
<td>1</td>
</tr>
<tr>
<td>Garden Grove</td>
<td>Orange</td>
<td>1</td>
</tr>
</tbody>
</table>
have a copy of that information that, although remotely located (perhaps it’s even more remote than the original source), is still quicker to access because the servers are snappier and the bandwidth for clients is higher.

Then there’s what I call the cut-to-the-chase aspect of querying that makes a cache quicker to access. Suppose I want to find out the vice president of an employee’s division. I have the employee ID, but to get the VP information I need to query an EMPLOYEE table to find the department ID for that employee, query a DEPARTMENT table to find the division ID for that department, and finally query a DIVISION table to find the division information I want. In contrast, a cache can simply map employee IDs to department VP information, as shown in Figure 4–2. The effect is the same: Caches can provide faster access than the original source.

In summary, there’s a tradeoff in the caching benefit equation, which can be shown as

\[
\Delta T \text{ime (per object)} = (RT_{\text{orig}} - RT_{\text{cache}}) + (ACC_{\text{orig}} - ACC_{\text{cache}})
\]

- \(RT\) = Roundtrip time
- \(ACC\) = Access time.
Thus, response time is a function of cache proximity as well as cache efficiency, both relative to the originating source.

As the equation indicates, that’s just the performance benefit. Caching can also provide scalability benefits because it reduces demand for server resources in the forms of less bandwidth, fewer CPUs, and fewer connections to manage. By copying information to a more local or more efficient source, we delegate the rising resource demands in an effort to provide consistent performance. Generally speaking, then, caching improves both performance and scalability.

Caching is not without its problems. The most obvious among them is guaranteeing data consistency: ensuring that the cache contains the same information as the server contains and that multiple caches are kept in sync. For example, if your application maintains a main memory cache of session data (such as user information) normally stored in a database, it’s important that any changes to the data be reflected in both. It’s even worse if you have multiple caches and/or multiple databases; then you need to make sure that the data is consistent across all sources.

Clearly, this can get tricky. Suppose you have a banking application that keeps track of checking and savings accounts. For improved performance, the application
caches account data in memory as accounts are accessed. So, while Joe Smith (an
account holder) is logged in, his checking balance of $100 and savings balance of $0
are kept in a memory cache.

Let’s say that this is a write-through cache, meaning that it forwards all updates
to the database as they’re made. Now, what will happen if—at the same time that Joe
is online—the bank processes a check he’s written for $50? Check processing has
nothing do with the online application (or so it would seem), so some sort of check-
processing software updates the database with Joe’s new balance. Of course, if Joe uses
his online session to transfer money from checking to savings, there can be a serious
problem because the cache is unaware of the changes made directly to the database.

This isn’t as unlikely as you might think. Valuable data often has many interfaces
and problems like this can occur unless these interfaces share a process for interact-
ing with the database. The extent of the damage in this example may depend on the
write-through nature of the cache. Are two SQL UPDATE statements taking place,
blindly ignoring the current database values and trusting the cache alone? If not,
where’s the logic that ensures that Joe won’t have a negative balance? In any case,
either Joe will get an error or the bank will give him some money back! Obviously,
neither of these is a satisfactory solution.

The consistency problem gets even worse as we try to make our application
more scalable. Say an organization provides scalable access to its information by
deploying its application across several servers and uses load balancing to distribute
the load among Web servers and application hosts. (Figure 4–3 shows this deploy-
ment.) Now, suppose that the application uses a memory cache to retain account
information; however, as Figure 4–4 shows, since the application is distributed there
are multiple caches.

Even if the organization has somehow ensured that the cached information will
be consistent with the database, how do they ensure consistency among caches? If
two users share volatile data—load-balanced and directed to an available server—
we can encounter this type of inconsistency between caches. Figure 4–4 shows that
the first entry for each of the two host-based caches is different. To prevent this,
some additional consistency mechanism must be built in or added to the system.

In short, caching solves some problems but creates others. Inconsistency is
merely one of several risks. Another is the security or ethical issue of caching: Should
Web proxies be allowed to cache content (is that legal?)? Should you cache
usernames and passwords in memory? Probably not, unless you can guarantee
their protection. The same may be true of very personal data, such as medical
records. Back to Web caching, what happens when intermediate caches don’t obey your HTTP cache directives? Are you liable?

Still another problem is cache reliability. What happens when caches without write-through policies crash? To illustrate, let’s take a look at the Oracle high-performance database management system (DBMS).

Oracle uses multiple processes (or multiple threads, depending on platform and database version) not only to cache query results but also to implement what might be called a “delayed write-through” policy regarding data updates. It avoids writing updates immediately to disk because disk I/O is slow and because permanent updates to data files shouldn’t be made until the corresponding transaction is committed. However, Oracle still must ensure that system failure doesn’t cause transactions to be lost. To solve this problem, it logs transactions so that if the server fails all committed transactions are saved, even those in memory awaiting update on disk.
Parallelism

Conceptually, parallelism is the notion of doing more than one task at once. Many parallel machines exist and have been used for research and for processing scientific data sets, but it’s probably safe to say that the majority of Web applications don’t run on parallel infrastructures. Nevertheless, many can benefit from parallelism on several levels. The key challenge is to understand what aspects of a particular application demand parallelism and what the options are. We’ll visit techniques for increasing parallelism later in this book, but first let’s define the types of parallelism that exist.

First, there are two parts of any application where parallelism can be extracted—in hardware and in software—and each offers a variety of ways to exploit it. At the hardware level, there are architectures that, by design, are parallel. The major ones relevant today are

- *Massively parallel processors (MPP)*, which consist of processing nodes that don’t share data (i.e., shared-nothing) but compute by routing data between nodes.
- Symmetric multiprocessing (SMP) machines, which consist of multiple processors that do share the same data. Basically, any processor is free to use any data.
- Clustered computing systems, which consist of multiple computers that typically don’t share the same data but route it between computers over a network.

The trend that emerges from the list is the growing distance between processor nodes. More distant nodes mean higher latencies, but closer nodes mean increased cost and complexity. For example, programming for MPP architectures is expensive and not something just any developer can do well. MPP architectures are likely nonexistent for Web applications (although some technologies, such as those for video streaming, may use them); however, many applications do run on SMP machines and clusters.

In addition to these basic architectures, there’s the notion of on-chip parallelism, which has to do with the ability to extract parallelism from a set of instructions or concurrent processes or threads. Most CPUs today do an excellent job of exploiting instruction-level parallelism (ILP), using techniques such as multiple issue, pipelining, and speculative execution to intelligently schedule and predict low-level machine instructions.

However, as processors have become more powerful, ILP techniques have actually leveled off. For example, the deeply pipelined processors of today can schedule many instructions at once. The problem is, which ones? Studies show that branches typically occur once every five machine instructions. Branch prediction and speculative execution can be employed, but they become less relevant as the size of the pipeline increases. For example, if a deeply pipelined processor allows many concurrent instructions, CPUs are forced to engage in highly speculative behavior, making predictions based on predictions of predictions of … and so on. You get the idea: the deeper you go in a pipeline, the less valuable its contents become.

