

12

Designing Web Applications and Servlet Patterns

In previous chapters we've seen that servlets are great at handling requests and responses, but they're not so good for generating content for end users. We also wouldn't use JSP pages to process business logic, although they do handle content generation very effectively. To make the most of servlets, or any other components within a web application, we need to understand when and where we should use them – what tasks do they perform best?

In this chapter we're going to look at the design of web applications, focusing particularly on when and where we should use servlets within the application.

We'll begin by discussing the most commonly used web application architectures:

- Model 1
- Model 2

We'll look at important factors such as the maintainability, reusability, and extensibility of each. We'll introduce **design patterns**, which we can use to recognize and solve common design problems. Then we'll demonstrate how these patterns can be applied to a web application.

In order to put all of these concepts into context we'll create an example application – an online discussion forum, to which we'll apply our designs and patterns. In order to compare and contrast good and bad design, we'll build two versions of this example: the first using Model 1 architecture, and the second using Model 2.

However, let's start simple, by considering why good application design is important.

Why is Good Application Design Important?

The design of applications is an important aspect of the project lifecycle, but one that is often neglected. When we design an application we should aim to enhance the following features:

- ❑ **Maintainability**
- ❑ **Reusability**
- ❑ **Extensibility**

Let's now take a closer look at each of these aspects of design.

Maintainability

Maintainability describes the amount of effort required to keep an application running correctly. Obviously, in a commercial sense, more maintenance means more time and more money spent. In reality the maintainability of an application is difficult to quantify, but there are techniques that we can use to increase the maintainability of a system.

So what aspects of the application need to be maintained? First of all there is the sourcecode itself. An undocumented, unstructured, and badly-formatted body of sourcecode will be harder to maintain than documented code with a clearly defined structure. If we make code easy to follow and understand, debugging and code modification usually become faster and easier too. As an example, trying to maintain a very large method that performs a lot of processing will be much more complex than trying to maintain a collection of shorter methods that achieve the same result.

The structure of the application also plays an important part. Designing a meaningful, logically partitioned application will greatly increase the maintainability of the application. In a typical application there might be several distinct parts; for example the user interface, classes that perform business processing, and classes that represent business entities. Introducing structure and making a distinction between these types of classes will increase the maintainability because it will be more apparent to developers how the pieces of the application fit together. Although creating neatly formatted sourcecode is undoubtedly important, knowing how to quickly find the piece of functionality that needs to be fixed is just as important.

Reusability

One of the goals of OO design and implementation is to strive for reusable components. Reusability is enhanced when a class has a low reliance on other objects, or **low coupling**, and provides a specialized set of tasks, or **high cohesion**.

Some degree of reusability is achievable without design, but thinking about the structure beforehand will help to ensure that the classes that we create are reusable across the application. Stepping back to look at the big picture will help in identifying those places in the system where functionality can be reused later if we wish to modify or enhance the application.

Extensibility

Changes in the business require changes in software. The extensibility of our software determines how much it can be extended and enhanced after it has been put into production use.

Ideally, we would like to add functionality to an application without altering the original code – much as we extend existing Java classes.

A good design will attempt to take extensibility into account. Of course it's impossible to foresee every eventual circumstance, such as a drastic change in the business, or other major changes in functional requirements. We can often enhance extensibility by logically partitioning the application into smaller parts, reducing the impact of changes on other parts of the system.

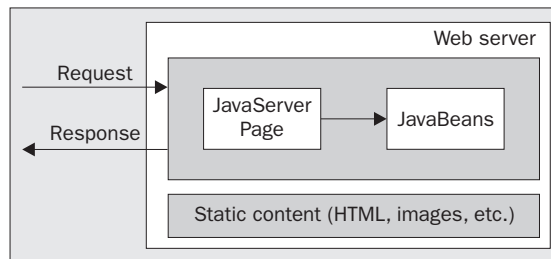
J2EE Web Application Design

Servlets are just one component that can be used in building a web application. We can also use JSP pages, JSP tag libraries, JavaBeans, and even business components such as Enterprise JavaBeans (EJBs) if we need or wish to. Each of these components has its strengths and weaknesses, so each is suited to a different role. Identifying the roles needed within the application and finding the components most suitable for these roles is the first step in understanding the design of our applications.

Two of the most commonly used web application architectures are known as **Model 1** and **Model 2**. In the next few sections we will discuss each of these, in order to understand where our different components will fit into them. Model 1 architecture is the simpler of the two, so let's consider this first.

Introducing Model 1 Architecture

A web application based on the Model 1 architecture is composed of a number of pages with which the user interacts. These pages generally utilize a **model**, which represents the business logic for the application. For example (and as shown in the diagram below), the pages could be implemented using JSP pages, with the model being one or more JavaBeans. In this situation, these beans would be used to represent information found in the application and might also contain a small amount of business logic:



In fact, the pages in Model 1 architecture are usually implemented using JSP pages (or sometimes servlets), so Model 1 architecture is known as **page-centric**.

The client directly accesses the pages served up by the web container through the web server, and these pages are used to service the entire request and send the response back. Any links to other parts of the web application are given as direct links to other pages.

Although each individual page could contain complex logic, this type of web application is very easy to assemble and is therefore very useful where there are a small number of pages, or where the structure of the application is very simple. However, problems with maintainability, extensibility, and security can arise if we attempt to use a page-centric architecture with larger or more complex web applications.

Maintainability Problems

If we consider a typical Model 1 web application, the structure of the application will be embodied within the pages. The pages will include diverse functionality, such as:

- ❑ Complex business logic
- ❑ Links to other parts of the web application
- ❑ The names of pages to which forms should be submitted

The fact that the pages often contain business logic often provides the biggest potential problem regarding the maintainability of a Model 1 architecture. It is usual to find blocks of Java code, known as **scriptlets**, in the page. This not only makes the code within the page more difficult to understand, but if we wanted to reuse the scriptlet code we would have to copy and paste it. If we wanted to incorporate the functionality into lots of pages this would be a time consuming (and error-prone) process.

Additionally, modifying the structure of our web application in any way would mean that we would need to search through the entire site to remove and/or change the names of the pages; again, a time-consuming and costly process.

Extensibility Problems

The extensibility of our web applications is also limited, as it is hard to modify or extend the functionality provided by the application since the pages contain such diverse functionality and are relatively tightly-coupled together. Changing some functionality could ripple horribly through the system causing more bugs and unexpected results.

Security Problems

Security is also a major problem for Model 1 architectures. For example, a web-based discussion forum might be divided into two distinct sections:

- ❑ Public area – for reading discussion threads and locating general information
- ❑ Restricted area – for creating threads and posting responses

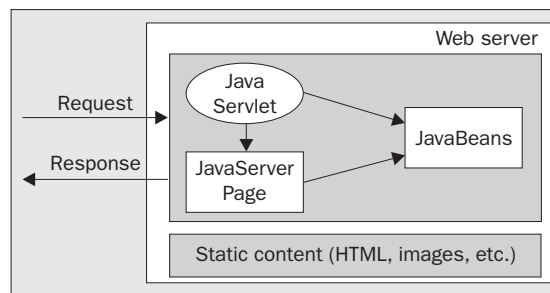
If we used Model 1 architecture to implement the forum, each page in the restricted area would have to perform its own security checks to ensure that the user is logged in and that the user is permitted to see the contents of the page.

To circumvent this problem, we could place the authorization and authentication code into reusable components such as JavaBeans or custom JSP tags, or at the simplest level another JSP that is included in a page wherever necessary. However, we still need to include the authentication and authorization code in each secure page using some approach. This invites trouble: for example developers may accidentally forget to include this security code in pages they create for the site.

A simpler approach to security would be to provide a central access point for all users of the site; this central point would contain the authentication and authorization code, so we only need to include it once in our application. We will see how we can adopt this approach using Model 2 architecture in the next section.

Introducing Model 2 Architecture

The Model 2 architecture overcomes many of the problems of the Model 1 architecture. Model 2 is based upon the **Model-View-Controller (MVC)** architecture. In a Model 1 web application, the pages are accessed directly, in a Model 2 architecture all access is routed through a **controller** component (typically implemented by a servlet). This architecture is shown below:



We can use the same controller for the entire site, or multiple controllers that are each responsible for sections of the site. For both, the principle of the design is the same: requests from clients are routed to a central controller, which decides how the request should be serviced. Later on we'll see how a controller can delegate this task but for now we'll assume that the controller processes the request.

When it processes the request, the controller may make some modifications to the underlying model (often JavaBeans). Once the request has been serviced, the controller delegates the task of sending back the response to a view component (often implemented by JSP pages). The view component generally contains a minimal amount of code and is focused on the task of presenting information to the end user. Both the controller and the view components can access and modify model components.

Benefits of Model 2 Architecture

So how does using Model 2 architecture improve the design of our web application over using Model 1?

Enhancing Maintainability

As the controller component in our architecture is responsible for determining which page in our web application we should see next, the structure of the web application is defined in a single place. This increases the maintainability of the application, particularly if we go further, separating out the structural definition from the controller to make the structure externally configurable. Additionally, the processing and business logic associated with servicing requests is easy to find as it is not embedded and scattered throughout the pages of the system (unlike in the Model 1 architecture).

Promoting Extensibility

The logic that processes each type of request in Model 2 is now much more centralized. This centralization, in conjunction with splitting the view components from the logic that services the request, means that the system is now much easier to extend than in Model 1 architecture. By introducing more componentization we make extensions easier to write and the chances of a ripple effect throughout the application much less likely.

