The Model-View-Controller Design Pattern: 'Model 2' JSP Development

IN THIS CHAPTER

- The Model-View-Controller Design Pattern
- The Origins of Model 1 / Model 2
- How Struts Implements the Model 2 Pattern
- Conclusions

Architecting great software takes skill, experience, and a lot of time. Of course, it goes much faster if you start with a significant amount of your code already written—especially if that code is based on architecture and designs that are time-tested and proven.

The goal of this chapter is to demonstrate how this principle is embodied in Struts. You'll also gain an understanding of the architecture and design patterns that Struts is based on. This will make you a better Struts developer and help you get up to speed faster.

In addition, you'll be presented a summary of the history and development of these design patterns. The perspective this provides will deepen your appreciation for the value that Struts adds to a development project. It will also help you recognize opportunities to reuse the patterns again in the future.

Specifically, in this chapter you will learn

- The MVC design pattern and how it speeds development and makes managing changes easier.
- What Model 1 and Model 2 JSP development are and where these terms originated
- How Struts implements the Model 2 pattern

The Model-View-Controller Design Pattern

A *design pattern* is a series of objects and object relationships that provide a proven, extensible solution to a particular software design problem. The Model-View-Controller (MVC) pattern is arguably the best known, most famous design pattern of them all.

MVC was originally developed in the late 1970s at the Xerox Palo Alto Research Center (PARC). It was originally built to manage the GUI and user interaction on some of the first window-based computers (another innovation from the PARC—in addition to Ethernet, local area networks, mice for input devices, and numerous other firsts).

The design problem that MVC solves is that of simplifying three primary functions that are common in many applications:

- Maintaining the data in a back-end store or remote system
- Building the end-user presentation layer
- Maintaining the conditional logic that decides which screens are presented to the user, what happens when errors occur, and exactly how and when the remote systems are updated

It is possible to combine all this processing into a single module and get a system to work. (In fact, a significant amount of early JSP development did exactly that!) Problems primarily occur when you try to perform maintenance on the code. In the case of JSP, this is compounded by the fact that the HTML designers who maintain the look and feel of the application are different people (and have different skill sets) from those who maintain the Java code that controls the processing.

MVC addresses this problem by separating the code into three distinct areas:

- Model components that maintain data in a back-end store or remote system
- Views that build the end-user presentation layer
- Controllers to maintain conditional logic that decides which screens are presented to the user, what happens when errors occur, and exactly how and when the remote systems are updated

MVC simplifies maintenance by keeping all this logic from becoming intertwined. It allows the details of each piece to be hidden from the others and reduces the coding linkages between them. This is how MVC provides a natural boundary between the people who write the Java and the people who maintain the HTML and presentation layer.

A good example of this is in how MVC can simplify exception processing. Imagine that after a user logs in, you send a request to a remote system to fetch the user's customer information. What do you do if the remote system is unavailable? In normal JSP processing, it's common to embed logic at the top of your JSP file to detect this and change what you display to the user when the problem occurs. Using MVC, you can pull this logic out of the JSP page altogether: You create a page dedicated to presenting the error message and have the Controller determine which page to send the user to. If the remote system is available, the user gets the first page. If not, the Controller sends him to the error page.

This approach to exception processing has multiple benefits. The first comes from the fact that, on many pages, multiple types of exceptions must be handled. Having a single JSP page that detects all possible errors and presents a different message when each error happens can become complicated fast. Moving that logic into a Controller makes things easier to maintain: The logic is maintained in the Controller, and only the presentation is maintained in the JSP file.

Of course, another primary benefit of pulling the exception logic out of the main JSP pages is that it makes maintaining the JSP pages easier!