Newer architectures promote the trading of deeper pipelines for multiple pipelines and the increased use of multithreading. This trend is a direct response to the limits of ILP as the size of processor pipelines increase. The increased use of multiple threads gives rise to a new category of parallelism—thread-level parallelism (TLP). Note that the emergence of TLP does not spell the death of ILP. Instead, it is widely envisioned that hybrid processor designs, reaping the benefits from both ILP and TLP, will yield the best performance. Finding the optimal tradeoff between ILP and TLP will likely remain an important issue in processor architecture research for years to come.

You may wonder: Why did multithreading emerge and why has it become a popular means for achieving parallelism? In fact, there has always been the need for
parallelism in computing. Computer architecture research and scientific programming have generated interest in parallel architectures since the 1960s. As you probably know, a great many mathematical operations and problems naturally lend themselves to parallel computation. Matrix multiplication is a good example; it is one of many complex operations that consist of a natural set of independent internal subproblems that can be computed in parallel.

However, parallelism for mass-market, consumer-oriented software (which typically consists of less scientific computations) is a relatively new phenomenon. There are two major reasons for this. One has to do with the recent increase in computer networking. Prior to the rise of the Internet, most consumers bought shrink-wrapped software that operated on only local data (i.e., data on disk). Of course, most applications today (in fact, all Web applications) involve a network. Increased network use yields applications that tend to be I/O-bound, and can thus significantly benefit from increased parallelism.

Another reason has to do with the phenomenon of Web applications and the need for concurrent request processing. Instead of traditional (local) applications that execute only on the machine owned by the client, more recent (Web) applications contain logic that executes on a remote server. This remote server often needs to handle hundreds, if not thousands, of simultaneous requests. Since each request is independent, there is great interest in parallelizing as much of this request processing as possible.

Advances in programming languages have also made threads more attractive. The Java phenomenon is probably the primary example. Only recently, programmers who could write threaded code were few and far between. Java changed all that by making threads accessible, easy to manage, and relatively safe. The result: more threaded software, which demands more TLP.

When you use Java threads for parallelism, you should carefully assess their value against the cost of the overhead involved. This isn’t to say that Java threads are expensive—they aren’t. It can be more expensive not to create them. However, it’s important to understand when to use them—specifically, the conditions in which they tend to improve efficiency.

Let’s see how two threads compare to one for two very different tasks. The first task will be a CPU-bound activity. Suppose an application server needs to increment multiple counters 10,000,000 times each. Is it faster to increment them by proceeding sequentially, incrementing each counter 10,000,000 times, or by proceeding in parallel? Of course, there may be hundreds of concurrent transactions that a given application server processes—as you’ll see, however, the effects of parallel versus sequential approaches becomes clear in the early going.
To answer our counting question, we can write two Java programs, one that counts sequentially and one that counts in parallel. The results are shown in Figure 4–5. Clearly, sequential counting is more efficient.

Now consider the task of fetching a page over the network. Suppose we need to fetch a set of 200 K Web pages. Is it faster to fetch one page at a time or fetch in parallel? Again, we can write two Java programs: one that fetches each copy sequentially and one that performs all fetches in parallel using threads. The results are shown in Figure 4–6. In short, parallel fetching is more efficient. Thus, a simple rule of thumb is to use threads when

- Your processing is at least somewhat I/O bound. For a single processor, multiple threads don’t yield extra parallelism if you’re entirely CPU bound (and you may adversely affect performance).

- You have multiple processors or an architecture that encourages TLP.

Especially for Web applications, it’s rare that all processing is CPU bound. After all, some of the time will be spent accessing remote functionality, remote databases, and the like. Thus, threads often make sense. However, if your application does engage in serious computation, don’t overdo it—or avoid writing multithreaded
code altogether. The overhead may very well not be worth it. Also, if you have the flexibility of choosing which processes (i.e., application servers) to run on which machine, try to mix I/O- and CPU-bound processes so that you get better efficiency.

**Redundancy**

Redundancy typically refers to the duplication of hardware or software so that more resources are available for execution. For example, it’s possible to have redundant Web servers or redundant disk drives, such as RAID. With either, the redundancy not only increases the ability of a system to scale, it also increases reliability—if one of several Web servers crashes, a Web site remains accessible.

Note that redundancy, as used here, is not the same as replication. The former refers to the duplication of resources, the latter refers to the duplication of data.

There can be a performance benefit to redundancy, depending on the resource involved. Consider a Web server farm. By replicating the number of Web servers, we can effectively increase the parallelism of request processing. We can achieve near-linear speedups for environments that normally have very busy Web servers, which is why server farms are such a popular scalability solution.
The only real drawbacks to redundancy are its deployment cost and the data consistency challenge. Cost increases not only for the duplicate resources but for the added hardware/software to employ them. Redundant Web servers, for example, require not only more machines to run them but a load balancer or redirector to control workload distribution. Dealing with this extra cost is simply a matter of money: Either you have it or you don’t. While throwing money at a Web application is generally not the best way to improve scalability, this is one case where it literally pays off.

Data consistency can be a more complex issue. As we discussed previously in greater detail in the section on caching, maintaining consistency requires careful planning and synchronization techniques. This can be such an arduous and vexing problem that, in some cases, it demonstrates how the costs of redundancy can outweigh its gains.

To see how, let’s return to our example of caching user account information, this time with redundant write-through caches. Initially, the caches were used to reduce the time to access the database—accessing a cache located in the same address space is faster than communicating with another process (the database server). However, as Figure 4-4 showed, if we have redundant caches on different machines, we have to update cache B whenever cache A is updated (in addition to eventually updating the database).

In short, the round-trip time we saved by not having to access the database server is erased by the round-trip time required for redundant cache synchronization. And, as the number of redundant caches increases, the problem obviously gets worse. Essentially, when we create a redundant cache we cause the access time to balloon, transforming simple updates into completed transactions.

The trick with redundancy is knowing when its costs outweigh its benefits. In almost all cases where redundancy is successful, the hardware or software is stateless. This is why redundant Web serving works so well: HTTP is by nature stateless, and thus Web servers don’t need to cache anything and can be reproduced at will. Incidentally, Web serving here refers to resolving static pages, not application request. As we’ll see in later chapters, certain application technologies can cause this stateless deployment strategy to become stateful and thus limit the redundancy of certain application components.

Remember, *redundancy* refers to “providing additional independent instances” of hardware or software, whereas *replication/caching* refers to data “copying.” This distinction is important and underscores how replication affects redundancy. For example, in revisiting the data consistency issue previously discussed, we see that it isn’t the redundancy that causes problems so much as it’s the replication of data.
Asynchrony

Dictionaries define the word *synchronous* to mean “happening at the same time.” For software and hardware systems, *asynchronous* has a different (but related) meaning: “happening independently of one another.”

We frequently encounter instances of synchrony throughout computer science. For example, clocks are said to be synchronous if they keep the same time (more specifically, if neither moves ahead of the other). Communication is said to be synchronous if one process makes a request of another and waits for a reply. In this sense, the exchange is considered synchronous because events happen in a dependent manner—a reply always follows a request. Similarly, code execution is synchronous: The next instruction comes after the previous one terminates. In each of these cases, synchrony is necessary to ensure reasonable activity; for example, if code really did execute out of order, we would get unpredictable results.

Synchronous communication is a part of almost all distributed computing systems. Web applications, being distributed, are inherently synchronous when the user is involved. When you use your Web browser to download a Web page, the download is a synchronous process: An HTTP request is issued and an HTTP response follows shortly thereafter. In most server-side application systems, servers contact each other synchronously (for example, using RMI, CORBA, or COM) as well as the database. All of this makes sense given that queries must be answered for the application to continue.

