Before we start our journey into the internals of Spring MVC, we first need to understand the different layers of a web application. And we’ll begin that discussion with a brief introduction of the MVC pattern in general, including what it is and why should we use it. We will also cover some of the interfaces and classes provided by the Spring Framework to express the different parts of the MVC pattern.

After reviewing the MVC Pattern, we will go through the different layers in a web application and see what role each layer plays in the application. We will also explore how the Spring Framework can help us out in the different layers and how we can use it to our advantage.

The MVC Pattern

The Model View Controller pattern (MVC pattern) was first described by Trygve Reenskaug when he was working on Smalltalk at Xerox. At that time, the pattern was aimed at desktop applications. This pattern divides the presentation layer into different kinds of components. Each component has its own responsibilities. The view uses the model to render itself. Based on a user action, the view triggers the controller, which in turn updates the model. The model then notifies the view to (re)render itself (see Figure 3-1).

![MVC Pattern Diagram](image)

Figure 3-1. The MVC pattern

The MVC pattern is all about separation of concerns. As we mentioned previously, each component has its own role (see Table 3-1). Separation of concerns is important in the presentation layer because it helps us keep the different components clean. This way, we don’t burden the actual view with business
logic, navigation logic, and model data. Following this approach keeps everything nicely separated, which makes it easier to maintain and test our application.

**Table 3-1. MVC in Short**

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>The model is the data needed by the view so that it can be rendered. It might be an order placed or a list of books requested by a user.</td>
</tr>
<tr>
<td>View</td>
<td>The view is the actual implementation, and it uses the model to render itself in a web application. This could be a JSP or JSF page, but it could also be a PDF or XML representation of a page.</td>
</tr>
<tr>
<td>Controller</td>
<td>The controller is the component that is responsible for responding to the action the user takes, such as form submission or clicking a link. The controller updates the model and takes other actions needed, such as invoking a service method to place an order.</td>
</tr>
</tbody>
</table>

The classic implementation of the MVC pattern (as shown in Figure 3-1) involves the user triggering an action. This prompts the controller to update the model, which in turn pushes the changes back to the view. The view then updates itself with the updated data from the model. This is the ideal implementation of an MVC pattern, and it works very well in desktop applications based on Swing, for example. However, this approach is not feasible in a web environment due to the nature of the HTTP protocol. For a web application, the user typically initiates action by issuing a request. This prompts the app to update and render the view, which is sent back to the user. This means that we need a slightly different approach in a web environment. Instead of pushing the changes to the view, we need to pull the changes from the server.

This approach seems quite workable, but it isn’t as straightforward to apply in a web application as one might think. The Web (or HTTP) is stateless by design, so keeping a model around can be quite difficult. For the Web, the MVC pattern is implemented as a Model 2 architecture (see Figure 3-2). The difference between the original pattern (Model 1 was shown in Figure 3-1) and the modified pattern is that it incorporates a front controller that dispatches the incoming requests to other controllers. These controllers handle the incoming request, return the model, and select the view.

---