These benefits really extend to all forms of conditional processing. Here are some other examples:

- If different Views are required depending on what data is retrieved from a database or remote system (for example, products on sale versus products not on sale), the Controller component can make the decision about which page to present. This keeps the logic out of the JSP page itself.
- If your site changes based on either the time of day or the day of the week, that logic is easy to implement in the Controller. You simply have the Controller check the date and forward the user to the appropriate page.
- Sometimes a data entry process can span several pages, some of which are optional. An example of this is signing up for insurance: You need to be shown the data entry pages for dependents only if you choose family coverage. In cases like this, MVC makes it easy to control the flow of pages that are shown to the user. Trying to embed this logic into the JSP pages makes things much more complex.

THE ORIGINS OF THE MVC DESIGN PATTERN

It is widely agreed that the MVC pattern was originally popularized in Smalltalk-80. MVC was used to manage the GUI relationship in some of the earliest window-based GUIs that were developed at the Xerox PARC.

In researching the origins of the MVC pattern, I came across an archived posting from the Usenet group `comp.lang.smalltalk` from 1994. The posting read, in part:

I thought you might be interested in a 'bit of history' on origin of the Model-View-Controller paradigm.

Prof. Trygve Reenskaug is generally cited as being the creator of the MVC concept. He worked with the Smalltalk group at Xerox PARC as a visiting scientist in 78/79. During this stay at Xerox PARC he developed the MVC. I know him well and have talked to him about this. He confirms it, although stating that it was a collaborative effort at Xerox PARC.

[…]

Regards,

Carl

Carl P. Swensson, Senior Systems Eng.

I then looked up Prof. Reenskaug's current home page on the Internet. In his biography, he lists the creation of "Model-View-Controller, the world's first reusable object oriented framework, in 1979" as one of his career accomplishments.

I traded e-mails with Prof. Reenskaug while writing this book. He described his initial thoughts on MVC like this:

My first idea was to separate presentation from information because the information structure is reasonably stable while the way we need to see it and manipulate it varies with the task. (This idea stems from around 1970 when I first became interested in distributed systems.)

Prof. Reenskaug is now a Professor Emeritus from the University of Oslo in Norway. More recently, he contributed to the development of the Unified Modeling Language (UML) versions 1.4 and 2.0. The Object Management Group (`http://www.omg.org`) has honored him with a special award for his many contributions to the field of object-oriented design.

The Origins of Model 1 / Model 2

In documentation and general discussions surrounding Struts and MVC architectures for JSP processing, you'll frequently run across the terms *Model 1* and *Model 2* processing. Understanding where these terms came from will give you a better understanding of Struts and the design goals behind it. (As an added benefit, you'll also know what these people are talking about!)

The JSP Specification Version 0.92

Way back in the dark ages of JSP processing (October 7, 1998 to be exact) Sun Microsystems released the JSP 0.92 specification. A number of early container providers adopted this standard and put out JSP/servlet containers based on this version of the specification.

In this version of the specification, there was an overview that included a "JavaServer Pages Access Model(s)" section. In fact, there were only two models—appropriately named Model 1 and Model 2. Little did the authors of this document realize that the names they gave these two models would last long after the document itself was no longer available!

Model 1 described JSP processing as it was most commonly being done at that time. It showed the HTTP request being sent directly to a JSP file. All processing was done directly in the JSP (or in the beans it interacted with) and the HTTP response came directly from this JSP file.

Model 2 was different. It indicated that a servlet, rather than a JSP file, should receive the initial HTTP request. The servlet was to handle the processing tasks required for the request, and then store the information in a bean. The bean was then to be passed to the JSP file, which would pull information from it and render the HTTP response.

MVC and *Model-View-Controller* don't appear anywhere in the specification, but Model 2 was clearly based on the MVC architecture that is behind Struts today.

JavaWorld, 1999

At JavaWorld in December 1999, Govind Seshadri presented an article that clearly identified Model 2 as being the MVC architecture. This article was published on the JavaWorld site and is still there as of this writing (`http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc_p.html`).

In this article, Seshadri outlines how the Model 2 (or MVC) architecture is the best approach for development because it provides a clean separation of tasks between page designers and Java developers.