However, while synchronous behavior may be necessary in certain scenarios, it’s *never desirable* from a performance or scalability standpoint because it effectively serializes execution. Synchronous communication means that a requestor is idle while its reply is being constructed. A called function or an instruction must complete before execution proceeds.

In many situations, asynchronous methods may be applicable and can dramatically improve parallelism and thus performance. For example, suppose you develop a Web application that broadcasts live pitch-by-pitch baseball coverage in which the user downloads some graphical client that shows the play by play. Under a synchronous model, the deploying Web site would have to keep track of all clients and contact them individually—waiting for their replies—whenever a pitch was thrown. But this seems inefficient. Since client replies don’t really mean anything, why waste server-side time waiting for them? If it were possible to asynchronously communicate or broadcast this information to the clients, server-side performance would improve substantially. For example, the server could write the updated data to some

*Although modern CPUs actually support out-of-order execution, committal of results remains in order.*
Asynchrony doesn’t apply only to communication between remote pieces of functionality. If single-threaded synchronous execution is recast as multithreaded, asynchronous execution within a given program can yield important efficiency benefits. Consider a Web application that downloads a set of URLs, and inserts their contents into a database \( db \).

A simple way to accomplish this task would involve the following pseudo-code:

```plaintext
for each url \( u \) in the list of URLs to be downloaded
    page ← FETCH-PAGE(\( u \))
    DATABASE-INSERT(\( db \), page)
```

This means that, after the first download, each successive download occurs only after the DATABASE-INSERT of the previous one. It seems like a waste. We know that fetching the page is an I/O-bound activity, while database insertion is generally CPU-bound. It would seem optimal to request the next page be downloaded while the database insertion of the previous one is taking place. Thus, both activities would be occurring in parallel. Now, the only thing one needs to do is identify a way for data (i.e., the downloaded page) to be passed between them.

For this type of parallel, asynchronous solution, we might create two threads—one for downloading and one for database insertion. In doing so, suppose that the threads are able to communicate over a common thread-safe queue \( q \). Given these assumptions, consider sample pseudo-code related to the first thread, which is in charge of downloading the content:

```plaintext
for each url \( u \) in the list of URLs to be downloaded
    page ← FETCH-PAGE(\( u \))
    ENQUEUE(\( q \), page)
    ENQUEUE(\( q \), End-Of-Stream)
```

Next, consider sample pseudo-code related to the second thread, which is in charge of inserting the extracted information \( i \) into a database \( d \) as it becomes available:

```plaintext
object ← DEQUEUE(\( q \))
while object ≠ End-Of-Stream
    DATABASE-INSERT(\( db \), object)
    object ← DEQUEUE(\( q \))
```
Thus, by allowing multiple threads to communicate asynchronously via a global data structure, we increase overall processing parallelism.

In this example, idle CPU cycles were put to more efficient use at the expense of a slightly more complex implementation. When applicable, asynchronous designs such as this perform well and are easier to scale because of the more efficient use of resources. The main challenge is in implementation—writing correct multithreaded code and ensuring serialization (when necessary) for certain important operations on shared data structures. Later in Chapter 9, we’ll also discuss how technologies like the Java Message Service (JMS) make distributed asynchronous solutions very easy.

### Resource Pooling

Popular Web applications have to deal with hundreds or thousands of requests in parallel, many of which rely on application and database resources. The overhead of providing access to these resources can balloon quickly as parallelism demands rise. Consider highly concurrent database access. Each client that wants to use the database will require a database connection. Each connection requires the overhead of memory to store its state and also the overhead to create and initialize it. When you’re talking about hundreds or thousands of database clients, this overhead can become unmanageable.

One very common technique to reduce overhead is resource pooling—in this case, pooling database connections. The idea here is that it’s cheaper to create a fixed pool of connections that are shared among many clients than to create one connection per client. By cheaper, we’re talking about the cost of memory and the cost of thread initialization. Does resource pooling actually work and provide these benefits? Let’s see.

Suppose we have some number $r$ of concurrent application server requests and that each request makes five JDBC calls. We’re interested in the throughput of requests as their number increases. Our example has two types of application server: one that uses a connection for each of the five queries made per request and one that borrows from a connection pool whenever it makes a JDBC query. For the sake of simplicity, our pool is small—say 10 connections.

The code for our example is shown in Listings 4–1 through 4–5 that follow. The five important classes are: the client test (`PoolTest.java`), our connection pool (`ConnPool.java`), the shared connection (`SharedConnection.java`), the JDBC query executor (`RequestRunnable.java`), and finally the barrier for test synchronization (`Barrier.java`). Because this isn’t a book about Java or synchronization, we’ll skip an exhaustive discussion of the code. Instead, we’ll simply describe the overall flow and identify selected interesting parts. Here’s the general flow:
- PoolTest.java takes in client input on the number of concurrent accesses to simulate (i.e., `java PoolTest 100`) and creates the proper number of threads.
- Barrier.Java makes sure that we start all of the threads at once—this may seem a bit unfair if we don’t make sure all threads are initialized before starting the test.
- RequestRunnable.java simulates each client request and deploys it as a thread.
- Each RequestRunnable performs five JDBC queries when it runs.
- To access the database, all RequestRunnable instances draw from the same connection pool (ConnPool), which was initialized in PoolTest.java.
- RequestRunnable objects “loan out” a SharedConnection object (they don’t own it) and return it when they’re done with each query.

**Listing 4–1: Class PoolTest.java**

```java
1 import java.sql.*;
2
3 /* Connection pool test client */
4
5 public class PoolTest
6 {
7   public static void main(String[] args) throws Exception
8   {
9     /* Register JDBC driver - Oracle used for example */
10     Class.forName("oracle.jdbc.driver.OracleDriver");
11
12     /* Identify number of concurrent threads for test */
13     int numThreads = args.length > 0 ?
14       Integer.parseInt(args[0]) : 1;
15
16     /* Initialize connection pool */
17     ConnPool p = new ConnPool(10);
18
19     /* Initialize and setup thread barrier for test */
20     Barrier b = new Barrier(numThreads);
21     for (int i=0; i<numThreads; i++)
22       (new Thread(new RequestRunnable(i, p, b))).start();
23
24     /* Let all threads attempt to execute at once */
25     b.release();
26   }
27 }```

Class RequestRunnable.java

```java
import java.sql.*;

/* Simulates a single request that requires 5 JDBC queries */

public class RequestRunnable
   implements Runnable
{
   private ConnPool m_pool;
   private Barrier m_barrier;
   private int m_id;

   public RequestRunnable(int a_id, ConnPool a_pool, Barrier a_barrier)
   {
     m_id = a_id;
     m_barrier = a_barrier;
     m_pool = a_pool;
   }

   public void run()
   {
     m_barrier.enter();

     /* Run 5 queries */

     PreparedStatement ops;
     ResultSet rset;

     for (int i=0; i<5; i++)
     {
       try {
         Connection oconn = m_pool.loanConn();
         ops = oconn.prepareStatement(
           "SELECT count(*) FROM EMPLOYEE");
         rset = ops.executeQuery();
         m_pool.returnConn(oconn);
       }
       catch (Exception e) {
         System.err.println("ERROR during querying.");
       }
     }
   }
}
```

Scalability and Performance Techniques
Listing 4-3: Class ConnPool.java