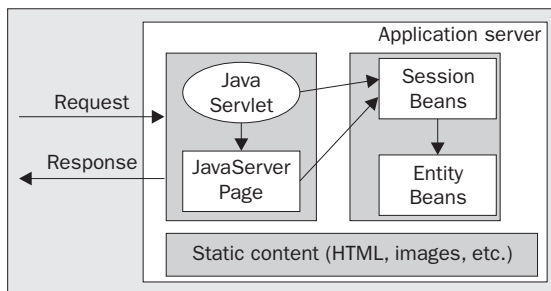
For example, to extend the application by adding new pages, all that we need to do is add new view components and make appropriate modifications to the controller component – either directly or through external configuration files.

Ensuring Security

When we discussed the security aspects of Model 1 earlier, we noted that routing all requests through a single point of entry is a simple way to handle web application security. In Model 2, we can use this technique, because all security processing can be moved to the controller(s), reducing the burden on developers and the risk of accidental security breaches.

Using Other J2EE Components in Model 2 Architecture

For many applications we use technologies other than JSP pages, servlets, and JavaBeans. We may want to introduce an EJB layer containing session and entity beans, where we place complex business processing or represent persistent business data:



While the basic architecture remains unchanged, introducing these additional components does, however, raise some questions:

- ❑ How should the functionality of our application be split between the components?
- ❑ What effect does introducing these components have on performance and scalability?

We will examine these issues more closely in the next two sections.

Using Entity Beans

Entity beans are used to represent data that is stored in some type of persistent data store such as a relational database. In smaller applications, we'd probably have the web components (our Servlets and JSPs) using JDBC to communicate with the database instead (possibly with JavaBeans representing the data).

Introducing entity beans means that they become the model on which our web application operates. For both Model 1 and Model 2 the components that handle the presentation of data on a web page (in other words the JSPs and servlets in the web container) will access and modify business data residing in the EJB tier.

From an architectural and design perspective, this separation of the components in our web application is certainly beneficial. Splitting these out makes for a more maintainable system since functionality has been logically partitioned across the application. When the entities in the business layer change, we instantly know which part of the application needs to change to reflect this. In addition, we might also end up with some components that we can reuse in future projects.

There are disadvantages to using entity beans though. From an implementation perspective, entity beans are remote components that are able to execute on a separate machine from the code that calls them. Introducing remote components means introducing additional network overhead in terms of the remote method calls that are required to access and modify the data residing within the EJBs. As always, you should weigh up the advantages and disadvantages in the context of the application(s) that will make use of entity beans before you decide to use them.

Using Session Beans

Traditionally, business logic has been embedded into client applications, whether they are desktop-based or web-based. Session beans can instead encapsulate this logic in a robust and reusable manner. However, session beans, like entity beans, are intended to be remote components and so the same network overheads may be incurred when using the functionality contained within session beans.

With these issues in mind, let's look at roles and responsibilities in a Model 2 architecture.

Component Roles in Model 2 Architecture

Let's consider the role of the controller in the Model 2 architecture example. It should:

- Act as a single point of entry for requests
- Process the requests, accessing and modifying the underlying model
- Delegate the task of presenting information to a specific view component

The controller can process requests in a number of ways. It can process requests itself which means that the controller encapsulates business logic. However, this is probably not an effective solution, as the controller would rapidly become bloated.

A more efficient system might be to delegate processing tasks to other components such as JavaBeans or session EJBs, or even standard Java classes. If this strategy is selected, how do we determine where specific functionality is to be situated, including that associated with servicing the request, validating any data, and of course, security? When we design our application, we need to answer these questions, deciding on the roles and responsibilities of components. This is often a tricky task, but fortunately many of the common problems encountered during the design process have been tackled before, and the solutions to these problems are well-documented in the form of **design patterns**.

Documenting Design Principals

Industry-recognized standards and guidelines help those involved with a project to communicate in a consistent and effective way. For example, we can use the Unified Modeling Language (UML) to represent the design of an application.

In the same way that coding conventions are important when writing sourcecode, the use of a standard means of documenting designs is also important, especially when reviews and walkthroughs are undertaken. Such documentation allows the developer to question the business and its methods by providing a platform that can be understood by non-technical personnel.

Using Design Patterns

Design patterns capture reusable solutions to common design problems, and provide a common language with which to describe them. In "*Design Patterns, Elements of Reusable Object-Oriented Software*" (Addison Wesley, ISBN 0-201633-61-2) twenty four common design problems are described and considered. Many other patterns in software development have been documented, including some directly related to J2EE development.

J2EE Patterns

The J2EE patterns catalog documents a number of patterns that provide solutions to frequently encountered problems in the design and implementation of J2EE applications. This includes issues such as decreasing the number of fine-grained method calls across the network by using value objects, encapsulating business logic and workflow into session beans through a session façade, and accessing large amounts of read-only data through data access objects. The catalog also contains patterns that are relevant when designing the presentation tier of J2EE applications.

Detailed information on all of the patterns contained within the J2EE patterns catalog can be found at <http://developer.java.sun.com/developer/technicalArticles/J2EE/patterns/>.

Why Use Patterns?

Without patterns it is still possible to build fairly complex systems that achieve the desired end goal. However, the problem with this approach is that more often than not, the way that problems are tackled and subsequently solved within the application may not be consistent. This inconsistency decreases the maintainability of the application.

Although using patterns does tend to increase the initial complexity of the implemented solution, patterns are an important tool for helping us structure our applications in a proven and consistent manner. Each component has a predefined purpose and a predefined way in which to interact with the others involved in the system. This is good from a maintainability perspective because it is easier to get to grips with a system based upon familiar, common structures than one that isn't.

In addition to this, some of the patterns are optimized for high performance and scalability. Another important benefit of using patterns is that they also provide developers with a standard way of communicating design ideas.

Creating a Web-Based Discussion Forum

In this section, we'll implement a small web application to illustrate the concepts of good and bad design we've been discussing. We'll create a discussion forum, as seen on many web sites. Before we can start designing our forum we need to define its features. This is an important stage in the development process, and it's important to fully state the requirements of our application:

- ❑ The discussion forum contains a number of topics
- ❑ Each topic may contain zero or more responses
- ❑ A user can view the list of topics, and also view all the responses made to a topic
- ❑ A user can post a response to a topic, but to ensure that responses are not anonymous, the user must be logged in
- ❑ We'll allow the users to delete their own responses too

This is a fairly simple set of requirements, so building the application shouldn't be too difficult. However, before we start implementing the functionality, we need to understand the underlying domain on which the application will operate.

Entities Within the Business Domain

An important aspect of object-orientated (OO) analysis and design is to identify the entities that are present in the application's domain. For our discussion forum we have three entities:

- ❑ **Topic** – represents a topic on the discussion forum
- ❑ **Response** – a response to a topic on a discussion forum, of which there may be zero or more
- ❑ **User** – the author of a topic or response

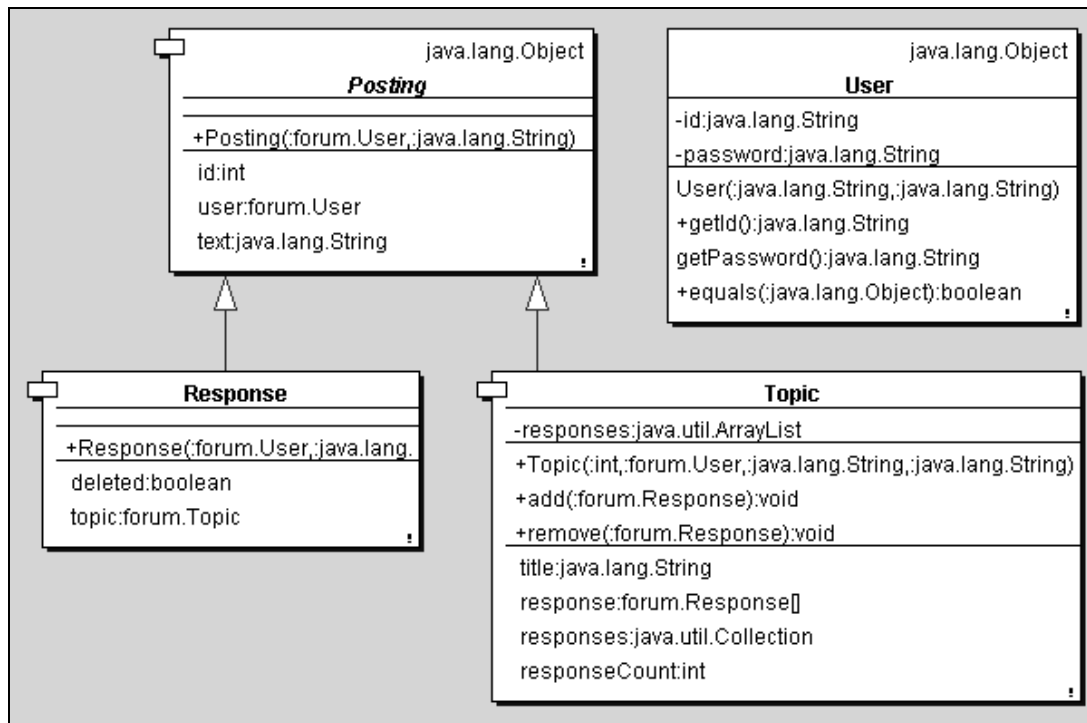
Our next step is to model the entities and the relationships between them using UML. This allows us to represent the domain clearly and concisely, so that we can communicate it effectively to both developers and non-developers within the team.

You can learn more about UML in "UML Distilled", 2nd edition (Addison Wesley, ISBN 0-201657-83-X), or "Instant UML" (Wrox Press, ISBN 1-861000-87-1).