Enter Craig McClanahan, Jakarta Tomcat, and Struts

In March of 2000, Craig McClanahan launched the Struts project as a subproject of the Apache Jakarta project. Craig had been active in the Tomcat project and with Apache JServ (an early Java servlet implementation) before that.

Craig has been, without question, the leading architect and visionary behind Struts. His dedication and contributions to the open source Java community have been invaluable and critical to the success of both Tomcat and Struts.

How Struts Implements the Model 2 Pattern

Struts implements the Model 2 (or MVC) design pattern by providing an overall framework for development. This framework provides a variety of system services in addition to managing the HTTP request and response flow.

This frees the developer to focus on building discrete components and assembling them into an application using the Struts configuration file. This component approach simplifies development, debugging, and maintenance. It also provides natural boundaries for breaking the project between developers and HTML designers—although there will still be a requirement for them to work closely together.

To begin with, it's important to look at how to implement MVC as a Web application in general.

MVC Architecture for Web Applications

Implementing the MVC design pattern using a Web application is a pretty natural fit given the underlying request/response cycle of the HTTP protocol. The basic ideas are altogether independent of Struts.

Figure 2.1 provides a graphical illustration of how this works.

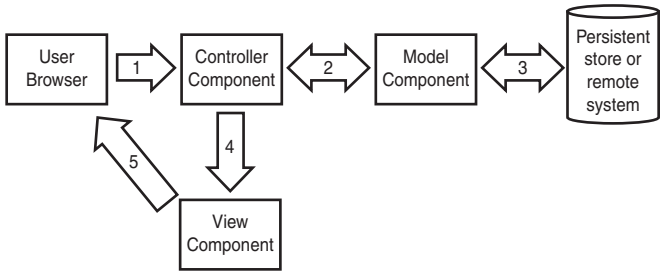

FIGURE 2.1 A MVC architecture diagram for a Web application.

Figure 2.1 shows the MVC pattern implemented for a Web application. Processing proceeds as follows:

1. The client browser issues an HTTP request to the application.
2. The Controller component receives the request. It then begins making decisions on how to proceed with processing based on the business rules encoded in it.
3. The Model components perform the actual interaction with the persistent data stores or remote systems that they manage interaction with.

4. Based on the results of processing and the data returned from the Model components, the Controller determines the View component that should be used to render the user display. Data is prepared for the View object to render.
5. The View component chosen renders the HTTP response to be sent to the user.

The following sections provide some details to the Model, View, and Controller elements that make up the MVC architecture. These sections provide information about Model, View, and Controller component design in general, as well as specific information about their implementations in Struts.

Model Components

Model components generally represent a business object or physical back-end system. For example, in a system that allows users to log in and maintain personal information, there might be a Model component that represents a User.

The User Model component would likely provide methods to access and modify the user's personal information such as his name, password, and so on. The specifics of the design would be driven by the application, but the critical ideas are

- Model components are used to access information from databases or remote systems for presentation to users.
- Model components should be designed to hide the implementation details for accessing that information.

Let's examine the first point: providing information for presentation. Model components provide access to the information from which the user presentation is built. Model components may store this information directly or simply provide convenient access to it.

In an application built using only a servlet container such as Tomcat, the Model component may simply be a Java bean providing a business view to some JDBC logic that maintains the information in a database. The rest of the application interacts with the Model component to read or write the information; only the Model component interacts with the database.

In an application built with the front-end using JavaServer Pages and the back-end accessed via Web services, Model components would be used to provide a business representation of the back-end system. The rest of the application would still access information through the Model, but only the Model would access the Web service.

It would be similar in a system in which an EJB server managed the back-end. Model components would manage access to the EJBs. In some designs, the Model components themselves may be EJBs.

This approach would be the same if the Model components managed access to a remote system. For example, in a system providing stock quotes, there might be a Model component that provides a programming representation of a stock quote. When accessed, the Stock Quote Model component would manage the access to the remote system providing the actual stock quote information.