```java
1 import java.sql.*;
2
3 /* A very simple connection pool class */
4
5 public class ConnPool
6 {
7   private SharedConnection[] m_list;
8
9   public ConnPool(int num) {
10     m_list = new SharedConnection[num];
11     try {
12       for (int i=0; i<num; i++) {
13         m_list[i] = new SharedConnection(
14           DriverManager.getConnection(
15           "jdbc:oracle:thin:@mydb")
16       }
17     }
18     catch (Exception e) {
19        System.err.println("Error when allocating connections.");
20        System.exit(1);
21     }
22
23   }
24
25   /* Distribute a connection if/when we have one */
26
27   public synchronized Connection loanConn() {
28     Connection conn = null;
29
30     while (true) {
31       for (int i=0; i<m_list.length; i++) {
32         if (!m_list[i].inUse()) {
33           m_list[i].markBusy();
34           conn = m_list[i].getConn();
35           break;
36         }
37       }
38       if (conn !=null)
39         break;
40     try {
41       wait();
42     }
43     catch (Exception e) {
44        System.err.println("Error when waiting for a connection.");
45        System.exit(1);
46     }
47
48   return conn;
49 }
```
public synchronized void returnConn(Connection a_conn) {
    for (int i=0; i<m_list.length; i++) {
        if (m_list[i].getConn() == a_conn) {
            m_list[i].markAvailable();
            notify();
        }
    }
}

Listing 4–4: Class SharedConnection.java

import java.sql.*;

/* Same as a normal database connection except that it is shared */

public class SharedConnection {
    private Connection m_conn;
    private boolean m_inUse;

    public SharedConnection(Connection a_conn) {
        m_conn = a_conn;
        markAvailable();
    }

    public Connection getConn() { return m_conn; }

    /* Keeps track of shared status */
    public synchronized void markAvailable() { m_inUse = false; }
    public synchronized void markBusy() { m_inUse = true; }
    public synchronized boolean inUse() { return m_inUse; }
}

Listing 4–5: Class Barrier.java

/* A very simple barrier class */

public class Barrier {
    /* Local class for synchronization bookkeeping */

    private class Marker {
        private boolean mLocked = true;
        ...
    }
}
public synchronized void setDone() {
    m_locked = false; notify();
}
public synchronized void waitDone() {
    if (m_locked) try { wait(); } catch(Exception e) { }
}

int m_num;
Marker m_marker;

public Barrier(int a_num) {
    m_marker = new Marker();
    m_num = a_num;
}

/* Add a thread to the barrier holding tank */
public synchronized void enter() {
    m_num--;
    try {
        if (m_num == 0)
            m_marker.setDone();
        wait();
    } catch (Exception e) {
        System.err.println("Error when entering barrier.");
        System.exit(1);
    }
}

/* Notify all listeners */
public void release() {
    m_marker.waitDone();
    synchronized (this) {
        notifyAll();
    }
}

Executing the code demonstrates the effect of connection pooling. Figure 4–7 shows how long it takes under both approaches (connection pool and no connection pool) to process various numbers of concurrent client requests.
In this chapter, we have focused on some of the most important general techniques for ensuring scalability and performance in Web applications. In fact, these techniques and concepts also apply to distributed systems in general, of which the Internet is but one. No matter how the Internet evolves in terms of specific new technologies (e.g., Web services), its continued existence as a distributed system ensures that the material covered here will continue to be relevant.

To recap, here are some of the key points of this chapter:

- **Caching** and data replication involve copying data for the sake of improving performance. Such techniques work because they reduce the latency between the requestor and the data (i.e., data locality is increased), they summarize the translation of request to response (fewer operations in between), or both. Although both caching and replication are useful techniques, the issue of data consistency sometimes makes their implementation difficult.

- **Parallelism** increases the amount of work done at one time. In an environment where processing power and bandwidth are unlimited, parallel execution can dramatically improve performance. In most practical environments, however, local resources are limited and it is simply not feasible to parallelize everything. Doing so could exhaust resources or create a situation where a great deal of time is spent context switching and communicating among multiple threads or processes.
processes. Nevertheless, there is a middle ground here. Understanding where your application is I/O-bound and where it is CPU-bound enables you to identify the best opportunities for parallel execution and to design accordingly.

- **Redundancy** enables better application scalability. As the number of concurrent requests rises, redundant software or hardware architectures allow those requests to be attended to without noticeable degradation of service. Well-designed redundant systems make very high scalability achievable by making it easy for operations staff to deploy new machines or new software instances to meet increased demand. One important key to a successful architecture for redundancy is the effectiveness related to balancing the request or processing load.

- **Asynchrony** encourages parallel execution; thus, it can be considered complementary to such architectures. Typically, asynchronous solutions decouple the execution of one component from the execution of another. Instead of having a consumer wait for the producer to finish producing everything, effective asynchronous solutions allow the producer to stream results to the consumer and to have the latter start executing (in parallel) as soon as possible.

- **Resource pooling** bounds the overhead required to serve concurrent demand to specific resources, such as a database. By pooling objects such as connections or threads, the cost to create each instance is bounded by the maximum size of the pool. This allows concurrent execution without skyrocketing overhead costs. However, the obvious challenge is to size the pool correctly (or to grow it effectively) and to reduce the synchronization overhead required to manage the pool (i.e., issue and reclaim resource instances).

Effective use of all the techniques can lead to very efficient systems. The key is to understand the demands on your application, and then to choose your battles accordingly during the design phase. Fortunately, as we saw in Chapter 3 and as we will see in greater detail in future chapters, J2EE embraces these techniques and makes them available to application deployments as part of an underlying platform. By understanding how such features are offered by J2EE and how they can be leveraged, you can build applications that are generally scalable and encourage high performance.
CHAPTER 12

CAPACITY PLANNING FOR THE WEBLOGIC SERVER

Excerpted from: *J2EE Applications and BEA WebLogic Server (2nd Edition)* by Angela Yochem, David Carlson, Tad Stephens
building the application, the next step is to many registered users should your system support? How many requests per second should it be able to handle? What sort of network and hardware infrastructure is required to meet your deployment goals? This chapter explains both the methodology used in capacity planning and performance information available. We cover the following:

- **WebLogic JRockit**—We review the optimized, server-side Java Virtual Machine (JVM) packaged with WebLogic Server 8.1 and WebLogic Platform 8.1. JRockit provides a scalable, reliable JVM for Intel-based machines.
- **Analysis**—We discuss factors to consider for capacity planning.
- **Metrics**—We set a baseline set of capacity-planning numbers derived from an existing application (we use our sample application, WebAuction) you can use in determining the infrastructure required to meet your deployment requirements.
- **Review**—We resent capacity planning best practices.

**WebLogic JRockit**

**Note:** This section is a review of the JRockit section in Chapter 14. Skip ahead to “Analysis of Capacity Planning” if a review of JRockit is not necessary for you.

The vision behind Java is to provide platform independence for applications. It allows you to write the application once and deploy it on other hardware platforms, without requiring costly code modifications. In its infancy, this platform neutrality came at a price in raw execution speed. However, with time this price has grown smaller and smaller. Products like WebLogic JRockit 8.1 provide a scalable, reliable, high-performing execution engine for deploying enterprise scale applications.
WebLogic JRockit is a Java Virtual Machine (JVM) optimized for server-side applications and for inter-process scalability. BEA designed JRockit specifically for Intel environments running Windows and Linux operating systems. It supports both 32-bit and 64-bit Intel architectures, fully utilizing the additional address space available. Java applications are reliant on the underlying virtual machine and perform based on how well the JVM handles code generation, garbage collection, memory and thread management, and handling of native methods. Originally designed in 1998, WebLogic JRockit delivers an optimized virtual machine designed to exploit the available hardware resources, using adaptive code generation, advanced garbage collection, optimized code execution, and a robust management framework for profiling and tuning.

Compilation Options

JRockit provides two means for optimizing the compiled Java code into native execution. The first mechanism is a Just-In-Time compiler that compiles during server startup. Server-side applications rely on fast execution; however, most JVMs are designed for faster startup and only interpret the Java byte code, waiting to compile into native code until they are requested. JRockit precompiles every class as it loads via the JIT compiler. Although start times are somewhat longer, this cost is amortized over time and the overall result is much faster execution times.

The second area where JRockit provides superior performance is in the optimizing compiler. While compiling all classes with all available optimizations would take too long during server startup, WebLogic JRockit identifies the frequently used methods and runs these through a second, optimizing compiler. By fully optimizing only those functions that will deliver maximum application performance, JRockit provides optimal application performance while still maintaining acceptable server startup times.

How does JRockit recognize what functions warrant the optimized compilation? There is a sampling thread that periodically “wakes up” and checks the status of the various threads executing within the application. The information is recorded and, based on internal heuristics, JRockit determines which threads or functions should be earmarked for optimization. Many functions are never optimized for the life of the server. Typically, these optimizations are done shortly after the server starts, with fewer and fewer optimizations taking place the longer the server runs.

Garbage Collection

A second area of optimization for WebLogic JRockit is in garbage collection. Java applications allocate memory on a heap and when the heap is full it must be harvested, freeing up stale objects and other memory no longer in use. The process of harvesting the heap is known as garbage collection. While a server is garbage collecting, the application performance is impacted, since the virtual machine cannot dedicate itself to execution only. You can increase the size of the heap, allowing more objects to be allocated and garbage collection to occur less frequently, but the price you pay is in longer garbage collection times. A tuned garbage collection model will allow optimal heap usage while minimizing the impact of the garbage collection on application performance. Lastly, the application design will have a significant impact on the most optimal garbage collection scheme, garbage collection that performs well for one application may negatively affect performance in another.

To address these needs, WebLogic JRockit supports four different garbage collectors:

- Generational Copying—In this model the memory is separated into two or more areas called “generations”. Instead of allocating all objects into a single space and garbage collecting it all
at once, objects are allocated into the new generation called the "nursery". This model is best suited to small heap sizes on single CPU machines.

- Single Spaced Concurrent—This collector uses the entire heap and removes pauses completely by executing concurrently with application processing. While the application is executing a background thread is harvesting the heap. This model is best suited for garbage collection without disrupting server execution, however, there is a risk if the garbage collection thread cannot harvest the heap as fast as the application places new objects on it.

- Generational Concurrent—This collector combines the approaches of the first two. New objects are allocated to the new generation, which is garbage collected by "stopping the world" and pausing execution. Active objects in the new generation are moved to a separate generation called the old generation. Garbage collection of the old generation follows the concurrent model, with a background thread harvesting concurrently with execution and not causing execution pauses.

- Parallel—In this model all garbage collection are performed in "stop the world" mode, pausing all execution while garbage collection takes place. Multiple threads are used for the harvesting, and although there are execution pauses this model maximizes throughput and memory utilization.

By default, JRockit uses the generational concurrent collection model. The garbage collection model right for your application depends on a number of variables. The ideal model is to minimize the impact of garbage collection, determined by the number of times garbage collection is run and the duration of each run. However, if the application has constraints around response time and cannot tolerate execution pauses, one of the concurrent models may the best choice.

Configuring the garbage collector is done by setting the `-Xgc <collector type>` parameter when starting the JVM. Valid collector types are `gencopy`, `singlecon`, `gencon`, and `parallel`. In addition, you can set the initial and maximum heap sizes using the flags `-Xms <heap size>` and `-Xmx <heap size>` parameters. When using the generational models, you can also control the size of the new generation using the `-Xns <heap size>` parameter. For example, to start JRockit using the generational concurrent collector with an initial and maximum heap size of 512MB and a new generation size of 256MB you would use the following command:

```
java -jrockit -Xgc:gencon -Xms512m -Xmx512m -Xns256m ...
```

Tuning the garbage collection with WebLogic JRockit can deliver optimal performance for your application. However, it is a good idea to consult BEA’s documentation before making changes to any of the JVM parameters, as these changes can also result in penalizing your application performance as well. Refer to http://e-docs.bea.com/wljrockit/docs81/tuning/index.html for more information on tuning JRockit.

**JVM Management and Monitoring**

A key capability in tuning your application for optimal performance is identifying the behavior occurring in the underlying JVM. WebLogic JRockit includes useful tools for profiling and tuning the JVM performance. JRockit includes a Management Console for observing real-time information about server and resource utilization. This console can be run stand-alone, or accessed using the WebLogic Server administration console. Through information presented in the console, you can identify bottlenecks to performance and change operating and environment variables to deliver
maximum performance. Figure 17-1 illustrates the JRockit graphical console accessed from the WebLogic Server console.

![JRockit Management Console](image)

**Figure 17–1**  
JRockit Management Console

JRockit 8.1 also supports the JVM Profiling Interface (JVMPI) and the JVM Debugging Interface (JVMDI), allowing Java applications to interact with the JVM and assist in profiling and debugging activities. This is particularly useful when using third-party tools designed to comply with these standard interfaces.

### Analysis of Capacity Planning

Now that we reviewed a powerful component in the WebLogic suite, let’s consider how to plan our applications for scalability. The art of determining requirements for a WebLogic Server deployment and planning an infrastructure that will support those requirements is collectively referred to as capacity planning. Capacity planning is an attempt to determine the resources, such as CPUs, Internet connection size, and LAN infrastructure, required to support performance. Capacity planning answers the question, What hardware infrastructure and network configuration will enable my WebLogic deployment to fulfill specified performance requirements? Ideally, with an architecture
that will scale in a consistent fashion, you can use capacity planning to determine how much of these resources will be required to support the volume of requests as your processing needs grow. A truly scalable infrastructure will not introduce performance bottlenecks as the application scales, putting off the dreaded “point of diminishing returns.”

Capacity planning is an inexact science because there are so many factors that influence the capacity of a given application deployment, including the following:

- How database-intensive is the application?
- How large are the pages that Java Server Pages (JSPs) display?
- What is the typical usage pattern for a user?
- How do users access the system?
- How fast is the underlying hardware?

In fact, so many factors influence capacity planning that it borders on being an art form. Nonetheless, it is possible to take some of the guesswork out of the process of capacity planning. In reality, the most influential factor on the hardware capacity necessary to support your application is the application itself. For this reason, this chapter aims only to provide information that will assist your own capacity testing and application design efforts. Creating “a one size fits all” formula to compute the amount of hardware required for a given application is a futile effort.

Factors Affecting Capacity

There are three major areas for focusing capacity-planning efforts for a WebLogic deployment:

- **Server hardware**—Obviously, the capacity of the servers where WebLogic runs directly affects the capacity of the WebLogic deployment. For example, every JSP request from a client requires both memory and CPU time to generate a response from WebLogic. You need to assess the number and power of CPUs, RAM size, JVM efficiency, and other factors relating to the server hardware platform: How much capacity is required from your server platforms?

- **LAN infrastructure**—As noted in previous chapters, a WebLogic cluster relies on a LAN for communication between cluster nodes. Depending upon the application, the requirements for a LAN can vary. For example, a large cluster that is doing in-memory replication for either Enterprise JavaBeans (EJBs) or servlet-session replication requires a higher bandwidth network than a small cluster. In addition, the size of session data, the size of a cluster, and the power of the server machines affect the requirements for the LAN infrastructure. You need to assess cluster network hardware performance: How much capacity is required from the LAN between the nodes of the WebLogic cluster?

- **External network connectivity**—The WebLogic deployment communicates to other resources such as databases or legacy systems or communicates externally to systems such as the Internet. You must assess the frequency of connections to external systems and the size of data being transferred. How much capacity is required from the network that connects the WebLogic cluster, the clients, and back-end resources?

This chapter focuses on each of these three areas independently. Capacity planning should be applied across all the components of a WebLogic deployment, not just the servers that run WebLogic. All the components in the deployment affect the capacity of a WebLogic deployment.
Methodology and Metrics for Capacity Planning

Capacity planning is focused on how the WebLogic deployment deals with maximum performance requirements. What is the peak load that your deployment will be able to handle? In other words, what is its maximum capacity for handling requests? Capacity-planning methodology focuses on the worst-case scenario, such as when your company’s advertisement appears on the Super Bowl and your deployment suddenly receives a flood of millions of new requests. How does an online brokerage service handle the volumes of requests when investors settle their positions at the end of the day? The underlying assumption of planning for the worst case is that your WebLogic Servers and the infrastructure should be able to scale up to that peak load.

In order to quantify the worst cases, set goals or measurable objectives for capacity: for example, the WebLogic deployment should be able to handle 10,000 open user sessions at a given time.

**Best Practice:** The application deployment should have distinct capacity goals that quantify the maximum capacity required for the deployment. When you have measurable objectives, you can design the WebLogic deployment to meet your needs and better understand the performance characteristics.

Setting Capacity Goals

The first step in capacity planning is to set goals for the deployment. These goals should be quantified as maximums:

- **User interactions per second with WebLogic.** This value represents the total number of user interactions that should be handled per second by a WebLogic deployment. User interactions are typically accesses to JSP pages or servlets for Web deployments. For application deployments, user interactions are accesses to EJBs.

- **Total number of concurrent user sessions.** This value represents the total number of user sessions that WebLogic should handle at a given time. Concurrent user sessions are mostly an issue for Web deployments when WebLogic is maintaining HTTP session objects around for each user. However, concurrency measures are also important when application deployments access stateful session EJBs. Remember that the number of concurrent user sessions isn’t just the users currently accessing the system but also includes the lifetime of current sessions. Long session lifetimes result in a greater number of concurrent user sessions.

- **Storage capacity for user information.** This value represents the capacity required to store user information. In the simplest case, this value is the disk and memory required to store security information for each user. User-related storage is not covered in this chapter because it is either trivial or because it directly depends on external systems such as databases. In the trivial case where the WebLogic-based security realm or the database is used for storage of user information, simply multiply the size of each user’s information by the total number of users. If each user requires 1MB of storage, then 20 registered users require 20MB; 1,000 users require 1GB, and so on.

To illustrate how capacity planning works for these three basic deployment characteristics, let’s come up with some requirements for deploying the WebAuction application. The capacity goal for the WebAuction application is 800 user requests per second, or 69,120,000 requests per day. Note that goals are stated in terms of maximums, the worst-case possibility for capacity.
Server Hardware Capacity Planning

Many usage-related factors affect the capacity of a deployed application:

- Client protocol
- Security profile
- Degree of platform optimization for running Java and WebLogic

This section covers each factor, detailing how each factor affects the overall capacity of the WebLogic deployment.

**Client Protocol**

The client protocol is directly related to the type of WebLogic deployment. Application deployments and mixed deployments of WebLogic generally rely heavily on the Remote Method Invocation (RMI) programming model to access WebLogic services.

RMI can rely on the native WebLogic T3 protocol or can use HTTP tunneling to allow the RMI calls to pass through a firewall. Performance of RMI tunneled over HTTP is typically worse than that of non-tunneled T3.

Application clients can use HTTP by directly making HTTP POSTs and GETs to access servlets. In these cases, the application client should be treated as a Web browser client generating HTTP requests.

