1

The Model-View-Controller Pattern

The first three chapters of this book give you the foundation on which you will build up your knowledge about the ASP.NET MVC framework.

The ASP.NET MVC framework, unlike traditional ASP.NET programming, incorporates the usage of the Model-View-Controller (MVC) pattern. So, before you look at how the framework works, it's critical that you understand how the pattern on which it's based works. Then, in the following chapters, you learn how Microsoft implements that pattern in its library and how to write your first ASP.NET MVC Web application.

In this chapter you learn:

- The principles behind this pattern
- How the MVC pattern works
- The other advantages that it provides

The History of MVC

The Model-View-Controller (MVC) pattern, even if it gained popularity only recently, mainly thanks to Ruby on Rails and to the ASP.NET MVC framework, is not new. It was invented back in the 1970s by Trygve Reenskaug, a Norwegian computer scientist. The original pattern, named Thing-Model-View-Editor, was invented to solve a problem of the shipbuilding industry in Norway. They had the need to build information systems that could easily fit into existing organizations and be adapted for their continual development.

The pattern was later renamed to Model-View-Controller when Trygve worked with the SmallTalk group at Xerox PARC. A year later other researchers at Xerox PARC implemented a version of MVC and included it into the Smalltalk-80 class library, thus making Smalltalk the first programming language with native support for MVC.

Later, many other GUI frameworks took inspiration from the SmallTalk implementation, the most notable of which is Cocoa, the main application programming environment for Apple Mac OS X, which is based on its precursors OpenStep and NextStep, both of which encouraged the use of the MVC pattern.

On the Web application side, the first framework based on MVC was Apache Struts, a framework for building Java Enterprise Edition Web applications. Then came Ruby on Rails and the many other MVC-based application frameworks.

The Principles of MVC

Web applications based on ASP.NET that use the WebForm approach sometimes commingle database code, page design code, and control flow code. In practice, what happens is that, unless these elements are separated, larger applications become difficult to maintain.

One way to separate elements in a software application is to use the Model-View-Controller pattern. This paradigm is based on breaking an application into three parts, loosely related to the three usual tiers of a three-tier application:

- **Model** — The model is the part of the application that is responsible for retrieving data from the database, converting it to objects, and applying domain-specific elaboration to it.
- **Controller** — The controller orchestrates the operations. It's responsible for handling and validating user input, calling the model, choosing which view to render, and handing it the data to be displayed to the user.
- **View** — The view is the front end of the application. It shows to the user the results of the operation and the data retrieved.

Separating the application in loosely coupled components brings many advantages that help in managing applications that need to evolve to keep up with the quick change of requirements typical of today's IT projects.

At this point you might be wondering why the pattern is named "MVC" but the components were explained here (and will be throughout the book) in the order of Model-Controller-View. The reason is simple: The most logical way to explain how the components work is by starting from the model, then going through the controller, and finally reaching the view. And "MCV" would not have been nearly as appealing a name to the ear as "MVC."

But before looking at which specific advantages this pattern brings, in the next section you learn about the flow of an application based on MVC.

How the MVC Pattern Flows

As you already know, the core principle of the MVC pattern is a strict separation of concerns among the parts of the application. Figure 1-1 represents the MVC pattern as applied to Web applications.

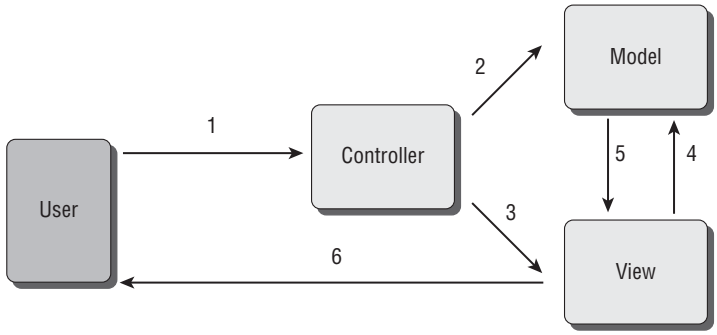

Figure 1-1

This implementation is also referred to as Front Controller or Model 2, from the name given to the first widespread implementation of this pattern in Apache Struts.

