

# 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);
}
```

```

// Get an instance of the controller
var instance = Activator.CreateInstance(controllerType);

// Invoke the action method on the controller instance
var methodInfo = controllerType.GetMethod(action,
    BindingFlags.Instance |
    BindingFlags.IgnoreCase |
    BindingFlags.Public);
var result = String.Empty;
if (methodInfo.GetParameters().Length == 0)
{
    result = methodInfo.Invoke(instance, null) as String;
}
else
{
    result = methodInfo.Invoke(instance, new Object[] { param1 }) as String;
}

// Write out results
context.Response.Write(result);
}

```

The preceding code assumes that the first token in the URL after the server name contains the key information to identify the specialized component that will serve the request. The second token refers to the name of the method to call on this component. Finally, the third token indicates a parameter to pass.

## Invoking the HTTP handler

Given a URL such as *home/test/\**, it turns out that *home* identifies the class, *test* identifies the methods, and whatever trails is the parameter. The name of the class is further worked out and extended to include a namespace and a suffix. According to the example, the final class name is *MvcEmule.Components.HomeController*. This class is expected to be available to the application. The class is also expected to expose a method named *Test*, as shown here:

```

namespace MvcEmule.Components
{
    public class HomeController
    {
        public String Test(Object param1)
        {
            var message = "<html><h1>Got it! You passed '{0}'</h1></html>";
            return String.Format(message, param1);
        }
    }
}

```

Figure 1-1 shows the effect of invoking a page-agnostic URL in an ASP.NET Web Forms application.



**FIGURE 1-1** Processing page-agnostic URLs in ASP.NET Web Forms.

This simple example demonstrates the basic mechanics used by ASP.NET MVC. The specialized component that serves a request is the controller. The controller is a class with just methods and no state. A unique system-level HTTP handler takes care of dispatching incoming requests to a specific controller class so that the instance of the class executes a given action method and produces a response.

What about the scheme of URLs? In this example, you just use a hardcoded URL. In ASP.NET MVC, you have a very flexible syntax that you can use to express those URLs that the application recognizes. In addition, a new system component in the run-time pipeline intercepts requests, processes the URL, and triggers the ASP.NET MVC HTTP handler. This component is the URL Routing HTTP module.

## The URL routing HTTP module

The URL routing HTTP module processes incoming requests by looking at the URLs and dispatching them to the most appropriate executor. The URL routing HTTP module supersedes the URL rewriting feature of older versions of ASP.NET. At its core, URL rewriting consists of hooking up a request, parsing the original URL, and instructing the HTTP run-time environment to serve a “possibly related but different” URL.

### Superseding URL rewriting

URL rewriting comes into play if you need to make tradeoffs between needing human-readable and search engine optimization (SEO)-friendly URLs and needing to programmatically deal with tons of URLs. For example, consider the following URL:

`http://northwind.com/news.aspx?id=1234`

The *news.aspx* page incorporates any logic required to retrieve, format, and display any given news. The ID for the specific news to retrieve is provided via a parameter on the query string. As a developer, implementing the page couldn't be easier; you get the query string parameter, run the query, and create the HTML. As a user or for a search engine, by simply looking at the URL you can't really understand the intent of the page, and you aren't likely to remember the address easily enough to pass it around.

URL rewriting helps you in two ways. First, it makes it possible for developers to use a generic front-end page, such as *news.aspx*, to display related content. Second, it also makes it possible for users to request friendly URLs that will be programmatically mapped to less intuitive but easier-to-manage URLs. In a nutshell, URL rewriting exists to decouple the requested URL from the physical webpage that serves the requests.