One of the strengths of MVC should be pretty obvious now: the flexibility of model components! The idea of building a programming interface that hides the details of some back-end data store (or remote system) is extremely powerful and flexible.

It's also pretty well established. In the early 1970s (even before the original MVC ideas were developed at the PARC), Professor David Parnas published his influential work on what he termed *information hiding* in the article "On the Criteria to Be Used in Decomposing Systems into Modules," for the journal *Communications of the Association for Computing Machinery*. In it, he wrote:

We have tried to demonstrate by these examples that it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others.

Using these same ideas, Struts Model components hide implementation details for the remote systems and databases they interact with.

View Components

In MVC, the View components focus on just that: creating the presentation layer that the user views. They should contain little in the way of business logic or complex analysis.

Using the HTTP request/response model of a Web application, View components are almost always those components associated with the response. More specifically, View components in JSP and Struts are the JSP files that render the HTML to be sent to the user.

A primary advantage of JSP is its ability to combine HTML, JSP tags, and even Java scriptlets to build dynamic pages (or Views). But although this makes JSP easier and more flexible than just using servlets, it can be a challenge for HTML designers who need to build and maintain the look and feel of the pages.

It can be too easy for a Java developer to overuse java scriptlets and embed conditional business logic and looping directly in the JSP. This makes the HTML designers' job even more difficult. In fact, the embedding of scriptlets into JSP files is one of the primary criticisms of JSP.

MVC addresses exactly this problem. By segregating complex processing into the Model and Controller components, MVC allows the JSP files themselves to be smaller and simpler. This simplifies and speeds development, testing, and maintenance for both Java developers and HTML designers.

Struts takes this approach even further. In addition to providing the MVC architecture, it also provides a whole series of custom JSP tags that are used for constructing View components. These tags (with names such as `<html:text>` and `<logic:iterate>`) give both Java developers and HTML designers the ability to build functionality with tags that would otherwise require the use of scriptlets.

To summarize, Struts Views are part of an MVC framework that simplifies the creation of JSP pages through 1) separating complex logic and Java scriptlets from JSP pages and moving them to Controller and Model components; and 2) providing a series of custom tags to extend the functionality that can be accomplished without requiring the use of Java scriptlets.

Controller Components

Controller components direct all the action. Whenever the user submits a form, it's the controller that processes it. If the user asks for another page, the controller decides what to show her. The Controller component also collects the data from the Model components so that the View components have something to display.

In Struts, a Controller component performs several primary activities:

- Validates that the data entered by the user was valid
- Makes decisions about which Model components need to be accessed or updated and manages these activities
- Collects the data that the View component will need for display
- Makes decisions about how to recover when errors occur in processing the request or response
- One of the most important activities is deciding which View component should be displayed to the user

Conclusions

In this chapter, we covered the basics of the Model-View-Controller (MVC) design pattern, its history, and how it is implemented in Struts. We also covered the origins of the terms *Model 1* and *Model 2* JSP development. Struts is based on Model 2.

The Model-View-Controller design pattern is a time-proven architecture for building software that manages interaction with users (using Views), implements business rules that are dependent on user input (using Controllers), and relies on data that exists in a remote database or system (accessed using Model components). MVC originated at the Xerox PARC in the late 1970s, although its roots go back even further.

The terms *Model 1* and *Model 2* originated in the JSP 0.92 specification. The primary characteristics of Model 1 are

- HTTP requests are posted directly to `.jsp` files.
- The logic for directing program flow, for accessing databases and remote systems, and for building user displays are all embedded directly in JSP files.

The primary characteristics of Model 2 are

- HTTP requests are posted to Java servlets.
- The logic for directing program flow (Controllers) and for accessing databases and remote systems (Models) are implemented in Java servlets and classes. All user displays (Views) are built using JSP files.

Struts implements the MVC design pattern and is based on Model 2. Struts implements MVC using

- Model components that provide a programming model of back-end databases and remote systems and services
- View components that use JSP and Struts custom tags to build pages for user presentation
- Controller components that implement the business logic that defines the program flow