PART I

ASP.NET MVC fundamentals

CHAPTER 1

ASP.NET MVC controllers

They always say time changes things, but you actually have to change them yourself.

—Andy Warhol

I think ASP.NET Web Forms started getting old the day that Ajax conquered the masses. As some have said, Ajax has been the poisonous arrow shot in the heel of ASP.NET—another Achilles. Ajax made getting more and more control over HTML and client-side code a true necessity. Over time, this led to different architectures and made ASP.NET Web Forms a little less up to the task with each passing day.

Applied to the existing ASP.NET runtime, the MVC pattern produced a new framework—ASP.NET MVC—that aligns web development to the needs of developers today.

In ASP.NET MVC, each request results in the execution of an action—ultimately, a method on a specific class. The results of executing the action are passed down to the view subsystem along with a view template. The results and template are then used to build the final response for the browser. Users don't point the browser to a page, they just place a request. Doesn't that sound like a big change?

Unlike Web Forms, ASP.NET MVC is made of various layers of code connected together but not intertwined and not forming a single monolithic block. For this reason, it's easy to replace any of these layers with custom components that enhance the maintainability as well as the testability of the solution. With ASP.NET MVC, you gain total control over the markup and can apply styles and inject script code at will using the JavaScript frameworks that you like most.

Based on the same run-time environment as Web Forms, ASP.NET MVC pushes a web-adapted implementation of the classic Model-View-Controller pattern and makes developing web applications a significantly different experience. In this chapter, you'll discover the role and structure of the controller—the foundation of ASP.NET MVC applications—and how requests are routed to controllers.

Although you might decide to keep using Web Forms, for today's web development, ASP.NET MVC is a much better choice. You don't need to invest a huge amount of time, but you need to understand exactly what's going on and the philosophy behind MVC. If you do that, any investment you make will pay you back sooner than you expect.

Note This book is based on ASP.NET MVC 5. This version of ASP.NET MVC is backward compatible with the previous versions. This means that you can install both versions side by side on the same computer and play with the new version without affecting any existing MVC code that you might have already.

Routing incoming requests

Originally, the entire ASP.NET platform was developed around the idea of serving requests for physical pages. It turns out that most URLs used within an ASP.NET application are made of two parts: the path to the physical webpage that contains the logic, and some data stuffed in the query string to provide parameters. This approach has worked for a few years, and it still works today. The ASP.NET run-time environment, however, doesn't limit you to just calling into resources identified by a specific location and file. By writing an ad hoc HTTP handler and binding it to a URL, you can use ASP.NET to execute code in response to a request regardless of the dependencies on physical files. This is just one of the aspects that most distinguishes ASP.NET MVC from ASP.NET Web Forms. Let's briefly see how to simulate the ASP.NET MVC behavior with an HTTP handler.

Note In software, the term URI (which stands for *Uniform Resource Identifier*) is used to refer to a resource by location or a name. When the URI identifies the resource by location, it's called a *URL*, or *Uniform Resource Locator*. When the URI identifies a resource by name, it becomes a *URN*, or *Uniform Resource Name*. In this regard, ASP.NET MVC is designed to deal with more generic URIs, whereas ASP.NET Web Forms was designed to deal with location-aware physical resources.

Simulating the ASP.NET MVC runtime

Let's build a simple ASP.NET Web Forms application and use HTTP handlers to figure out the internal mechanics of ASP.NET MVC applications. You can start from the basic ASP.NET Web Forms application you get from your Microsoft Visual Studio project manager.

Defining the syntax of recognized URLs

In a world in which requested URLs don't necessarily match up with physical files on the web server, the first step to take is listing which URLs are meaningful for the application. To avoid being too specific, let's assume that you support only a few fixed URLs, each mapped to an HTTP handler component. The following code snippet shows the changes required to be made to the default *web.config* file:

```
<httpHandlers>
   <add verb="*"
        path="home/test/*"
        type="MvcEmule.Components.MvcEmuleHandler" />
</httpHandlers>
```

Whenever the application receives a request that matches the specified URL, it will pass it on to the specified handler.

Defining the behavior of the HTTP handler

In ASP.NET, an HTTP handler is a component that implements the *IHttpHandler* interface. The interface is simple and consists of two members, as shown here:

```
public class MvcEmuleHandler : IHttpHandler
{
   public void ProcessRequest(HttpContext context)
   {
      // Logic goes here
      ...
   }

   public Boolean IsReusable
   {
      get { return false; }
   }
}
```

Most of the time, an HTTP handler has a hardcoded behavior influenced only by some input data passed via the query string. However, nothing prevents us from using the handler as an abstract factory for adding one more level of indirection. The handler, in fact, can use information from the request to determine an external component to call to actually serve the request. In this way, a single HTTP handler can serve a variety of requests and just dispatch the call among a few more specialized components.

The HTTP handler could parse out the URL in tokens and use that information to identify the class and the method to invoke. Here's an example of how it could work:

```
public void ProcessRequest(HttpContext context)
{
    // Parse out the URL and extract controller, action, and parameter
    var segments = context.Request.Url.Segments;
    var controller = segments[1].TrimEnd('/');
    var action = segments[2].TrimEnd('/');
    var param1 = segments[3].TrimEnd('/');

    // Complete controller class name with suffix and (default) namespace
    var fullName = String.Format("{0}.{1}Controller",
                       this.GetType().Namespace, controller);
    var controllerType = Type.GetType(fullName, true, true);
```