In the latest version of ASP.NET 4 Web Forms, you can use URL routing to match incoming URLs to other URLs without incurring the costs of HTTP 302 redirects. Conversely, in ASP.NET MVC, URL routing serves the purpose of mapping incoming URLs to a controller class and an action method.



**Note** Originally developed as an ASP.NET MVC component, the URL routing module is now a native part of the ASP.NET platform and, as mentioned, offers its services to both ASP.NET MVC and ASP.NET Web Forms applications, though through a slightly different API.

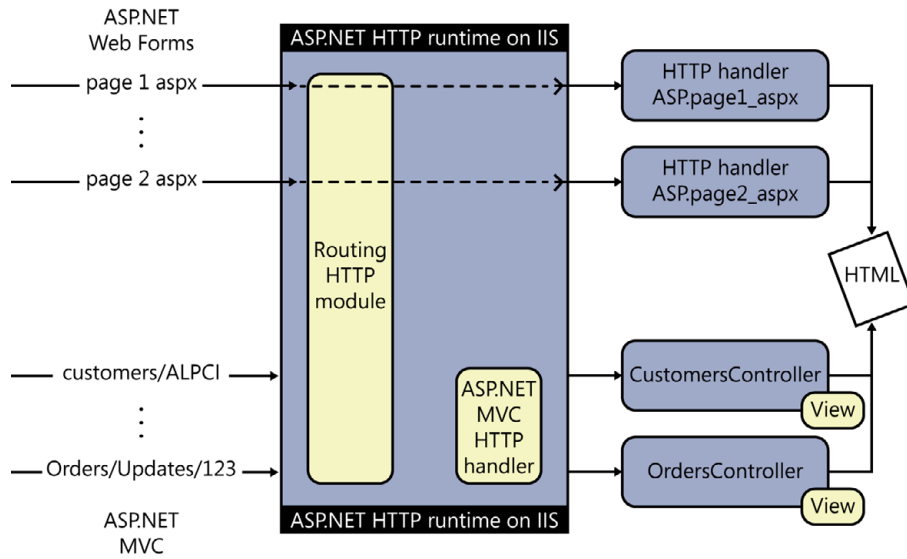
## Routing the requests

What happens exactly when a request knocks at the Internet Information Services (IIS) gate? Figure 1-2 gives you an overall picture of the various steps involved and how things work differently in ASP.NET MVC and ASP.NET Web Forms applications.

The URL routing module intercepts any requests for the application that could not be served otherwise by IIS. If the URL refers to a physical file (for example, an ASPX file), the routing module ignores the request, unless it's otherwise configured. The request then falls down to the classic ASP.NET machinery to be processed as usual, in terms of a page handler.

Otherwise, the URL routing module attempts to match the URL of the request to any of the application-defined routes. If a match is found, the request goes into the ASP.NET MVC space to be processed in terms of a call to a controller class. If no match is found, the request will be served by the standard ASP.NET runtime in the best possible way and likely results in an HTTP 404 error.

In the end, only requests that match predefined URL patterns (also known as routes) are allowed to enjoy the ASP.NET MVC runtime. All such requests are routed to a common HTTP handler that instantiates a controller class and invokes a defined method on it. Next, the controller method, in turn, selects a view component to generate the actual response.



**FIGURE 1-2** The role of the routing module in ASP.NET MVC.

## The internal structure of the URL routing module

In terms of implementation, I should note that the URL routing engine is an HTTP module that wires up the *PostResolveRequestCache* event. The event fires right after checking that no response for the request is available in the ASP.NET cache.

The HTTP module matches the requested URL to one of the user-defined URL routes and sets the HTTP context to using the ASP.NET MVC standard HTTP handler to serve the request. As a developer, you're not likely to deal with the URL routing module directly. The module is provided by the system and you don't need to perform any specific form of configuration. Instead, you are responsible for providing the routes that your application supports and that the routing module will actually consume.

## Application routes

By design, an ASP.NET MVC application is not forced to depend on physical pages. In ASP.NET MVC, users place requests for acting on resources. The framework, however, doesn't mandate the syntax for describing resources and actions. I'm aware that the expression "acting on resources" will likely make you think of Representational State Transfer (REST). And, of course, you will not be too far off the mark in thinking so.

Although you can definitely use a pure REST approach within an ASP.NET MVC application, I would rather say that ASP.NET MVC is loosely REST-oriented in that it does acknowledge concepts such as resource and action, but it leaves you free to use your own syntax to express and implement resources and actions. As an example, in a pure REST solution you would use HTTP verbs to express actions—GET, POST, PUT, and DELETE—and the URL to identify the resource. Implementing a pure REST solution in ASP.NET MVC is possible but requires some extra work on your part.

The default behavior in ASP.NET MVC is using custom URLs where you make yourself responsible for the syntax through which actions and resources are specified. This syntax is expressed through a collection of URL patterns, also known as routes.

## URL patterns and routes

A route is a pattern-matching string that represents the absolute path of a URL—namely, the URL string without protocol, server, and port information. A route might be a constant string, but it will more likely contain a few placeholders. Here’s a sample route:

```
/home/test
```

The route is a constant string and is matched only by URLs whose absolute path is `/home/test`. Most of the time, however, you deal with parametric routes that incorporate one or more placeholders. Here are a couple of examples:

```
{resource}/{action}  
/Customer/{action}
```

Both routes are matched by any URLs that contain exactly two segments. The latter, though, requires that the first segment equals the string “Customer”. The former, instead, doesn’t pose specific constraints on the content of the segments.

Often referred to as a *URL parameter*, a placeholder is a name enclosed in curly brackets { }. You can have multiple placeholders in a route as long as they are separated by a constant or delimiter. The forward slash (/) character acts as a delimiter between the various parts of the route. The name of the placeholder (for example, action) is the key that your code will use to programmatically retrieve the content of the corresponding segment from the actual URL.

Here’s the default route for an ASP.NET MVC application:

```
{controller}/{action}/{id}
```

In this case, the sample route contains three placeholders separated by the delimiter. A URL that matches the preceding route is the following:

```
/Customers/Edit/ALFKI
```

You can add as many routes as you want with as many placeholders as appropriate. You can even remove the default route.



## Defining application routes

Routes for an application are usually registered in the `global.asax` file, and they are processed at the application startup. Let's have a look at the section of the `global.asax` file that deals with routes:

```
public class MvcApplication : HttpApplication
{
    protected void Application_Start()
    {
        RouteConfig.RegisterRoutes(RouteTable.Routes);

        // Other code
        ...
    }
}
```

*RegisterRoutes* is a method on the *RouteConfig* class defined in a separate folder, usually named `App_Start`. (You can rename the folder at will, though.) Here's the implementation of the class:

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        // Other code
        ...

        // Listing routes
        routes.MapRoute(
            "Default",
            "{controller}/{action}/{id}",
            new {
                controller = "Home",
                action = "Index",
                id = UrlParameter.Optional
            });
    }
}
```

As you can see, the *Application\_Start* event handler calls into a public static method named *RegisterRoutes* that lists all routes. Note that the name of the *RegisterRoutes* method as well as the prototype is arbitrary and you can change it if there's a valid reason.

Supported routes must be added to a static collection of *Route* objects managed by ASP.NET MVC. This collection is *RouteTable.Routes*. You typically use the handy *MapRoute* method to populate the collection. The *MapRoute* method offers a variety of overloads and works well most of the time. However, it doesn't let you configure every possible aspect of a route object. If there's something you need to set on a route that *MapRoute* doesn't support, you might want to resort to the following code:

```
// Create a new route and add it to the system collection
var route = new Route(...);
RouteTable.Routes.Add("NameOfTheRoute", route);
```

A route is characterized by a few attributes, such as name, URL pattern, default values, constraints, data tokens, and a route handler. The attributes you set most often are name, URL pattern, and default values. Let's expand on the code you get for the default route:

```
routes.MapRoute(
    "Default",
    "{controller}/{action}/{id}",
    new {
        controller = "Home",
        action = "Index",
        id = UrlParameter.Optional
    });
```

The first parameter is the name of the route; each route should have a unique name. The second parameter is the URL pattern. The third parameter is an object that specifies default values for the URL parameters.

Note that a URL can match the pattern even in an incomplete form. Let's consider the root URL—*http://yourserver.com*. At first sight, such a URL wouldn't match the route. However, if a default value is specified for a URL parameter, the segment is considered optional. As a result, for the preceding example, when you request the root URL, the request is resolved by invoking the method *Index* on the *Home* controller.

## Processing routes

The ASP.NET URL routing module employs a number of rules when trying to match an incoming requested URL to a defined route. The most important rule is that routes must be checked in the order in which they were registered in *global.asax*.

To ensure that routes are processed in the correct order, you must list them from the most specific to the least specific. In any case, keep in mind that the search for a matching route always ends at the first match. This means that just adding a new route at the bottom of the list might not work and might also cause you a bit of trouble. In addition, be aware that placing a catch-all pattern at the top of the list will make any other patterns, no matter how specific, pass unnoticed.