The Class Model for the Discussion Forum

From our model of the domain and through a process of steps beyond the scope of this book, we eventually arrive at a design for the classes in our system that will represent the entities within our business domain. This could be a simple process such as picking out the nouns in a requirements specification, or by looking at the entities present in a UML analysis model of the business.

The classes required for our application map almost exactly onto the entities discovered in the business domain, and this is often the case in real-world projects. Here is the class diagram:



As you can see, we have `Topic`, `Response` and `User` classes that map to our entities, and we also have a `Posting`. This is an abstraction of both `Topic` and `Response` and represents the commonality between them. In this case, both topics and responses have some text (the message) and a user associated with them.

The User Class

The `User` class in this example is just a wrapper for a user `id` and a `password`. In the future, however, we might want to modify it to contain further information, so a reference to the `User` object is stored in the `Posting` rather than just the username. The complete `User` class is as follows:

```

package forum;

public class User {

    private String id;
    private String password;

    User(String id, String password) {
        this.id = id;
        this.password = password;
    }
}
  
```

```
public String getId() {
    return this.id;
}

String getPassword() {
    return this.password;
}

public boolean equals(Object o) {
    if (o instanceof User) {
        User u = (User)o;
        return getId().equals(u.getId());
    } else {
        return false;
    }
}
}
```

The Posting Class

A Posting object is an abstraction of both a topic and a response. Here's the class:

```
package forum;

public abstract class Posting {

    protected int id;
    protected String text;
    protected User user;

    public Posting(User user, String text) {
        this.user = user;
        this.text = text;
    }

    public int getId() {
        return this.id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public User getUser() {
        return this.user;
    }

    public String getText() {
        return this.text;
    }
}
```

As mentioned, topics and responses have an id, some text (the message) and an associated User. In a similar way to the User class, Posting is again a wrapper around this basic information.

The Topic Class

The `Topic` class is a little more than a simple wrapper this time. It extends `Posting` and therefore inherits all of the basic information illustrated previously. In addition to this, `Topic` introduces a `title` and a collection of `responses` that have been made.

`Topic` also introduces some functionality – the ability to add a response, remove a response, get a response(s), and finally get a count of the number of responses that have been added so far.

Many users would use a real-world discussion forum concurrently. For this reason, we would need to ensure that data is not corrupted by synchronizing access to the data. However, for simplicity's sake, this example does not include such synchronization code.

```
package forum;

import java.util.*;

public class Topic extends Posting {

    private String title;
    private ArrayList responses = new ArrayList();

    public Topic(int id, User user, String title, String text) {
        super(user, text);

        this.id = id;
        this.title = title;
    }

    public String getTitle() {
        return this.title;
    }

    public void add(Response response) {
        responses.add(response);

        response.setId(responses.size() - 1);
        response.setTopic(this);
    }

    public void remove(Response response) {
        response.setDeleted(true);
    }

    public Response getResponse(int id) {
        return (Response)responses.get(id);
    }

    public Collection getResponses() {
        return responses;
    }
}
```

The `getResponseCount()` method gets the number of (non-deleted) responses that have been made to this topic:

```
public int getResponseCount() {
    int count = 0;
    Response response;

    Iterator it = responses.iterator();
    while (it.hasNext()) {
        response = (Response)it.next();
```

We don't include those responses that have been deleted:

```
        if (!response.isDeleted()) {
            count++;
        }
    }
    return count;
}
```

The Response Class

Finally we have the `Response` class. Again this extends `Posting`, this time adding a reference to the parent `Topic` and a flag to indicate whether the response has been deleted:

```
package forum;

public class Response extends Posting {

    private Topic topic;

    private boolean deleted = false;

    public Response(User user, String text) {
        super(user, text);
    }

    public boolean isDeleted() {
        return this.deleted;
    }

    public void setDeleted(boolean b) {
        this.deleted = b;
    }

    public void setTopic(Topic topic) {
        this.topic = topic;
    }

    public Topic getTopic() {
        return this.topic;
    }
}
```

Building the Forum Using Model 1 Architecture

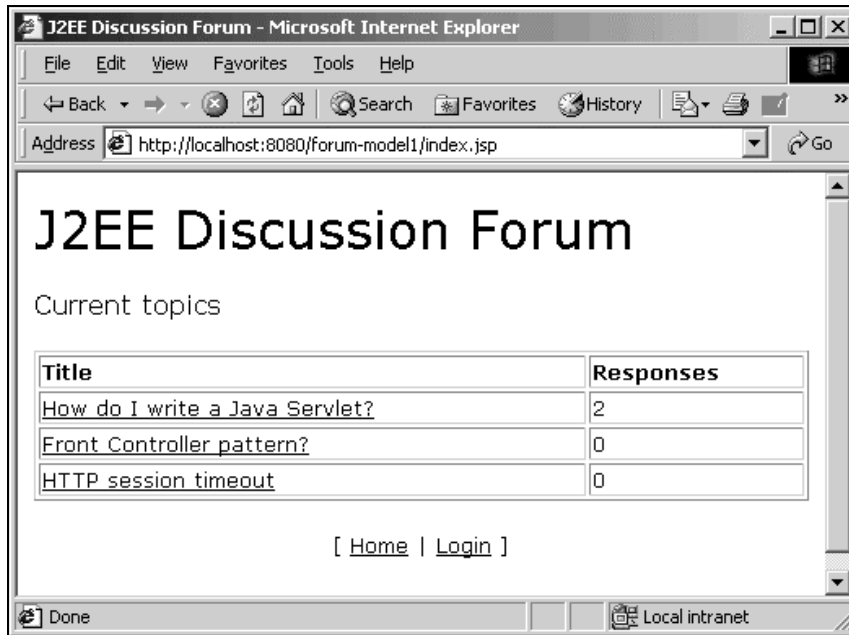
As our discussion forum is only simple, we'll just go straight ahead and build it using a Model 1 page-centric architecture.

The presentation and logic of our application will be constructed from a number of JSP pages. We've already designed the underlying representation of our domain model so all we have to do now is build the user interface. With this in mind, we'll develop JSP pages that make use of our `Topic`, `Response`, and `User` classes.

Although we are building an application based upon the Model 1 architecture, we could in fact use servlets to make use of the classes that we have just defined. However, because the majority of the code associated with building these pages is presentational, using JSP pages makes this easier. The same results could be achieved with servlets, but in this case the code would become complex and cluttered with statements simply printing the HTML back to the client.

Viewing the Topics

The first page that we need to build is the home page (`index.jsp`), which shows a list of the topics that are currently in our system. Here's what the page looks like in the browser:



We've created links for each topic title so that the user can navigate to another page showing the responses that have been made so far. In terms of how this has been implemented, the table of topics is generated by some Java code embedded inside a JSP scriptlet. Our JSP page looks up the topics that are currently known by the system and then iterates over them. We make use of a class called `Topics`, which represents the set of sample data for this example. In a real system the topics would probably be stored in a database.

You should note that, for simplicity, surrounding HTML tags (such as <head> and <body>), Java import statements, and so on, have been omitted in the code below.

Here's the relevant code from `index.jsp`:

```
<h2>Current topics</h2>

<p>
<table width="100%" border="1">

  <tr>
    <td><b>Title</b></td>
    <td><b>Responses</b></td>
  </tr>

  <%
    Collection topics = Topics.getTopics ();
    Iterator it = topics.iterator();
    Topic topic;
    int id;

    while (it.hasNext()) {
      topic = (Topic)it.next();
      id = topic.getId();
    }
  <%>

  <tr>
    <td>
```

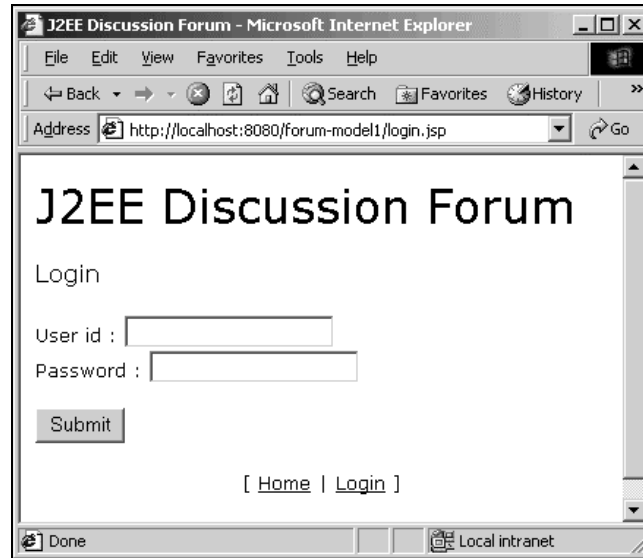
In order to generate the appropriate hyperlink that will take the user to the responses page, we've just hard-coded the name of the page, passing the `id` of the topic as a parameter:

```
    <a href="view-topic.jsp?id=<%= id %>"><%= topic.getTitle() %></a>
  </td>
  <td>
    <%= topic.getResponseCount() %>
  </td>
</tr>
<%
}
%>

</table>
</p>
```

The Login Page

In order to keep anonymous responses off our discussion forum we will require users to log in. If we assume that the users are already set up on the system, then they simply need to log in with their username and password. For our forum there are two users set up – Sam and Simon, both with a password of password. Here's what the page looks like:



As you can see, we have a simple HTML form containing two text fields – one for the user ID (called `id`) and one for the password (called `password`): Once a user types in their information and clicks the **Submit** button, we need to process it and determine whether they are a valid user. To do this, we specify the name of another JSP in the `action` attribute of the `form` tag. So that the user's password doesn't get displayed in the address bar at the top of the browser, we've opted to use the `POST` method, again defined within the opening `form` tag of `login.jsp`:

```
<h2>Login</h2>

<form action="process-login.jsp" method="post">
User id : <input type="text" name="id">
<br>
Password : <input type="password" name="password">
<br>
<br>
<input type="submit" value="Submit">
</form>
```