> *The name is Model 2 as opposed to Model 1, which refers to the old-style development of Web applications, where the page handled everything from the validation of the user input, to the data retrieval, to the rendering of the markup sent to the browser. Model 1 is pretty much how most of the Web applications developed using the WebForm paradigm.*
>
> *Model 2 and Model 1 are names of different flavors of the Model-View-Controller pattern; they are not different types of models used within the pattern. Outside of this section that discusses the flavors of Model-View-Controller, you will generally see the term "model" used to refer to the model within the Model-View-Controller pattern.*

The flow of the request follows this path:

1. The user interacts with the browser, typing a URL in the address bar or clicking a link or a button on a Web page. This starts the request (Step 1 in Figure 1-1).
2. The request hits the controller, which performs some validations on the user input ("Is the user ID a valid one? Is it a number or has someone passed in a string?") and delegates the execution of the request to the model (Step 2 in Figure 1-1).
3. Based on the results returned by the model, the controller goes on orchestrating the processing. It chooses the correct view that needs to be rendered to the user and calls it (Step 3 in Figure 1-1). For example, if the model says that the product the user wants to buy is not available, the view rendered will be different from the one that will be sent back if the product was available.
4. The view then calls the model that has been selected by the controller and asks it for the data needed to populate the page (Steps 4 and 5 in Figure 1-1). During this phase no logic or processing decision operations are performed, only data retrieval.
5. As the last step, the view receives the data needed, formats it in the appropriate manner, and sends the response back to the user (Step 6 in Figure 1-1).

Each part of the application performs well-defined tasks and communicates with the other components only to pass the results of their operations.

Reading on blogs, you might have heard about the Routing component. This component is responsible for routing the requests to the appropriate controller. This component is not part of the MVC pattern itself, but it's one of the implementation details of ASP.NET MVC. You read a little about it in Chapter 2, and get a detailed explanation in Chapter 7.

The Passive View

Some people started to think that separation of concerns can be pushed even further than in the Front Controller model and that the dependency of the view on the model can be removed. Doing this makes the three parts of the application completely independent and reduces the number of operations on the data storage during the rendering phase of the view.

This operational flow is depicted in Figure 1-2.

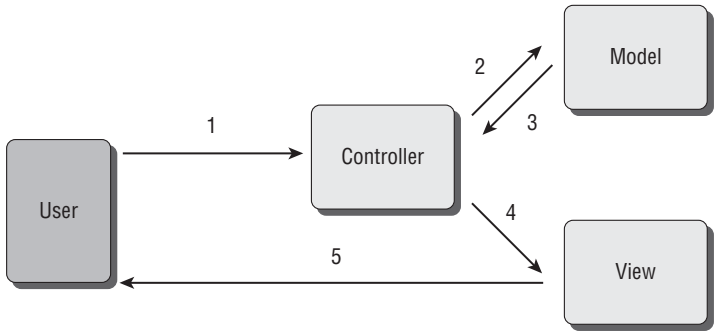

Figure 1-2

In this flavor of MVC, the process flow is the same as the traditional MVC for the first two steps, but then it changes radically:

1. As before, the user, interacting with the browser, starts the request (Step 1 in Figure 1-2).
2. The controller receives the request, validates it, and delegates the execution to the model (Step 2 in Figure 1-2).
3. The model hits the database and returns the data retrieved back to the controller (Step 3 in Figure 1-2).
4. At this point the controller selects the view and passes to it the data previously retrieved by the model. Actually, it doesn't pass exactly the same data, but it rearranges the data in a way that is better suited for the view (Step 4 in Figure 1-2).
5. As the last step, the view renders the output and sends it back to the user (Step 5 in Figure 1-2).

The main difference between the two flavors is that, whereas in the traditional MVC the view interacts with the model to get the data and then reformats it to render the Web page, in this flavor the model sends all the data to the controller, which then, again with the help of the model, reformats it and sends it to the view, whose only role is to render the data in the output sent to the user. Because of this passive role of the view, this flavor of MVC is called "Passive View."

The Presentation Model

Later, in Chapter 4, you learn about the model part and the Presentation Model in detail, but to help you better understand the differences between the two flavors of MVC and why the Passive View is usually a better option, this section gives you a quick introduction of the Presentation Model.

The model is an interface to the objects stored in the database. The graph in which these object are organized represents the connections that they have in real life: A customer places many orders, and each order is made up of many items, each with its name and quantity (see Figure 1-3).

Figure 1-3

Imagine that you want to display on a Web page the list of customers who ordered a long-sleeved shirt last week. If the view retrieves the data organized in the hierarchy of Figure 1-3, it must know how to navigate the object graph to reformat the data in a plan table. But this means that the view has to know about how the data is organized inside the model. This is not good because it means that if the organization of the model changes, the view has to change as well, and this is something you want to avoid when using the MVC pattern. (One of the reasons for using MVC in the first place is to separate these responsibilities so that you can increase maintainability by managing these matters separately.)