Beyond order of appearance, other factors affect the process of matching URLs to routes. As mentioned, one is the set of default values that you might have provided for a route. Default values are simply values that are automatically assigned to defined placeholders in case the URL doesn't provide specific values. Consider the following two routes:

```
{Orders}/{Year}/{Month}
{Orders}/{Year}
```

If in the first route you assign default values for both *{Year}* and *{Month}*, the second route will never be evaluated because, thanks to the default values, the first route is always a match regardless of whether the URL specifies a year and a month.

A trailing forward slash (/) is also a pitfall. The routes `{Orders}/{Year}` and `{Orders}/{Year}/` are two very different things. One won't match to the other, even though logically, at least from a user's perspective, you'd expect them to.

Another factor that influences the URL-to-route match is the list of constraints that you optionally define for a route. A route constraint is an additional condition that a given URL parameter must fulfill to make the URL match the route. The URL not only should be compatible with the URL pattern, it also needs to contain compatible data. A constraint can be defined in various ways, including through a regular expression. Here's a sample route with constraints:

```
routes.MapRoute(
    "ProductInfo",
    "{controller}/{productId}/{locale}",
    new { controller = "Product", action = "Index", locale="en-us" },
    new { productId = @"\d{8}",
        locale = "[a-z]{2}-[a-z]{2}" });
```

In particular, the route requires that the *productId* placeholder must be a numeric sequence of exactly eight digits, whereas the local placeholder must be a pair of two-letter strings separated by a dash. Constraints don't ensure that all invalid product IDs and locale codes are stopped at the gate, but at least they cut off a good deal of work.

## Route handler

The route defines a bare-minimum set of rules, according to which the routing module decides whether the incoming request URL is acceptable to the application. The component that ultimately decides how to remap the requested URL is another one entirely. Precisely, it is the route handler. The route handler is the object that processes any requests that match a given route. Its sole purpose in life is returning the HTTP handler that will actually serve any matching request.

Technically speaking, a route handler is a class that implements the *IRouteHandler* interface. The interface is defined as shown here:

```
public interface IRouteHandler
{
    IHttpHandler GetHandler(HttpContext requestContext);
}
```

Defined in the *System.Web.Routing* namespace, the *RequestContext* class encapsulates the HTTP context of the request plus any route-specific information available, such as the *Route* object itself, URL parameters, and constraints. This data is grouped into a *RouteData* object. Here's the signature of the *RequestContext* class:

```
public class RequestContext
{
    public RequestContext(HttpContextBase httpContext, RouteData routeData);

    // Properties
    public HttpContextBase HttpContext { get; set; }
    public RouteData RouteData { get; set; }
}
```

The ASP.NET MVC framework doesn't offer many built-in route handlers, and this is probably a sign that the need to use a custom route handler is not that common. Yet, the extensibility point exists and, in case of need, you can take advantage of it. I'll return to custom route handlers and provide an example later in the chapter.

## Handling requests for physical files

Another configurable aspect of the routing system that contributes to a successful URL-to-route matching is whether the routing system has to handle requests that match a physical file.

By default, the ASP.NET routing system ignores requests whose URL can be mapped to a file that physically exists on the server. Note that if the server file exists, the routing system ignores the request even if the request matches a route.

If you need to, you can force the routing system to handle all requests by setting the *Route ExistingFiles* property of the *RouteCollection* object to *true*, as shown here:

```
// In global.asax.cs
public static void RegisterRoutes(RouteCollection routes)
{
    routes.RouteExistingFiles = true;
    ...
}
```

Note that having all requests handled via routing can create some issues in an ASP.NET MVC application. For example, if you add the preceding code to the *global.asax.cs* file of a sample ASP.NET MVC application and run it, you'll immediately face an HTTP 404 error when accessing *default.aspx*.

## Preventing routing for defined URLs

The ASP.NET URL routing module doesn't limit you to maintaining a list of acceptable URL patterns; you can also keep certain URLs off the routing mechanism. You can prevent the routing system from handling certain URLs in two steps. First, you define a pattern for those URLs and save it to a route. Second, you link that route to a special route handler—the *StopRoutingHandler* class. All it does is throw a *NotSupported* exception when its *GetHttpHandler* method is invoked.

For example, the following code instructs the routing system to ignore any *.axd* requests:

```
// In global.asax.cs
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
    ...
}
```

All that *IgnoreRoute* does is associate a *StopRoutingHandler* route handler to the route built around the specified URL pattern.