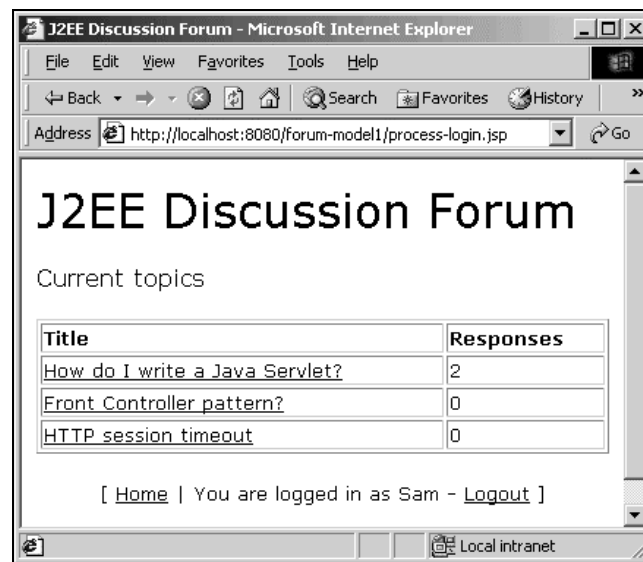
Processing the Login

Next is the page that processes the login details, `process-login.jsp`. We've decided that the user should be forwarded back to the home page once they successfully log in. Therefore, this JSP page doesn't actually contain any visual element. Once more we use a class to represent sample data that would usually be stored in a database, in this case `Users`. Here's the relevant code from the JSP page:

```
<%@ page import="forum.*" %>

<%
    String id = request.getParameter("id");
    String password = request.getParameter("password");
    if (Users.exists(id, password)) {
        session.setAttribute("user", Users.getUser(id));
        pageContext.forward("index.jsp");
    } else {
        pageContext.forward("login.jsp");
    }
%>
```

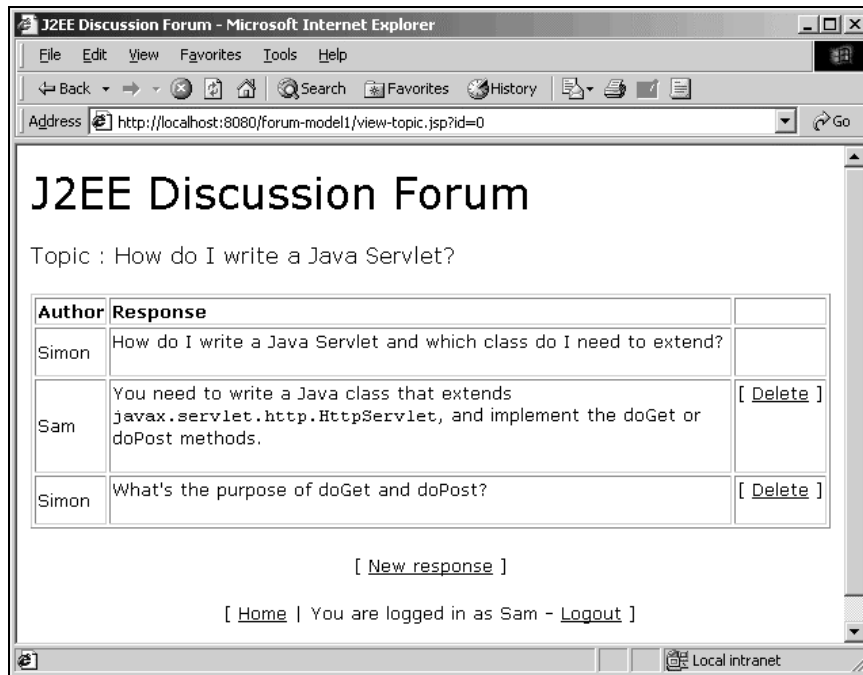
We get the values of the user ID and password that were sent as a result of the user submitting the form and use them to determine whether a user exists on the system. If this is the case, we look up the appropriate `User` instance, place this in the HTTP session under the name `user`, and redirect them to the home page:



If a user doesn't exist with the specified user ID and password, we'll (perhaps unhelpfully) simply return them to the login page.

Viewing the Responses

Next we want the page to display the responses to a particular topic, reached by clicking on a linked topic title on the home page:



Again, this was built as a JSP page, `view-topic.jsp`. It takes the `id` parameter passed as part of the query string and uses it to look up the appropriate Topic:

```
<%
String id = request.getParameter("id");
Topic topic = Topics.getTopic(Integer.parseInt (id));
%>
```

Once it has these parameters, it can get a list of all of the responses to the selected topic and again generate an HTML table:

```
<h2>Topic : <%= topic.getTitle() %></h2>

<p>
<table width="100%" border="1">

<tr>
  <td><b>Author</b></td>
  <td><b>Response</b></td>
  <td>&nbsp;</td>
</tr>
```

```

<tr>
  <td><%= topic.getUser().getId() %></td>
  <td><%= topic.getText() %><br><br></td>
  <td>&nbsp;</td>
</tr>

<%
  Iterator it = topic.getResponses().iterator();
  Response res;

```

Here it builds up the table one row at a time, only displaying those responses that have not been deleted:

```

  while (it.hasNext()) {
    res = (Response)it.next();

    if (!res.isDeleted()) {
%>
      <tr>

```

The first row of data in the table contains the message associated with the original topic, with the remainder representing each response in the order in which it was added. As we're going to offer the ability to delete responses, we have an appropriate link for this that is generated in the last column. This is just a dynamically generated hyperlink that appends the following information to the query string:

- ❑ topic – the ID of the current topic
- ❑ response – the ID of the response to be deleted

```

      <td><%= res.getUser().getId() %></td>
      <td><%= res.getText() %><br><br></td>
      <td valign="top" align="center">
        [ <a href="delete-response.jsp?
          topic=<%= topic.getId() %>&response=<%= res.getId() %>">
          Delete</a> ]
      </td>
    </tr>
  <%
    }
  }
%>

</table>
</p>

```

Finally, we have a link that will allow users to add new responses:

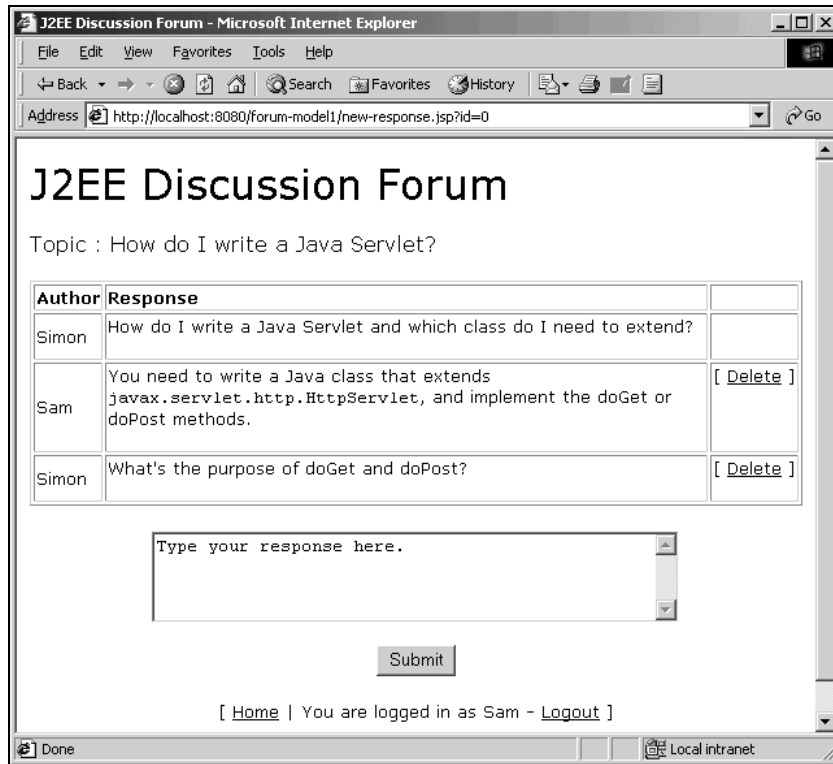
```

<p>
<center>
[ <a href="new-response.jsp?id=<%= topic.getId() %>">New response</a> ]
</center>
</p>

```

Adding a New Response

Adding a new response to an existing topic is a straightforward process, captured by a small HTML form containing a text area in which to write the response. Here's what it looks like:



And here is the relevant HTML in the JSP page, `new-response.jsp`:

```
// ... same as view-topic.jsp ...

<p>
<center>
<form action="process-new-response.jsp" method="post">
<textarea name="text" rows="4" cols="48">Type your response here.</textarea>
<input type="hidden" name="id" value="<%= id %>">
<br>
<br>
<input type="submit" value="Submit">
</form>
</center>
</p>
```

Here we have a text area called `text` representing the body of the response, and a hidden field representing the `id` of the topic to which the new response should be added.

Processing a New Response

When the Submit button is clicked on the `new-response.jsp` page, we need to extract the information from the request and use it to add a new response to the appropriate topic. This can be performed as follows (this code is from `process-new-response.jsp` in the code download):

```
<%@ page import="forum.*" %>

<%
    String text = request.getParameter("text");
    int id = Integer.parseInt(request.getParameter("id"));

    User user = (User)session.getAttribute("user");

    Topic topic = Topics.getTopic(id);
    topic.add(new Response(user, text));

    pageContext.forward("view-topic.jsp?id=" + id);
%>
```

We extract the text and the topic ID from the HTTP request, while the user information can be found in the HTTP Session object because that's where we placed it when the user logged in. Using all of this information we can find the appropriate `Topic` instance and add a new `Response`. Once complete we can forward the user to the responses page once again so that they can see their new response.