When using the Passive View flavor of MVC, the controller reformats the data with the help of the Presentation Model. The Presentation Model is the only part of the application that knows how to transform the hierarchical object graph in the plan object (see Figure 1-4) needed to display a list of customers with the shirts they ordered.

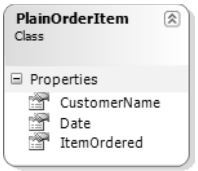

Figure 1-4

> **The view must only consume data, eventually formatting it, but the view should never generate (retrieve or transform) new data.**

The Model-View-Controller is not the only pattern that tries to split the responsibilities across different parts of the application. Before explaining what the advantages of using the MVC pattern are, the next section quickly introduces the Model-View-Presenter pattern.

The Model-View-Presenter Pattern

The other pattern that has gained a lot of popularity in recent years is the Model-View-Presenter (MVP) pattern. As the name itself suggests, the MVP pattern is quite similar to MVC. In fact, some developers joke about that, saying that the only difference between the MVC and the MVP pattern is the letter P instead of the C. And to an extent, they are right. The first noticeable difference between the controller and the presenter is only their name.

The MVP also differs from the MVC pattern in two other ways:

- Where the requests hit the system
- How the parts are wired up together

The flow of processing is also different from the one you saw in the previous section:

1. The user interacts directly with the view (Step 1 in Figure 1-5).
2. The view, raising an event, notifies the presenter that something happened (Step 2), and then the presenter accesses the properties that the view exposes through its interface Iview (Step 3 in Figure 1-5). These properties are wrappers to the actual UI elements of the view.
3. The presenter then calls the model (Step 4), which then returns the results (Step 5).
4. The presenter transforms the data and then sets the values in the UI, always through the IView interface (Step 6 in Figure 1-5).
5. The output is the returned to the user (Step 7).

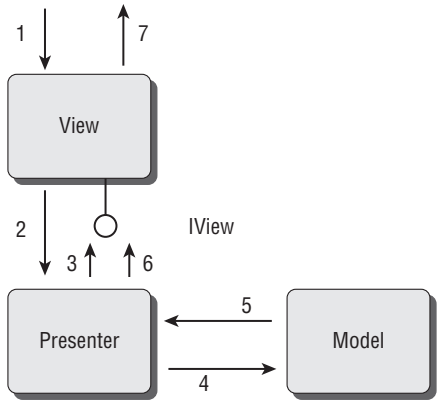

Figure 1-5

In the Model-View-Presenter pattern the view is even more "passive" than the one in the MVC. The view raises events, but it is up to the presenter to read and set the values of the UI elements. In fact, this pattern has been designed as an enhancement over the MVC, to try and make the view even dumber to make it easier to swap views.

So why did Microsoft decide to built its new Web application framework based on the MVC pattern and not based on MVP? The problem with MVP is that the wiring of the view and the presenter is more complicated than in MVC. Every view has its own interface, and because it is specific to the contents of the page, its creation cannot be easily delegated to the framework, but must be created by the developer.

The Web Client Software Factory, developed by the Pattern & Practices team at Microsoft, addresses the problem of automatically creating all the components needed to implement the MVP pattern. It's a graphical tool that allows the developer to design the interactions and the UI elements that need to be handled by the presenter. But this is a development helper, not a full-blown framework, and the kind of developer that the MVC framework has been mainly designed for prefers simpler solutions that can be managed with "stylistically nice" code over more complicated solutions that need the aid of a tool to be implemented.

Now that you understand what the pattern is and how it works, in the following sections you learn the advantages of using the Model-View-Controller pattern over the traditional WebForm approach.

Advantages of MVC over Traditional Web Development

The MVC paradigm, because of the great emphasis it puts on separating responsibilities, brings several advantages to Web development: Unit testing and test driven development are probably the most important advantages, and probably the factors that drove the creation of the ASP.NET MVC

framework. It also enables an approach called *Interface First*, a design methodology that advocates the design of the UI before the business logic and data access.

The next section gives a quick overview of why the MVC pattern enables these three scenarios.

Unit Testing

The first and most important practice that is facilitated by MVC is unit testing. You read more about unit testing in Chapter 8, but here you get a quick introduction to what it is and why it is important. Unit testing is about isolating a specific "unit" of your code to prove its correctness. In object-oriented programming, a unit, which is the smallest part of the application, is a method. Each test case must be autonomous and repeatable, so that it can be easily automated.