Finally, a little explanation is required for the *{\*pathInfo}* placeholder in the URL. The token *pathInfo* simply represents a placeholder for any content following the *.axd* URL. The asterisk (\*), though, indicates that the last parameter should match the rest of the URL. In other words, anything that follows the *.axd* extension goes into the *pathInfo* parameter. Such parameters are referred to as *catch-all parameters*.

## Attribute routing

A popular NuGet package included in ASP.NET MVC 5 is *AttributeRouting*. (See <http://attributerouting.net>.) Attribute routing is all about defining routes directly on controller actions by using attributes. As previously demonstrated, classic routing is based on the conventions established in *global.asax*, at the startup of the application.

Any time a request comes in, the URL is matched against the template of registered routes. If a match is found, the appropriate controller and action method to serve the request are determined. If not, the request is denied and the result is usually a 404 message. Now, in large applications, or even in medium-sized applications with a strong REST flavor, the number of routes can be quite large and could easily be in the order of hundreds. You might quickly find that classic routing becomes a bit overwhelming to handle. For this reason, the *AttributeRouting* project was started and is now integrated in ASP.NET MVC 5 and even in Web API, as is discussed in Chapter 10, “An executive guide to Web API.”

```
[HttpGet("orders/{orderId}/show")]
public ActionResult GetOrderById(int orderId)
{
    ...
}
```

The code sets the method *GetOrderById* to be available over a HTTP GET call only if the URL template matches the specified pattern. The route parameter—the *orderId* token—must match one of the parameters defined in the method’s signature. There are a few more attributes available (for each HTTP verb), but the gist of attribute routes is all here. For more information (for example, configuration), you can refer to <http://attributerouting.net> because the integration for ASP.NET MVC is a direct emanation of the existing NuGet package.

## The controller class

---

In spite of the explicit reference to the Model-View-Controller pattern in the name, the ASP.NET MVC architecture is essentially centered on one pillar—the controller. The controller governs the processing of a request and orchestrates the back end of the system (for example, business layer, services, data access layer) to grab raw data for the response. Next, the controller wraps up raw data computed for the request into a valid response for the caller. When the response is a markup view, the controller relies on the view engine module to combine data and view templates and produce HTML.

## Aspects of a controller

Any request that passes the URL routing filter is mapped to a controller class and served by executing a given method on the class. Therefore, the controller class is the place where developers write the actual code required to serve a request. Let's briefly explore some characterizing aspects of controllers.

### Granularity of controllers

An ASP.NET MVC application usually comprises of a variety of controller classes. How many controllers should you have? The actual number is up to you and depends only on how you want to organize your application's actions. In fact, you could arrange an application around a single controller class that contains methods for any possible requests.

A common practice consists of having a controller class for each significant functionality your application implements. For example, you can have a *CustomerController* class that takes care of requests related to querying, deleting, updating, and inserting customers. Likewise, you can create a *ProductController* class for dealing with products, and so forth. Most of the time, these objects are directly related to items in the application's main menu.

In general, you can say that the granularity of the controller is a function of the granularity of the user interface. Plan to have a controller for each significant source of requests you have in the user interface.

### Stateless components

A new instance of the selected controller class is instantiated for each request. Any state you might add to the class is bound to the same lifetime of the request. The controller class then must be able to retrieve any data it needs to work from the HTTP request stream and the HTTP context.

### Further layering is up to you

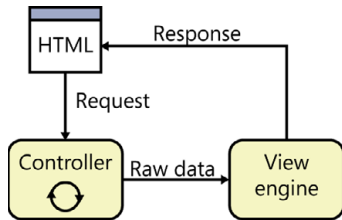
Often, ASP.NET MVC and controller classes are presented as a magic wand that you wave to write layered code that is cleaner and easier to read and maintain. The stateless nature of the controller class helps a lot in this regard, but it is not enough.

In ASP.NET MVC, the controller is isolated from both the user interface that triggered the request and the engine that produces the view for the browser. The controller sits in between the view and the back end of the system. Although this sort of isolation from the view is welcome and fixes a weak point of ASP.NET Web Forms, it alone doesn't ensure that your code will be respectful of the venerable principle of Separation of Concerns (SoC).