Deleting an Existing Response

Finally we must build the functionality to delete a response. Once again, this is provided by a JSP (`delete-response.jsp`) as follows:

```
<%@ page import="forum.*" %>

<%
    int topicId = Integer.parseInt(request.getParameter("topic"));
    int responseId = Integer.parseInt(request.getParameter("response"));

    User user = (User)session.getAttribute("user");

    Topic topic = Topics.getTopic(topicId);
    Response res = topic.getResponse(responseId);
%>
```

We must check to see if the user is the author of the response. We won't allow deletion if this is not the case:

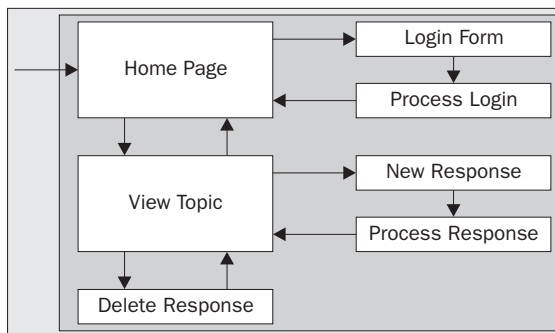
```
    if (res.getUser().equals(user)) {
        topic.remove(res);
    }

    pageContext.forward("view-topic.jsp?id=" + topicId);
%>
```

This is again fairly straightforward. Whatever happens, we simply forward back to the `view-topic.jsp` page where the current state of the topic will be seen.

A Closer Look at the Page Flow for the Application

With our development complete, we can begin to test our web application. The page flow is as shown below:



As with most web sites, all pages link back to the home page. For clarity these links have not been included on the diagram. Also not included on the diagram are the constraints to indicate that a user must be logged in to be able to post a new response, or delete a response, and so on.

As it stands, the application works, but there are a few problems that you may have spotted already.

Authentication

Although we have a login link on the front page, the current structure of the web site means that people can look at the topics without logging in. In fact, any user could bypass the login page completely, meaning that they would also be able to add new responses anonymously. For example, a user could open their browser and jump straight to the following page: <http://localhost:8080/forum-model1/new-response.jsp?id=0>.

One solution would be to require users to login before they can view the list of topics in the forum. However, this might reduce the likelihood of them visiting our site in the future. A better alternative would be to ask the user to login the first time that they wanted to perform some action on the site such as adding or deleting a response – that way they can still view all the valuable information quickly and easily.

This is easy enough and can be achieved by adding the following scriptlet to the top of those restricted pages:

```
<%
  if (session.getAttribute("user") == null) {
    pageContext.forward("login.jsp");
  }
%>
```

In this scriptlet we check whether a `User` instance has been set in the user's session. If one does exist, we can assume that the user has already logged in and the page will be rendered as usual. If a `User` instance doesn't exist (under the name of `user`), we can forward the user to the login page.

This code fragment could be copied and pasted into each JSP that needs to be secure. This style of reuse is a step in the right direction but is hardly the most efficient. After all, if we needed to make some changes then we've certainly got our work cut trying to find each and every place that it's used.

Of course we could place this functionality into a reusable software component such as a JavaBean or a custom tag, or at a simpler level, another JSP that gets included wherever necessary. This is a further step in the right direction but still requires that we make use of the functionality everywhere that we need it. We've already noted earlier in the chapter that centralizing user access would make securing our application an easier task.

Authorization

In the same way as we have done for authentication, authorization code could be duplicated throughout the system. An example of this is the code that checks whether the current user can delete the selected response. If we forgot to make use of this functionality somewhere then data could become available to or corrupted by inappropriate users. While it is probably not much of a problem in a small discussion forum, e-commerce sites generally have confidential information available (such as user's credit card numbers and home addresses). This information should certainly not be accessed by anybody except its rightful owner.

Extending the Application

We might want to extend our application so that we log a user's progress through the system, and track which pages in the system are being used more than others. This is especially useful for e-commerce type solutions where customers often leave if a web site is deemed too slow.

If we wanted to create a log that tracked the users throughout the site, whereabouts would this functionality be placed? Also, how easily could it be turned off when we've finished analyzing the data? In this example application, there is no single place – no central point – where such logging could be located. Instead, it would have to be placed into each page, either explicitly, by calling some included code, or by using a reusable component such as a JavaBean or custom tag. The decision to include a logging facility is therefore one that should be made at design time, before the coding commences.

Refactoring Applications

The problems we've seen in our discussion forum highlight the types of problems that often tend to appear in applications. Fortunately all is not lost, even if the application contains such problems: we can attempt to refactor the application. **Refactoring** basically means changing from one form to another, and although not strictly "design", it does promote some of the same thought processes. When we refactor a piece of software, it generally means that we change the internal structure in some way without damaging the functionality that it provides.

Refactoring is something that we probably all do already without realizing. Some of the reasons why we might want to refactor a particular piece of software might include to tidy it up, to fix bugs, or to simply extend it. The downside to all of this is that refactoring can be a very time-consuming process.

If we had taken a step back before we plowed straight into coding our discussion forum, we could have foreseen that we needed authentication on many pages and designed a flexible, standard way of performing this task. As it stands, we've had to copy the same piece of code into several different pages in our application, therefore reducing the maintainability and increasing the likelihood that somebody might breach our system.

We highlighted many of these issues when we discussed the Model 1 architecture near the start of the chapter. Lack of extensibility is one of the key downsides to web applications built upon the Model 1 architecture – there is no centralized place for business logic and it is generally embedded into the view components such as JSPs. This makes extensibility difficult to achieve, which in turn makes it hard to keep up with the ever-changing requirements of businesses in the internet world. This isn't to say that applications built using a Model 1 architecture don't have their place – small sites, prototypes, and proofs of concepts are well-suited to this. It's just that most large-scale web sites need a more maintainable and extensible solution.

With this in mind (and instead of refactoring the entire system), let's look at how we can use Model 2 architecture and a handful of J2EE patterns to design and build a more maintainable, extensible, and secure version of our discussion forum.

Building the Forum Using Model 2 Architecture

Model 2 architecture accepts requests through a controller component where it can service the request itself or delegate this task to some other component. The task of rendering the response is then dispatched to the appropriate view component. There are of course various ways in which this can be implemented. We're going to look at four of these design patterns:

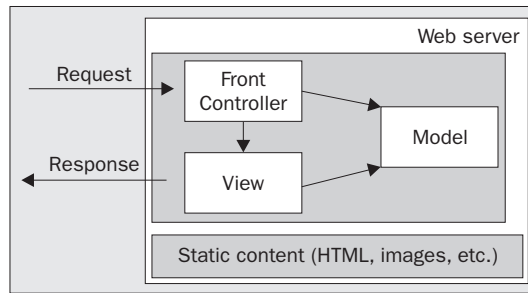
- ❑ Front Controller
- ❑ Intercepting Filter
- ❑ View Helper
- ❑ Service to Worker

Using the Front Controller Pattern

Our current discussion forum has many views. System services (for example, data retrieval) and business logic are intermingled within the views, leading to a decrease in maintainability and a reduction in reusability. JSP pages are often developed by two types of people: web designers (who handle the look and feel of the site) and J2EE developers (who take care of the business logic). Placing too much Java code into the page usually makes it more difficult for web designers to modify the look and feel as they battle with Java code that they might not understand. For example, the logic that retrieves a `Topic` instance from the "database" based upon a topic ID, or the business rule that says only the author of a response can delete that response.

The other feature of our current implementation is that all of the links to other pages are hard-coded within the pages themselves. Should we need to change the flow of the system in any way, we'll have to resort to modifying a number of JSP pages.

Ideally we would like to place all of this logic in a single location to help us ensure that this type of functionality is consistent, maintainable, and extensible in the future. The **Front Controller** pattern can help us to achieve these goals. Here's how it fits into the Model 2 (programming-centric) architecture:



This is exactly the same as the Model 2 architecture diagram that we saw towards the start of the chapter where a controller component is the entry point for web requests. The controller performs some processing of the request and may make changes to the underlying model. The task of rendering the response is then dispatched to a view component.

A front controller is also a good place in which to centralize services such as error handling and logging. Centralizing system services and the flow between pages has many benefits as it moves business logic and other system-level service code out of JSP scriptlets, back into reusable components, therefore promoting reusability of functionality across different types of requests, as we'll be seeing shortly.

There are various strategies by which a front controller can be implemented, with the typical strategy focusing on the use of a servlet. There are several reasons why this role is better implemented using a servlet than a JSP. As we've noted before, outputting content back to the client is not an ideal use for a servlet because it involves writing lots of print statements – effectively tying together the content and the logic. JSP pages, on the other hand, are much better suited to delivering content, because they are written as content containing small bits of logic wherever necessary. As a controller component doesn't actually deliver any content itself, implementing it as a JSP results in a page containing no content – just the logic required to process the requests. For this reason, a servlet is the preferred strategy.

Of course centralizing all of this functionality can lead to a large, bloated controller component that has responsibilities for the entire web application. There are a number of ways in which this problem can be solved, one of which is to have several front controllers, each responsible for particular area of the site. For example, an e-commerce site might have one controller responsible for servicing all requests related to products and another to service all payment requests. Another solution is to use the **Command and Controller** implementation strategy we will discuss shortly.