Why is isolating a specific "unit" so important? Imagine that you want to prove the correctness of a method that renders a list of all the customers who live in a certain city and who ordered a specific kind of shirt. This method hits the database to retrieve the list based on the city, then filters out all the ones who didn't buy the shirt, orders the results, and formats the output. This test can fail because the filtering part, which is the code you want to test, went wrong. But it can also fail for external reasons that are out of the control of the method you want to test. The data access layer might have some problem, the database administrator might have dropped a stored procedure by mistake, or there even could have been a network problem that prevented the communication with the database.

With traditional Web application development, it can be difficult to isolate single parts of an application because, most of the time, the code that accesses the data repository is not formally separated from the code that transforms that data or that manages the process flow.

With the MVC pattern, you can replace the real implementation of the model with an implementation made on purpose for the test, for example with a fake model that always returns 10 customers. This model will never fail, so the only reason for the preceding test to fail is that the filtering algorithm fails.

Furthermore, with the WebForm paradigm, the request processing is tightly tied to the Web server runtime. With MVC, instead, the parts that are doing most of the job, the controller and the model, don't interact directly with the Web server runtime, allowing tests also to be executed outside the Web server container.

Another reason why this isolation and repeatability is important is that all test cases need to run automatically, as a way to ensure that the code still works even after major changes, and that the implementation of new features doesn't break the ones that have already been implemented and tested.

The loose coupling and the strict separation of concerns also makes it easier to adopt practices coming from the Agile world such as test driven development or the UI First approach. In the following sections, you learn about these practices.

Test Driven Development

The name test driven development, even if it contains the word "test," is not a testing methodology but a design and development methodology. It consists of writing a test case that covers the feature that you want to implement and then writing the code necessary for the feature to pass the test.

When adopting this development methodology, you tend to run the entire automated suite of tests a lot of times, and unit testing will save you precious time. Restoring the database to a known state and hitting the database for each test case (could be thousands of times) is much more time-consuming than using fake models that return predefined sets of data (as you do with unit testing).

If you like this approach, you can read about it in more detail in Chapters 8 and 9.

The Interface First Approach

Another design methodology that can benefit from the use of the MVC pattern is the one proposed by the book *Getting Real*, written by the software company called 37signals, the creators of some popular Web 2.0 project management applications like Basecamp and Campfire, and of the Web application framework that probably started all this interest around MVC: Ruby on Rails.

They found that using a top-down approach works better for Web applications than the usual bottom-up approach. Starting from the database and then going up to the user interface adds too much inertia to your development process. When you show the UI to the end users, your application has already been developed, and even small changes in the requirements can raise the costs of development a lot.

Starting from the UI and going down instead allows you to show the end users the application and to gather immediate feedback. Changing a requirement at this stage of the process is just a matter of changing some HTML and some code in the controller. This is possible because the complex and expensive parts of the applications are all in the model, which at this stage is not implemented yet, but only loosely sketched out.

Instead of starting from the model or the controller and then going up to the view as in other usual methodologies, with the Interface First approach you first design the view, then work on the controller, and finally, you design and develop the (Domain) model of your application.

We won't talk about this development methodology, because it's not as formalized and widespread as the previous two. If you are interested in this kind of approach, I encourage you to visit the *Getting Real* Web site (`http://gettingreal.37signals.com`) and buy the book or read it online.

Summary

The Model-View-Controller is a pattern that encourages loose coupling and separation of duties to make the design of the application flexible and quickly adaptable to changes and to facilitate the usage of the methodologies typical of the Agile world.

In this chapter you learned:

- That the Model-View-Controller pattern was created in the late 1970s with SmallTalk and gained popularity in the new century thanks to Struts and Ruby on Rails
- That Model-View-Controller splits the application into three components:
  - Model, which encapsulates the operation with the database

- View, which is responsible for the rendering of the final output
- Controller, which orchestrates all the operations

- What the process flow of the MVC pattern is and how the traditional implementation differs from the Passive View one
- How to use a Presentation Model
- How the MVC compares to the Model-View-Presenter pattern
- That the MVC pattern facilitates the adoption of Agile practices like unit testing, test driven development, and the Interface First approach

With the knowledge acquired about MVC, in the next chapter you learn more specifically how the MVC pattern works using the ASP.NET MVC framework, but before moving on try to answer to the following questions.

Exercises

1. Can you list the three components of the MVC pattern and the roles of each of them?
2. What's the role of the Presentation Model?
3. What are the differences between the MVC and the MVP patterns?