The system gets you a minimal level of separation from the view—everything else is up to you. Keep in mind that nothing, not even in ASP.NET MVC, prevents you from using direct ADO.NET calls and plain Transact-SQL (T-SQL) statements directly in the controller class. The controller class is not the back end of the system, and it is not the business layer. Instead, it should be considered as the MVC counterpart of the code-behind class of Web Forms. As such, it definitely belongs to the presentation layer, not the business layer.

## Highly testable

The inherent statelessness of the controller and its neat separation from the view make the controller class potentially easy to test. However, the real testability of the controller class should be measured against its effective layering. Let's have a look at Figure 1-3.



**FIGURE 1-3** Controllers and views in ASP.NET MVC.

Although you can feed the controller class any fixed input you like and its output can be asserted without major issues, nothing can be said about the internal structure of action methods. The more the implementation of these methods is tightly bound to external resources (for example, databases, services, components), the less likely it is that testing a controller will be quick and easy.

## Writing controller classes

The writing of a controller class can be summarized in two simple steps: creating a class that inherits (either directly or indirectly) from *Controller*, and adding a bunch of public methods. However, a couple of important details must be clarified: how the system gets to know the controller class to instantiate, and how it figures out the method to invoke.

### From routing to controllers

Regardless of how you define your URL patterns, any request must always be resolved in terms of a controller name and an action name. This is one of the pillars of ASP.NET MVC. The controller name is automatically read from the URL if the URL includes a *{controller}* placeholder. The same happens for action names if the URL contains an *{action}* placeholder.

Having completely custom URLs devoid of such placeholders is still acceptable, though. In this case, however, it is your responsibility to indicate the controller and action through default values, as shown here:

```
routes.MapRoute(
    "SampleRoute",
    "about",
    new { controller = "Home", action = "About" }
);
```

If controller and action names can't be resolved in a static way, you might want to write a custom route handler, explore the details of the request, and figure out controller and action names. Then, you just store them in the *RouteData* collection, as shown here:

```
public class AboutRouteHandler : IRouteHandler
{
    public IHttpHandler GetHttpHandler(RequestContext requestContext)
    {
        if (requestContext.HttpContext.Request.Url.AbsolutePath == "/about")
        {
            requestContext.RouteData.Values["controller"] = "home";
            requestContext.RouteData.Values["action"] = "about";
        }
        return new MvcHandler(requestContext);
    }
}
```

For a route that requires a custom handler, the registration process is a bit different from what you saw earlier. Here's the code you need to have in *RegisterRoutes*:

```
public static void RegisterRoutes(RouteCollection routes)
{
    var aboutRoute = new Route("about", new AboutRouteHandler());
    routes.Add("SampleAboutRoute", aboutRoute);
    ...
}
```

Be sure to note that the controller name you obtain from the routing module doesn't match exactly the actual name of class that will be invoked. By default, the controller class is named after the controller name with a *Controller* suffix added. In the previous example, if *home* is the controller name, the class name is assumed to be *HomeController*. Note that conventions apply not just to the class name but also to the namespace. In particular, the class is expected to be scoped in the *Controllers* namespace under the default project namespace.



**Note** When you add a route based on a custom route handler that sets controller and action names programmatically, you might run into trouble with the links generated by the *Html.ActionLink* helper. You commonly use this helper to create route-based links for menus and other visual elements of the user interface. If you add a route with a custom handler, you might be surprised to see that the links you get from the helper are unexpectedly based on this route. To solve the issue, either you change *ActionLink* with *RouteLink* and expressly indicate which route you want the URL to be created after, or you specify in the custom route that controller and action are optional parameters.



## From routing to actions

When the ASP.NET MVC run-time environment has a valid instance of the selected controller class, it yields to the action invoker component for the actual execution of the request. The action invoker gets the action name and attempts to match it to a public method on the controller class.

The action parameter indicates the name of the action to perform. Most of the time, the controller class just has a method with the same name. If this is the case, the invoker will execute it. Note, though, that you can associate an action name attribute to any public method, thus decoupling the method name from the action name. Here's an example:

```
public class HomeController : Controller
{
    // Implicit action name: Index
    public ActionResult Index()
    {
        ...
    }

    [NonAction]
    public ActionResult About()
    {
        ...
    }

    [ActionName("About")]
    public ActionResult LoveGermanShepherds()
    {
        ...
    }
}
```

The method *Index* is not decorated with attributes, so it is implicitly bound to an action with the same name. The third public method has a very fancy name, but it is explicitly bound to the action *About* via the *ActionName* attribute. Finally, note that to prevent a public controller method from being implicitly bound to an action name, you use the *NonAction* attribute. Therefore, given the previous code snippet when the user requests the about action, the method *LoveGermanShepherds* runs regardless of the HTTP verb used to place the request.