Implementing the Front Controller Pattern

For the moment, here is an example skeleton implementation of the Front Controller pattern.

```

package forum;

import java.io.IOException;

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

public class FrontController extends HttpServlet {
  
```

```

protected void processRequest(HttpServletRequest req,
                             HttpServletResponse res)
    throws ServletException, IOException {
    RequestDispatcher dispatcher =
        getServletContext().getRequestDispatcher("name of view component");
    dispatcher.forward(req, res);
}

protected void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    processRequest(req, res);
}

protected void doPost(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    processRequest(req, res);
}
}

```

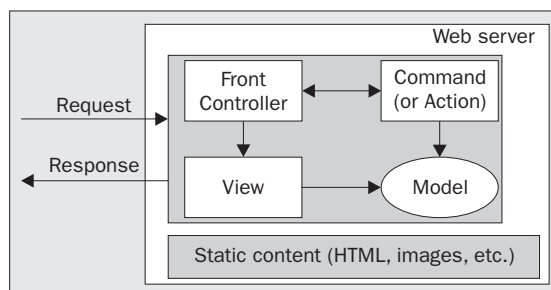
In this case, the front controller is simply an extension of `HttpServletRequest` with default implementations of the `doGet()` and `doPost()` methods that delegate processing of the request to another method called `processRequest()`. This is done to ensure that no matter how the request is made, the front controller will service it.

We've left out the majority of the body of the `processRequest()` method, but essentially a front controller will perform some processing that is associated with a request, and then dispatch to a view component to render the response. The view components are typically JSP pages. Once the controller has completed performing its business logic, it can dispatch to a JSP page through the `RequestDispatcher`.

At this point, you may have a question. If the controller is to be responsible for handling all requests, how does it know what the request is and therefore how to handle it?

The Command and Controller Strategy

In the Command and Controller strategy the logic to handle each specific request is moved out into a separate component. This architecture is illustrated below:



Each of these extra components represents a specific command (or **action**), and the component encapsulates the logic to perform that action.

The front controller delegates the handling of requests to the appropriate command component, which may modify the state of the underlying model. Once the command component has completed its work, the controller again dispatches the task of rendering the response to the appropriate view component. But how does the controller know which command component to use? We'll tackle this issue in a moment.

Action Classes

The first thing we need to do to implement the Command and Controller strategy is to define the interface between the controller and action components. We can do this either by creating an interface or an abstract class. For the purposes of our example, we'll define an abstract class that all actions in our web application must extend (this means that we have a single place in which to add common functionality for all actions in the future):

```
package forum;

import javax.servlet.http.*;

public abstract class Action {
```

Subclasses will implement their specific business and processing logic in the `process()` method. To ensure that the action classes have access to the same environment and information as servlets, the `process()` method takes a reference to the HTTP Request and Response objects in the same way that the `processRequest()` method in the `FrontController` class did. Once the processing has been completed, the action can return a string to identify the view component to which the controller should dispatch next:

```
    public abstract String process(HttpServletRequest request,
        HttpServletResponse response);
}
```

Defining the interface between the controller and the action components helps to decouple them from each other, which means that we can change the controller or the actions without affecting the other.

The next stage is to put together the logic that will figure out what the request is and delegate the processing to the appropriate action component.

Communicating the Type of Request

There are a number of ways that the type of request can be communicated to the controller servlet, most of which are focused around passing parameters to the servlet over HTTP (in a similar way to how we pass the `topic id` to the `view-topic.jsp` page). The problem with sending additional parameters to indicate the type of request is that the URLs need to be written carefully – after all, it is easy to misspell parameter names and get the query string syntax wrong.

Another mechanism would be to use a string that represented the type of request that we wanted to perform and pass this to the servlet as additional path information. We already know how mappings can be defined between a servlet and a URI via the `web.xml` file. An example of this is shown overleaf, which defines a mapping between our `FrontController Servlet` and the URI `/controller/*`:

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

<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>

  <servlet>
    <servlet-name>FrontController</servlet-name>
    <servlet-class>forum.FrontController</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>FrontController</servlet-name>
    <url-pattern>/controller/*</url-pattern>
  </servlet-mapping>

</web-app>
```

Now we can make calls to the controller by appending `controller/ViewTopic?id=0` to our URL. Here, `ViewTopic` is the additional path information (also known as **path info**) and represents the type of request we are making. The query string can be used as usual, with `id=0` providing the parameters for our specific request to view a topic. This is much cleaner than adding additional parameters to indicate the type of request. Instead, this information is now part of the URL that we use. With this side of the request figured out, how does the controller know what to do, and where does this logic reside?

What we need to do here is to map the string that we obtain from the additional path info onto a specific action instance. Once again there are many strategies for this, with the most flexible being externalizing this mapping, through an XML file for example. Other examples here include hard-coding the mapping inside the controller, or perhaps using a properties file.

For simplicity, in our example we'll build a separate component in which this mapping will be encapsulated. In a production application, hard-coding such configuration information means that changes require the appropriate Java class to be modified, recompiled, and redeployed. In other words, it decreases the maintainability. However, hard coding this information into a separate component will suffice for our simple example and we'll call this class our `ActionHelper`. The mapping information is stored in a `HashMap`. Here's the code listing for this class:

```
package forum;

import java.util.HashMap;

public class ActionHelper {

    private static HashMap actions = new HashMap();

    static {
        actions.put("ViewTopic", "forum.ViewTopicAction");
        actions.put("Login", "forum.LoginAction");
        actions.put("Logout", "forum.LogoutAction");
        actions.put("NewResponse", "forum.NewResponseAction");
        actions.put("ProcessNewResponse", "forum.ProcessNewResponseAction");
        actions.put("DeleteResponse", "forum.DeleteResponseAction");
    }
}
```

The `ActionHelper` class effectively maintains a mapping between request names (or types) and the fully qualified class names of the classes that can process the requests. Given the name of a request, the static `getAction()` method returns an instance of an `Action` class that can be used for processing:

```
public static Action getAction(String name) {
    Action action = null;

    try {
        Class c = Class.forName((String)actions.get(name));
        action = (Action)c.newInstance();
    } catch (Exception e) {
        e.printStackTrace();
    }

    return action;
}
```

The next step is to plug all of this into the `processRequest()` method of our `FrontController` as this will be the single entry point for all requests in our web application:

```
package forum;

import java.io.IOException;

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

public class FrontController extends HttpServlet {

    protected void processRequest(HttpServletRequest req,
                                  HttpServletResponse res)
        throws ServletException, IOException {
```

Here we find which action should be used:

```
String actionName = req.getPathInfo().substring(1);
```

Now we use the helper class to locate the action:

```
Action action = ActionHelper.getAction(actionName);
```

The next step is to process the action, so that we can find out which view to show the user next:

```
String nextView = action.process(req, res);
```

Finally we redirect to the appropriate view:

```
RequestDispatcher dispatcher =
    getServletContext().getRequestDispatcher(nextView);
dispatcher.forward(req, res);
}

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

    processRequest(req, res);
}

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

    processRequest(req, res);
}
}
```

With the code to look up actions and process requests encapsulated elsewhere, our controller servlet is fairly minimal. In fact all of the components that we have implemented as part of the Command and Controller strategy have been fairly lightweight. This is good from both a maintainability and reusability perspective as small components are generally easier to maintain (and bug fix) and are also more likely to be reusable elsewhere. The framework that we've built is also extensible because new request handlers (actions) can easily be built and added to the system.

The Apache Struts Framework

The implementation of the Front Controller pattern presented here is fairly simple and straightforward but there are many improvements that can be made. One such example is externalizing the mapping between action names and the `Action` classes that are going to service the request. Although such work isn't particularly complex, it is still time consuming nonetheless. For this reason, there are number of third-party frameworks available for developers to use as a starting point for their web applications. One such example is **Struts** from the Jakarta Project (<http://jakarta.apache.org>).

Struts is an open source framework providing an implementation of not only a front controller, but a complete implementation (or **framework**) of the Model 2 architecture. The key components are similar to those presented here, including for example, a controller servlet (called `ActionServlet`), `Action` classes in which to place functionality for processing a request and `ActionBean` classes in which to encapsulate data coming in from a request. In addition to this, Struts also contains a comprehensive collection of JSP tag libraries for the easy assembly of HTML-based forms and JSP pages in general.

One of the most important features of Struts is that pretty much everything about it is externally configured through the `struts-config.xml` file. This includes the mappings between action names and `Action` classes (called `ActionMappings`) and also a concept called `ActionForwards`.

For the simple example presented here, the `Action` classes return a string representing the URI of the JSP that should be displayed next. In Struts, however, the `Action` classes return a symbolic name (wrapped up in an `ActionForward` instance) representing which view component should be displayed next. As an example, consider a login form containing a username and password. On submission of the form, the login is either successful or not. In this case, the `Action` class could return symbolic names such as `success` or `failure`. These symbolic names are then configured in the `struts-config.xml` file and it's here that the physical URI to the appropriate JSP is specified. This is a very powerful feature and is yet another way in which the flow and structure of the web application can be taken out of the code to increase maintainability. After all, if the structure of the site changes, only the configuration file needs to change.

A full explanation of Struts is beyond the scope of this book, but a more detailed study of this framework can be found in the following Wrox titles: Professional JSP, 2nd Edition (ISBN 1-861004-95-8) and Professional JSP Site Design (ISBN 1-861005-51-2).

Now let's get back on track. With the design and framework of a Model 2-based discussion forum in place, let's take a look at how the forum can be implemented once again in Model 2 architecture.

Viewing the Responses

Previously, to view the responses to a topic we made a direct request to the `view-topic.jsp` page, passing the `id` of the specific topic. At the top of this page was a JSP scriptlet that found this parameter and used it to look up the appropriate `Topic` instance for use further down the page when building up the table of responses. One of the benefits of the Command and Controller strategy is that we can move this sort of code out of the JSP and back into a reusable component. With this in mind, we can implement the action to view a topic:

```
package forum;

import javax.servlet.http.*;

public class ViewTopicAction extends Action {

    public String process(HttpServletRequest request,
        HttpServletResponse response) {
```

In the `process()` method, we first get the `id` parameter and use it to look up the appropriate `Topic` instance:

```
String id = request.getParameter("id");
Topic topic = Topics.getTopic(Integer.parseInt(id));
```

Now that we have the topic, we place it in the request ready for use on the view:

```
request.setAttribute("topic", topic);
return "/view-topic.jsp";
}
}
```

We've literally just moved the code into an `Action` subclass. Once the `Topic` instance has been located, we then need to make this available to the page in order that it can build the table of responses. To do this we use the HTTP Request object as a temporary storage area for this object:

```
request.setAttribute("topic", topic);
```

Using the HTTP Request (or even the Session) is a useful and frequently used mechanism for passing information between the various components within web applications. This works between two servlets, two JSP pages, or a mixture of the two.

On completion of this action, the controller will dispatch control to the `view-topic.jsp` page on which we can use the standard `<jsp:useBean/>` action to locate the object again. The remainder of the page remains unchanged:

```
<jsp:useBean id="topic" class="forum.Topic" scope="request"/>

<p>
<h2>Topic : <%= topic.getTitle() %></h2>
</p>
```

Following this we just build the table of responses as before.

The final step in using this is to ensure that we change all direct links to the `view-topic.jsp` into links to the controller, not forgetting to include the name of the action that we wish to perform.

```
/controller/ViewTopic?id=0
```

Although this particular action is fairly small, it does illustrate the point that we raised earlier about enhanced maintainability and reusability. We can now make use of this action elsewhere in our application rather than coding a scriptlet on a JSP. This means that should we wish to modify this behavior, we'll know where to find it and the modifications will be reflected elsewhere in our application automatically.

Processing the Login

In a similar way, we can encapsulate the logic associated with processing the user's login into an `Action` class, `LoginAction`:

```
package forum;

import javax.servlet.http.*;

public class LoginAction extends Action {

    public String process(HttpServletRequest request,
        HttpServletResponse response) {
```

```
String id = request.getParameter("id");
String password = request.getParameter("password");
String view;
if (Users.exists(id, password)) {
    request.getSession().setAttribute("user", Users.getUser(id));
    view = "/index.jsp";
} else {
    view = "/login.jsp";
}

return view;
}
```

As you can see, we've literally moved the code from the `process-login.jsp` page. This page didn't actually contain any presentation at all in the page-centric implementation and this is another good reason for moving the code into a small reusable component. It's not only easier to find this code to maintain it in the future, but JSP pages are really suited to rendering responses rather than being containers for lots of Java code.

To ensure that our action gets called, we again need to change any references to the `process-login.jsp` page, including the `action` attribute of HTML form tags:

```
<form action="controller/Login" method="post">
```

Adding a New Response

We've already seen how the `view-topic.jsp` page has changed: the short scriptlet at the top of the page has been removed. The `new-response.jsp` page is effectively the same, with an additional HTML form underneath the table of responses. For this reason and rather than recoding the logic to look up the appropriate topic, we can extend the `ViewTopicAction` class helping to ensure that the functionality is consistent.

```
package forum;

import javax.servlet.http.*;

public class NewResponseAction extends ViewTopicAction {

    public String process(HttpServletRequest request,
        HttpServletResponse response) {

        super.process(request, response);

        return "/new-response.jsp";
    }
}
```

Although a simple example, this shows how the action components can be reused across different types of requests. Here, we are using the same functionality but dispatching to a different view.

Processing a New Response

The logic to add a new response from the HTML form can also be placed within an Action class as illustrated next.

```
package forum;

import javax.servlet.http.*;

public class ProcessNewResponseAction extends Action {

    public String process(HttpServletRequest request,
        HttpServletResponse response) {

        String text = request.getParameter("text");
        int id = Integer.parseInt(request.getParameter("id"));

        User user = (User)request.getSession().getAttribute("user");

        Topic topic = Topics.getTopic(id);
        topic.add(new Response(user, text));

        return "/controller/ViewTopic?id=" + id;
    }
}
```

Instead of dispatching to a JSP, we've asked that the request be forwarded onto the `ViewTopic` action. This is a useful way of chaining actions together and again shows the ability to reuse them across different types of requests.

Deleting an Existing Response

The final action that we have in our web application is the ability to delete an existing response. Once again we can move the business logic associated with this into an Action component:

```
package forum;

import javax.servlet.http.*;

public class DeleteResponseAction extends Action {

    public String process(HttpServletRequest request,
        HttpServletResponse response) {

        int topicId = Integer.parseInt(request.getParameter("topic"));
        int responseId = Integer.parseInt(request.getParameter("response"));

        User user = (User)request.getSession().getAttribute("user");

        Topic topic = Topics.getTopic(topicId);
        Response res = topic.getResponse(responseId);
    }
}
```

Only the original author of a response can delete it, and for this reason the `DeleteResponseAction` contains the appropriate logic to make this check. However, what it (or the other actions) doesn't check is whether the user is actually logged on. We'll deal with this issue in the next section.

```
        if (res.getUser().equals(user)) {
            // yes, so delete it
            topic.remove(res);
        }

        return "/controller/ViewTopic?id=" + topicId;
    }
}
```

Using the Intercepting Filter Pattern

Checking that the user is logged on is another common piece of functionality that would be an ideal candidate to encapsulate into a reusable component. We could of course wrap this verification up into a small component and call it from the appropriate actions. However, what we really want is for unauthenticated requests to be redirected to the login page. Let's look at how the **Intercepting Filter** pattern can help.

A Closer Look at User Verification in the Discussion Forum

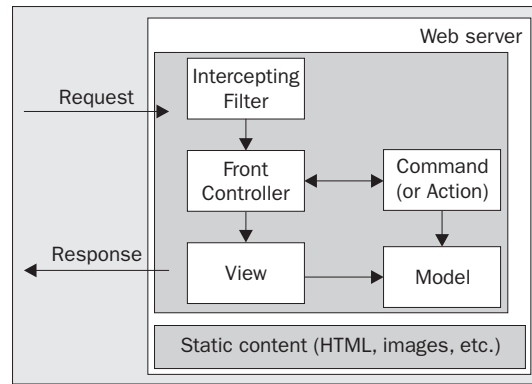
One of the features that our discussion board doesn't support is anonymous response to topics. To ensure that this doesn't happen, our current implementation uses a small snippet of code at the top of those pages that require a user to be logged in:

```
<%
    if (session.getAttribute("user") == null) {
        // no, so redirect them to the login page
    }
%>
```

We've seen this already, and all it does is check that the user is logged in before allowing them to see the page that they requested. If the user isn't logged in yet, we simply forward them on to the login page.

The problem with this approach is that this code is scattered throughout the views (pages) in our application. This means that we might not be entirely sure which views contain the code. Subsequently, some pages that should have this code may not, and if we need to update the verification code we need to modify every page that contains it.

One way around this problem is to use intercepting filters. Intercepting filters are a useful way of performing pre-processing on requests before they are handled by the components in our web application. If we extend the architecture yet further, we can insert an intercepting filter to pre-process the requests:



The types of tasks that we might want an intercepting filter to perform could be:

- Checking the client type so that the appropriate type of site can be served up
- Checking whether the user has a valid session
- Checking that the user is logged in

In our discussion forum we can use an intercepting filter to ensure that any requests to specific pages are authenticated. Having identified that an intercepting filter is a useful component to have in our web application, how do we go about building one?

Implementing an Intercepting Filter

In a previous chapter we introduced **filters**, a new feature of the Servlet 2.3 specification. Using the same techniques as we described in that chapter, we can build a simple filter, and define a handful of mappings specifying those URLs on which we wish it to operate.

First of all, let's start with the filter class itself. We'll call it `AuthenticationFilter`:

```
package forum;

import java.io.IOException;

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

public class AuthenticationFilter implements Filter {

    private FilterConfig filterConfig;

    public void init(FilterConfig filterConfig) throws ServletException {
        this.filterConfig = filterConfig;
    }

    public void doFilter(ServletRequest req, ServletResponse res,
        FilterChain chain) throws IOException, ServletException {

        HttpServletRequest request = (HttpServletRequest)req;
        HttpServletResponse response = (HttpServletResponse)res;
```

In the `doFilter()` method, we check to see if the current user logged in. If not, we redirect them to the login page:

```

    if (request.getSession().getAttribute("user") == null) {
        RequestDispatcher dispatcher =
            filterConfig.getServletContext().getRequestDispatcher("/login.jsp");
        dispatcher.forward(request, response);
    } else {
        chain.doFilter(request, response);
    }
}

public void destroy() {
}
}

```

What is important here is that the functionality associated with this filter is almost identical to the scriptlet of code that we used in our Model 1 version of the discussion forum. All it does is check that the user is logged in and redirect them to the `login.jsp` page if this is not the case. One of the goals of OO is to encapsulate reusable functionality and that's exactly what we've done here. We've moved this logic out of the views and back towards the front of the request handling process.

With the filter built, we now need to plug it into our web application. As we've seen before, this is achieved through the web application deployment descriptor – the `web.xml` file:

```

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

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">

<web-app>

    <filter>
        <filter-name>AuthenticationFilter</filter-name>
        <filter-class>forum.AuthenticationFilter</filter-class>
    </filter>

    <filter-mapping>
        <filter-name>AuthenticationFilter</filter-name>
        <url-pattern>/controller/NewResponse</url-pattern>
    </filter-mapping>