**Security Profile**

The level of security that is put in place between clients and WebLogic is a factor in determining the capacity of the deployment. WebLogic supports SSL (Secure Sockets Layer) as the security mechanism to ensure privacy and to authenticate users. SSL protects JSP pages for credit card purchases and bank statements, ensuring that attackers cannot view sensitive information.

SSL is a very intensive computing operation. The overhead of SSL cryptography means that WebLogic Server can handle fewer simultaneous connections than in a system without SSL.

You should note the total number of SSL connections required, over time, to determine your average client load. Typically, the server can handle three non-SSL connections for every one SSL connection. Given that users need not use an SSL connection for every request, SSL reduces the capacity of the server substantially. The amount of overhead incurred from SSL is directly related to how many client interactions use it.

**Clustering Profile**

The clustering profile affects the capacity of the WebLogic deployment. Two factors in clustering affect capacity:

1. Cluster size
2. Usage pattern of in-memory replication

We are assuming that in-memory replication will be used in the cluster architecture because of the scalability advantages it provides. For more information on in-memory replication, refer to the section on clustering in Chapter 12, "WebLogic Administration."

The cluster size directly influences how much network traffic is required to support the cluster. Various categories of traffic flow over the LAN that connects the WebLogic nodes in the cluster. The nodes in the cluster can coordinate some of this traffic, but a larger cluster requires a higher power network to communicate efficiently.
In-memory replication of session information is usually the largest consumer of LAN network bandwidth in a WebLogic cluster. As you recall from previous chapters, both servlets and session EJBs can replicate their session information across the cluster. This provides a hot backup in the case of failure of a given node in the cluster.

To keep the hot backup current with the latest information, the LAN propagates changes to sessions among nodes in the cluster. The use of in-memory replication most directly affects the LAN infrastructure required to support the cluster. See the capacity-planning methodology later in this chapter.

**Application Profile**

The application profile is a summary of all of the tasks that the WebLogic application must perform in response to client requests. These include serving Web pages, handling client requests, processing forms, dealing with user sessions, opening database connections, managing connection pools, and so forth.

Applications can be simple or they can be quite complex. The WebAuction application is moderately complex, involving a number of different components, including JMS and clustering for its deployment. Clearly, it is possible to create an even more complex application with even more business logic and more complex code paths. Unfortunately, there isn’t a good metric for how the application profile can affect capacity. However, the reliance upon back-end systems, such as databases, can help us estimate the complexity of applications using WebLogic.

**Dependence on Legacy/Back-End Systems**

Most WebLogic deployments rely upon back-end systems such as databases or messaging systems on mainframes. Typically, a WebLogic application uses a database to generate Web content, such as looking up an employee’s record. However, as demonstrated in the WebAuction application, requests are made that do not rely upon a back-end system to be served. For example, the welcome page `index.jsp` in the WebAuction application makes no calls to the database or back-end systems.

Obviously, the more reliance there is on back-end systems, the more resources are consumed to meet requests. Each client request for which WebLogic acts as an intermediary to a back-end system incurs the overhead of accessing the database system.

**Note:** If your application has a high ratio of legacy system access or accesses more than one legacy system to fill client requests, you should consider your application more complex than the WebAuction application.

**Session-Based Information**

The amount of session-based information that the WebLogic deployment has to handle also directly affects capacity. The WebLogic deployment must track session objects in memory for each session. This memory is either directly available in RAM or is obtained by swapping out to disk.

**Best Practice:** The hardware must have enough memory to contain all the current session objects without swapping to disk. As session objects per user become larger or more users’ sessions must be held simultaneously, more RAM is required. This RAM is accessible to Java via the Java heap, which is where the Java environment stores all objects and application data. As the system begins swapping, performance is negatively impacted. For optimal performance, the best practice is to make sure there is sufficient memory to hold the current session objects without swapping to disk.
A Baseline Capacity Profile

This section includes baseline numbers for capacity planning using the WebAuction application on standard server hardware in a configuration that simulates a real-world deployment. In this section we use the same testing configuration used in Chapter 14, “Designing the Production Deployment.” Figure 17–2 presents the test configuration.

Two client machines generate load onto a single server running the WebAuction application. Oracle 8, running on another identical server, acts as the database. The entire configuration is connected by a 100Mb Ethernet. The WebLogic Server uses the WebLogic JRockit Java Development Kit (JDK) 1.4 for Windows 2000 included in the WebLogic distribution with a 256MB heap size, 15 connections in the connection pool to the database, and 15 execute threads.

To generate baseline numbers, a scenario was developed to provide the absolute worst performance possible. By choosing a usage case by a user that demanded the most of the application, we can be sure that any other use will result in better performance, making it a safe bet for capacity planning. Remember, the idea is to measure the amount of hardware required to support your application, and the capacity-planning effort determines the low bar for application performance.

As was determined in Chapter 13, when testing the WebAuction application, updates to the database to add new users were the most expensive operations both in terms of database and application server resources. For this reason, a scenario was created with the following user action flow for one of the client machines:

**Figure 17–2**
Testing configuration.
Chapter 17 • Capacity Planning for the WebLogic Server

1. Visit the welcome page (main.jsp).
2. Create a new user account (newuser.jsp).
3. Browse items in the books category (browseitems.jsp).
4. Bid on an item (bid.jsp).
5. Browse the user’s bids (currentbid.jsp).

Notice that each iteration through the flow results in a new user account being created. The second client machine focuses on doing read operations to simulate what users would be viewing items in the categories:

1. Visit the welcome page (main.jsp).
2. Browse items in the books category (browseitems.jsp).

Together, these two machines are simultaneously directed at the WebAuction application. The results represent what kind of performance can be expected in a real-world deployment (see Table 17–1).

Table 17–1 Results of Performance Testing With Two Machines

<table>
<thead>
<tr>
<th>CPUs</th>
<th>New Users per Second</th>
<th>New Users per Day</th>
<th>Http Requests Served per Second</th>
<th>Http Requests Served per Day</th>
<th>Server CPU Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3.97</td>
<td>342,720</td>
<td>312</td>
<td>26,956,800</td>
<td>87%</td>
</tr>
<tr>
<td>3</td>
<td>2.72</td>
<td>234,720</td>
<td>238</td>
<td>20,563,200</td>
<td>86%</td>
</tr>
<tr>
<td>2</td>
<td>2.10</td>
<td>181,440</td>
<td>168</td>
<td>14,515,200</td>
<td>90%</td>
</tr>
<tr>
<td>1</td>
<td>1.08</td>
<td>93,600</td>
<td>90</td>
<td>7,776,000</td>
<td>89%</td>
</tr>
</tbody>
</table>

These numbers provide a basis for comparison for other applications. By using clustering and in-memory session replication, you can see that WebLogic scales in a near linear fashion. Thus, by simple multiplication, we can get an idea of what sort of hardware is required to support larger configurations of WebLogic. Next, we need to see how your particular application differs from the WebAuction application.

To illustrate this process of comparing your application to the baseline capacity profile for the WebAuction application, we detail capacity planning for WebAuction in the next section.

LAN Infrastructure Capacity Planning

Now that the server hardware has been determined, we can proceed to plan the capacity of the LAN infrastructure. Fortunately, capacity planning for the LAN infrastructure is simpler than capacity planning for the server hardware. This is mainly because only one factor heavily affects the requirements on the LAN infrastructure: in-memory replication of session information for servlets and stateful session EJBs.

In-memory replication of session information is the largest consumer of LAN network bandwidth in a WebLogic cluster. As you recall from previous chapters, both servlets and session EJBs can have their session information replicated across the cluster. This provides a hot backup in the case of failure of a given node in the cluster.
To keep the hot backup current with the latest information, the LAN is used to transmit session changes to nodes in the cluster. Typically, the recommended session size is 5KB to 15KB in size. Larger sessions require higher network bandwidth to support efficient operation.

Table 17–2 summarizes the network requirements for both cluster size and in-memory replication details. Where “switched” is noted, the LAN infrastructure should be based on a switch rather than a hub, which reduces the saturation of the network under load.