## Actions and HTTP verbs

ASP.NET MVC is flexible enough to let you bind a method to an action for a specific HTTP verb. To associate a controller method with an HTTP verb, you either use the parametric *AcceptVerbs* attribute or direct attributes such as *HttpGet*, *HttpPost*, and *HttpPut*. Using the *AcceptVerbs* attribute, you can specify which HTTP verb is required to execute a given method. Let's consider the following example:

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(Customer customer)
{
    ...
}
```

Given that code, it turns out that the *Edit* method can't be invoked by using a *GET*. Note also that you are not allowed to have multiple *AcceptVerbs* attributes on a single method. Your code won't compile if you add multiple *AcceptVerbs* attributes (or analogous direct HTTP verb attributes) to an action method.

The *AcceptVerbs* attribute takes any value from the *HttpVerbs* enum type:

```
public enum HttpVerbs
{
    Get = 1,
    Post = 2,
    Put = 4,
    Delete = 8,
    Head = 0x10
}
```

The *HttpVerbs* enum is decorated with the *Flags* attribute, so you can combine together multiple values from the enumeration by using the bitwise OR (*|*) operator and still obtain another *HttpVerbs* value.

```
[AcceptVerbs(HttpVerbs.Post|HttpVerbs.Put)]
public ActionResult Edit(Customer customer)
{
    ...
}
```

You perform an HTTP *GET* command when you follow a link or type the URL into the address bar. You perform an HTTP *POST* when you submit the content of an HTML form. You can perform any other HTTP command only via AJAX, or perhaps from a Windows client that sends requests to the ASP.NET MVC application.

The ability to assign a specific verb to a given action method naturally leads to duplicate method names. Two methods with the same name are acceptable in a controller class as long as they accept distinct HTTP verbs. Otherwise, an exception will be thrown because ASP.NET MVC doesn't know how to resolve the ambiguity.



**Note** You can also use multiple individual attributes, one for each HTTP verb. Examples are *HttpGet* and *HttpPost*.

## Action methods

Let's have a look at a sample controller class with a couple of simple but functional action methods:

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        // Process input data
        ...
    }
}
```

```

        // Perform expected task
        ...

        // Generate the result of the action
        return View();
    }

    public ActionResult About()
    {
        // Process input data
        ...

        // Perform expected task
        ...

        // Generate the result of the action
        return View();
    }
}

```

An action method grabs available input data by using any standard HTTP channels. Next, it arranges for some action and possibly involves the middle tier of the application. We can summarize the template of an action method as follows:

- **Process input data** An action method gets input arguments from a couple of sources: route values and collections exposed by the *Request* object. ASP.NET MVC doesn't mandate a particular signature for action methods. However, for testability reasons, it's highly recommended that any input parameter is received through the signature. If you can, avoid methods that retrieve input data programmatically from *Request* or other sources. As you'll see later in this chapter, and even more thoroughly in Chapter 3, "The model-binding architecture," an entire subsystem exists—the model binding layer—to map HTTP parameters to action method arguments.
- **Perform the task** The action method does its job based on input arguments and attempts to obtain expected results. In doing so, the method likely needs to interact with the middle tier. As is discussed further in Chapter 7, "Design considerations for ASP.NET MVC controllers," it is recommended that any interaction takes place through ad hoc dedicated services. At the end of the task, any (computed or referenced) values that should be incorporated in the response are packaged as appropriate. If the method returns JavaScript Object Notation (JSON), data is composed into a JSON-serializable object. If the method returns HTML, data is packaged into a container object and sent to the view engine. The container object is often referred to as the view-model and can be a plain dictionary of name/value pairs or a view-specific, strongly typed class.
- **Generate the results** In ASP.NET MVC, a controller's method is not responsible for producing the response itself. However, it is responsible for triggering the process that will use a distinct object (often, a view object) to render content to the output stream. The method identifies the type of response (file, plain data, HTML, JavaScript, or JSON) and sets up an *ActionResult* object, as appropriate.