    <filter-mapping>
        <filter-name>AuthenticationFilter</filter-name>
        <url-pattern>/controller/ProcessNewResponse</url-pattern>
    </filter-mapping>

    <filter-mapping>
        <filter-name>AuthenticationFilter</filter-name>
        <url-pattern>/controller/DeleteResponse</url-pattern>
    </filter-mapping>

```

```
<servlet>
  <servlet-name>FrontController</servlet-name>
  <servlet-class>forum.FrontController</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>FrontController</servlet-name>
  <url-pattern>/controller/*</url-pattern>
</servlet-mapping>

</web-app>
```

We've already defined that all URLs starting `/controller/` are directed to our `FrontController` servlet. Alongside this we define a filter called `AuthenticationFilter` and tell our web application to use the `AuthenticationFilter` class from the `forum` package. Once this has been set up, we can specify the set of URLs that trigger the filter. In our application, we need to pre-process all requests coming in for the following actions:

- `NewResponse`
- `ProcessNewResponse`
- `DeleteResponse`

This is a very powerful and flexible mechanism, because we can easily reconfigure our filter should the security requirements of our application change.

The patterns that we have just discussed cover much of our example application. Although this book is about servlets, a chapter about web application design wouldn't be complete without a quick look at a few more patterns:

- `View Helper`
- `Service to Worker`

Using the View Helper Pattern

The View Helper pattern is a way of taking logic embedded in the view (for example scriptlets inside a JSP) and wrapping it up as reusable functionality for use in other view components.

Advantages of the View Helper Pattern

Although we've tried to move much of the functionality away from the view components and back towards the front of the request handling cycle, there still remains some functionality in the views. A good example would be the JSP scriptlets inside the `view-topic.jsp` page that determine whether a particular response has been deleted and should be hidden.

The purpose of the view component is to present information back to the user. All of the business processing associated with the request should have been performed by this stage, leaving the view component to perform any logic specifically related to presenting the information. While we can include business logic in a JSP page via scriptlets, we have already noted that this is not a wise approach because the code in the scriptlet is not in a very reusable form.

To solve this problem, this type of functionality can be moved into helper components and subsequently reused across the web application. On an implementation level, these helpers could be built as either of the following:

- JavaBeans – for use in servlet/JSP views
- Custom tags – for use in JSP views

In fact we've already seen an example of the View Helper pattern in the `view-topic.jsp` page. The `ViewTopicAction` looks up the appropriate `Topic` instance and places it into the HTTP Request object, ready for the JSP to find it. To recap, here's the code for the `process()` method in the `ViewTopicAction`:

```
public String process(HttpServletRequest request,
                    HttpServletResponse response) {

    String id = request.getParameter("id");
    Topic topic = Topics.getTopic(Integer.parseInt(id));
    request.setAttribute("topic", topic);
    return "/view-topic.jsp";
}
```

To make use of the information that this action places into the HTTP request, we might have implemented this lookup in `view-topic.jsp` using a simple JSP scriptlet as follows:

```
<%
    Topic topic = (Topic)request.getAttribute("topic");
%>
```

Instead, however, we decided to use the `<jsp:useBean/>` action in `view-topic.jsp`:

```
<jsp:useBean id="topic" class="forum.Topic" scope="request"/>
```

This is just one such example, and it just happens that in this situation the helper is already written for us. What is important is that we are removing as much logic as possible from the view components in our web application, and placing it inside reusable components.

Another example includes the JSP scriptlet that is used to iterate over a collection of `Response` objects in the `view-topic.jsp` page:

```
<%
    Iterator it = topic.getResponses().iterator();
    Response res;

    while (it.hasNext()) {
        res = (Response)it.next();
        if (!res.isDeleted()) {

```

```

<tr>
  <td><%= res.getUser().getId() %></td>
  <td><%= res.getText() %><br><br></td>
  <td valign="top" align="center">
    [ <a href="controller/DeleteResponse?
      topic=<%= topic.getId() %>&response=<%= res.getId() %>">
      Delete</a> ]
  </td>
</tr>
<%
}
%>

```

In this example there is much more Java code and therefore much more that can go wrong. Ideally we would like to be able to reuse this type of functionality on other pages that build up the content by iterating over a collection of Java objects. In this situation, a JSP custom tag could be used as a view helper as follows:

```

<forum:iterate id="res"
  className="forum.Response"
  collection="<%= topic.getResponses() %>">
  <%
  if (!res.isDeleted()) {
  %>
  <tr>
    <td><%= res.getUser().getId() %></td>
    <td><%= res.getText() %><br><br></td>
    <td valign="top" align="center">
      [ <a href="controller/DeleteResponse?
        topic=<%= topic.getId() %>&response=<%= res.getId() %>">
        Delete</a> ]
    </td>
  </tr>
  <%
  }
  %>
</forum:iterate>

```

Here, the Java code responsible for performing the iteration has been moved inside a custom tag, moving it away from the page and resulting in a much cleaner JSP containing much less Java code than before. In doing this, we have created a reusable view helper that can be used on other pages in the web application.

Although the discussion of how such a custom tag might be implemented is beyond the scope of this chapter, the full sourcecode for it is provided with the downloadable sourcecode for the book.

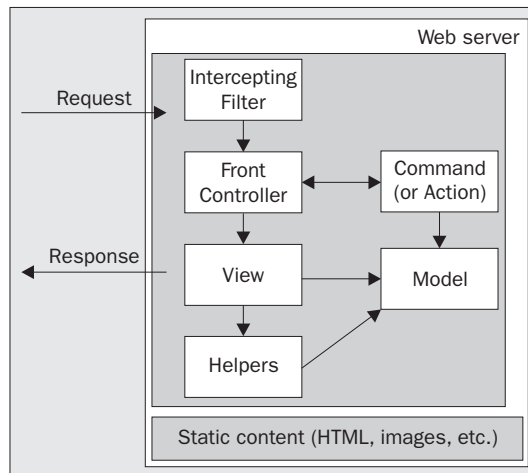
The type of functionality that this pattern is applicable to includes:

- ❑ Retrieval of data (for example getting data from the model to display)
- ❑ Logic related to presentation of data but not to formatting (for instance determining whether a particular piece of data should be displayed)
- ❑ General presentation layer-related logic (such as iterating over a collection of data items)

Using the Service to Worker Pattern

The final pattern we will consider is **Service to Worker**, which can be described as a "macro pattern". It's really just a combination of a front controller pattern, along with views and their helpers.

This pattern describes the architecture of our discussion forum, where we've wrapped up a large amount of the processing and presentation logic into reusable components. In other words, we're delegating the task off to worker components as illustrated in the following diagram:



As with our discussion forum, the controller delegates processing to the actions, which in turn perform processing associated with servicing the request. These actions may modify the underlying model. The controller then dispatches the request to the next view component, which will in turn use the underlying model and helper components (such as JavaBeans and/or custom tags) to build and render the response.

Each of the patterns that we've looked at focuses on one particular area of development:

- ❑ Front Controller – a centralized place for logic associated with servicing requests; here views are responsible only for the presentation of information to the user
- ❑ Intercepting Filter – a centralized place for intercepting requests, for example to allow or disallow them
- ❑ View Helper – wraps up reusable functionality used by the views and moves unnecessary logic away from them

Although each of these does address a particular problem, using them in isolation means that only that particular problem will be solved. In the case of our discussion forum, using the Front Controller in isolation may still mean that the view components contain unnecessary Java code embedded inside the JSP pages. On the other hand, using View Helpers in isolation means that the pages will contain unnecessary business logic required to process the request.

The Service to Worker pattern is a combination of all of these patterns and ensures that each of the problem areas is addressed. This means that the logic associated with processing a request is centralized, while the logic associated with presenting information is also centralized, and reusable.

As we can now see from our discussion forum, we not only have a more structured system, but one that is easily maintainable. If we encounter a bug, it shouldn't take us long to track it down. We also have a system that is extensible. The addition of new functionality simply entails adding new actions into our request-handling framework. There is now much more scope for reusability of the components within our application and this again improves the maintainability.

Summary

Application design is an important facet of software development and is often neglected because of the ever-decreasing timescales of Internet development and time-to-market. Including time in the development process to perform design will ultimately save time in the long term. While it is very easy to put together a web application representing the business needs of a company, the lack of good design will result in an application that is expensive to maintain and extend.

We've covered two common architectures for building web applications in this chapter:

- ❑ Model 1 – page-centric
- ❑ Model 2 – programming-centric

During our discussion of these two architectures, we looked at the typical implementation strategies along with some of their characteristics, including:

- ❑ Maintainability – how easy it is to keep the application running smoothly
- ❑ Extensibility – the ease with which we can add new functionality to the system
- ❑ Security

We then introduced some of the other J2EE components into the picture and this led us to a discussion about how to partition functionality within the web application.

Following this we looked more closely at application design. We considered how designs are documented, introducing the concept of a design pattern, and the J2EE patterns catalog.

We then moved on to look at how a simple discussion forum application could be built using a Model 1 architecture. This was approached without any design consideration and subsequently we found some problems with authentication and duplication of business logic – both of which were intermingled within the view components – which adversely affect maintainability, extensibility, and reusability.

Following on from this, we took a step back and looked at designing the application using a Model 2 architecture which makes use of a handful of J2EE patterns:

- ❑ Front Controller
- ❑ Intercepting Filter
- ❑ View Helper
- ❑ Service to Worker

We explored the purpose of these patterns and looked at how they are useful when we design our discussion forum to maximize maintainability, extensibility, and reusability. During this we looked at some of the implementation strategies for these patterns and presented a new implementation for our example application.

In the next chapter we will look at how we can optimize and scale web applications effectively.