Table 17–2  Network Requirements

<table>
<thead>
<tr>
<th>CPUs Running WebLogic Instances in Cluster</th>
<th>In-Memory Replication Session Size (in KB)</th>
<th>Approximate LAN Capacity Minimum (in MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2–8</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>2–8</td>
<td>&lt;15</td>
<td>100</td>
</tr>
<tr>
<td>8–16</td>
<td>&gt;15</td>
<td>Switched 100</td>
</tr>
<tr>
<td>8–16</td>
<td>&lt;15</td>
<td>Switched 100</td>
</tr>
<tr>
<td>8–16</td>
<td>&gt;15</td>
<td>Switched 1 Gbps</td>
</tr>
<tr>
<td>16+</td>
<td>0</td>
<td>Switched 100</td>
</tr>
<tr>
<td>16+</td>
<td>&lt;15</td>
<td>Switched 100</td>
</tr>
<tr>
<td>16+</td>
<td>&gt;15</td>
<td>Switched 1 Gbps</td>
</tr>
</tbody>
</table>

As you can see, as the cluster size grows, the infrastructure requirements on the network consume more and more bandwidth.

External Connectivity Capacity Planning

Connectivity to external clients and resources is also a factor in capacity planning. The amount of traffic to an external resource such as a database is, of course, application-specific. This section looks at connectivity requirements for clients to WebLogic as well as connectivity requirements to legacy/back-end systems.

Client Connectivity Requirements

A network connection is required to connect WebLogic to its clients on the Web. This network connection may be over the Internet or across the corporate network. In the case of the corporate network, bandwidth is typically very high and capacity planning is not required. However, when connections to WebLogic clients are made over the Internet, it is necessary to make sure that the bandwidth across the Internet is appropriate.

In most cases, simple calculations can be made to estimate the amount of bandwidth required to support Web users. This begins with determining the size of responses that will be sent to clients. To determine the size of these responses, you should look at your application and determine the average size of the responses. Your calculation should include both the HTML code as well as any static images that you’re serving, such as JPEG. After determining the average size of the transmission to the requesters, we can create a weighted average based on how often the various pages are served.
Let’s do this calculation for the WebAuction application. We have the following JSP pages that respond to requests (see Table 17–3).

**Table 17–3  Responses per JSP Page**

<table>
<thead>
<tr>
<th>JSP Page</th>
<th>Total Size (with Images) in Kb</th>
<th>Estimated Percent of Total Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>login.jsp</td>
<td>25</td>
<td>48%</td>
</tr>
<tr>
<td>trade.jsp</td>
<td>20</td>
<td>48%</td>
</tr>
<tr>
<td>error.jsp</td>
<td>15</td>
<td>4%</td>
</tr>
</tbody>
</table>

The weighted average of these pages is shown in Table 17–4).

**Table 17–4  Weighted Average of Responses per Page**

<table>
<thead>
<tr>
<th>JSP Page</th>
<th>Weighted Total Size (with Images) Based on the Percent of the Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>login.jsp</td>
<td>12Kb (= 25 Kb * 0.48)</td>
</tr>
<tr>
<td>trade.jsp</td>
<td>9.6Kb (= 20 Kb * 0.48)</td>
</tr>
<tr>
<td>error.jsp</td>
<td>0.6Kb (= 15 Kb * 0.04)</td>
</tr>
<tr>
<td>Weighted Average:</td>
<td>22.2Kb (= 12 Kb + 9.6 Kb + 0.6 Kb)</td>
</tr>
</tbody>
</table>

As you can see, only 4 percent of the total requests come from the error.jsp JSP page. The login and trade JSP pages account for 48 percent of the responses. In weighting these responses, we can estimate that the average response to each client request will be 22.2KB.

We can take this average response size of 22.2KB and look at how many requests per second could possibly be handled by different, standard Internet connectivity and network connectivity links (see Table 17–5).

**Table 17–5  Theoretical Maximum Number of Responses**

<table>
<thead>
<tr>
<th>Network Connectivity Speed</th>
<th>Theoretical Maximum Responses Served per Second</th>
<th>Theoretical Maximum Responses Served per Day</th>
</tr>
</thead>
<tbody>
<tr>
<td>56.6Kbit modem @ 7.075Kbps*</td>
<td>0.32</td>
<td>27,648</td>
</tr>
<tr>
<td>Digital Subscriber Line (DSL) @ 128Kbit/16Kbps</td>
<td>0.72</td>
<td>62,208</td>
</tr>
<tr>
<td>T-1/OC-1 Line @ 1.544Mbps/193Kbps</td>
<td>8.69</td>
<td>750,816</td>
</tr>
<tr>
<td>T3/OC-3 Line @ 44.736Mbps/5,592Kbps</td>
<td>251.89</td>
<td>21,763,296</td>
</tr>
<tr>
<td>OC-4 Line @ 274.176Mbps/34,272Kbps</td>
<td>1543.73</td>
<td>133,378,222</td>
</tr>
</tbody>
</table>

* 56.6k modems are measured in terms of bits per second.
This chart represents only the theoretical maximum because it does not take into account the bandwidth consumed by subsequent requests. Check with your local Internet service providers for more information on the different capacities of digital lines.

In reality, the bandwidth provided by these connections is about 50 to 75 percent of the theoretical maximum. So, a T3 line in practice can transmit 10 to 15 million 22KB pages per day. The numbers here only mean to provide a rough estimate for what one would expect in a WebLogic deployment with a typical response size. It is highly recommended that you consult your Internet service provider for the appropriate level of bandwidth for your application.

Back-End Resource Connectivity Requirements

Many factors affect connectivity requirements to resources, including how much data is transferred, how often it is transferred, and what the capacity is of the back-end resource. All of these factors vary from application to application. By default, connectivity to back-end resources is viewed in the traditional client/server model in which WebLogic is the client and the back-end resource is the server.

In the case of most legacy resources that connect to WebLogic, such as a database or a mainframe data store, the systems are mature enough to already offer recommendations for capacity planning and connectivity. For example, in the case of Oracle databases, capacity-planning information is available that includes both network connectivity and hardware server requirements. However, much of this capacity-planning information for legacy systems is based on the traditional client/server model, in which application clients connect directly to the database and therefore are not applicable.

Because WebLogic introduces a third tier to the application, in which connections are made from the client to the application server and then translated into requests for the legacy application, you must often use abstractions for capacity planning to back-end resources with WebLogic. In this abstraction, you should view the WebLogic application as a proxy for the client requests as it executes work for each individual user. So, if your WebLogic deployment is to service 5,000 requests per second that translate into database access, then you should plan the capacity of your database connectivity and database hardware to be able to handle 5,000 requests per second.

Fortunately, as the three-tier model for applications becomes more common, the database vendors also are beginning to offer capacity-planning information that is tailored to application servers. The requirement that you abstract WebLogic Server as a client to the database is only temporary until the database vendors catch up.
Chapter 17 • Capacity Planning for the WebLogic Server

Capacity Planning Best Practices

There exist a number of best practices that should be followed when capacity planning for a WebLogic deployment. The following are a few key items to be aware of.

Be Conservative with Your Capacity-Planning Estimates

The steps detailed in this chapter provide a high-level estimate of the configuration required to meet your deployment goals. Capacity planning with WebLogic is not an exact science. For this reason, it is a best practice to err on the side of caution in terms of estimates. Many successful deployments take server hardware capacity-planning estimates and increase them by 50 percent in order to be absolutely sure that they will have adequate capacity. For deployments where absolute reliability is preferable over cost savings, this is a common practice.

Load-Test Your Application

A great number of things can go wrong in terms of an application’s capacity that can never be identified until the application is deployed in practice. For this reason, you should plan to load-test your application either in a prototype form or using the hardware that you plan to deploy upon.

Optimize Your Application

The application built on top of WebLogic is often the most limiting factor to capacity. For this reason, you should plan to optimize your application during the testing process. A number of tools provide insight into hot spots and inefficiencies in your application based on WebLogic:

- jProbe from the KL Group (http://www.klgroup.com) and OptimizeIT from Intuitive Systems (http://www.optimizeit.com) can be used at development time to find bottlenecks in the code.
- Introscope from Wily Technology (http://www.wilytech.com) is a Java product that allows for runtime performance monitoring of any Java component in WebLogic. This tool is designed for monitoring production systems and not for tuning during development.

Plan for Growth

One of the major benefits of WebLogic is the ease in which the infrastructure can be scaled. Growing a cluster is as simple as incrementally adding server machines, even mixing and matching hardware server types within the same cluster. Most WebLogic deployments start small but grow substantially over time as more clients and services come online. For this reason, it is a good practice to plan your WebLogic deployment for growth. You may want to choose a LAN infrastructure that is larger than your current deployment so that it is ready when you want to grow.

In terms of external connectivity to other resources, such as the Internet or legacy systems, the bandwidth and hardware resources should be extensible. Many ISPs offer instant upgrades to Internet connections for higher bandwidth.