A controller's method is expected to return an *ActionResult* object or any object that inherits from the *ActionResult* class. Often, though, a controller's method doesn't directly instantiate an *ActionResult* object. Instead, it uses an action helper—that is, an object that internally instantiates and returns an *ActionResult* object. The method *View* in the preceding example provides an excellent illustration of an action helper. Another great example of such a helper method is *Json*, which is used when the method needs to return a JSON string. I'll return to this point in just a moment.

## Processing input data

Controller action methods can access any input data posted by using the HTTP request. Input data can be retrieved from a variety of sources, including form data, query strings, cookies, route values, and posted files.

The signature of a controller action method is free. If you define parameter-less methods, you make yourself responsible for programmatically retrieving any input data your code requires. If you add parameters to the method's signature, ASP.NET MVC will offer automatic parameter resolution. In particular, ASP.NET MVC will attempt to match the names of formal parameters to named members in a request-scoped dictionary that joins together values from the query string, route, posting form, and more.

In this chapter, I discuss how to manually retrieve input data from within a controller action method. Chapter 3 discusses automatic parameter resolution—the most common choice in ASP.NET MVC applications.

### Getting input data from the *Request* object

When writing the body of an action method, you can certainly access any input data that comes through the familiar *Request* object and its child collections, such as *Form*, *Cookies*, *ServerVariables*, and *QueryString*. As you'll see later in the book, when it comes to input parameters of a controller method, ASP.NET MVC offers quite compelling facilities (for example, model binders) that you might want to use to keep your code cleaner, more compact, and easier to test. Having said that, though, nothing at all prevents you from writing old-style *Request*-based code, as shown here:

```
public ActionResult Echo()
{
    // Capture data in a manual way
    var data = Request.Params["today"] ?? String.Empty;
    ...
}
```

In ASP.NET, the *Request.Params* dictionary results from the combination of four distinct dictionaries: *QueryString*, *Form*, *Cookies*, and *ServerVariables*. You can also use the *Item* indexer property of the *Request* object, which provides the same capabilities and searches dictionaries for a matching entry in the following order: *QueryString*, *Form*, *Cookies*, and *ServerVariables*. The following code is fully equivalent to that just shown:

```
public ActionResult Echo()
{
    // Capture data in a manual way
    var data = Request["today"] ?? String.Empty;
    ...
}
```

Observe that the search for a matching entry is case insensitive.

## Getting input data from the route

In ASP.NET MVC, you often provide input parameters through the URL. These values are captured by the routing module and made available to the application. Route values are not exposed to applications through the *Request* object. You have to use a slightly different approach to retrieve them programmatically, as demonstrated here:

```
public ActionResult Echo()
{
    // Capture data in a manual way
    var data = RouteData.Values["data"] ?? String.Empty;
    ...
}
```

Route data is exposed through the *RouteData* property of the *Controller* class. Also, in this case, the search for a matching entry is conducted in a case-insensitive manner.

The *RouteData.Values* dictionary is a *String/Object* dictionary. The dictionary contains only strings most of the time. However, if you populate this dictionary programmatically (for example, via a custom route handler), it can contain other types of values. In this case, you're responsible for any necessary type cast.

## Getting input data from multiple sources

Of course, you can mix *RouteData* and *Request* calls in the same controller method. As an example, let's consider the following route:

```
routes.MapRoute(
    "EchoRoute",
    "echo/{data}",
    new { controller = "Home", action = "Echo", data = UrlParameter.Optional }
);
```

The following is a valid URL: <http://yourserver/echo/Sunday>. The code shown next will easily grab the value of the data parameter (Sunday). Here's a possible implementation of the *Echo* method in the *HomeController* class:

```
public ActionResult Echo()
{
    // Capture data in a manual way
    var data = RouteData.Values["data"];
    ...
}
```

What if you call the following URL, instead?

`http://yourserver/echo?today=3/27/2011`

The URL still matches the route pattern, but it doesn't provide a value for the data parameter. Still, the URL adds some input value in the query string for the controller action to consider. Here's the modified version of the *Echo* method that supports both scenarios:

```
public ActionResult Echo()
{
    // Capture data in a manual way
    var data = RouteData.Values["data"] ??
        (Request.Params["today"] ?? String.Empty);
    ...
}
```

The question is, "Should I plan to have a distinct branch of code for each possible input channel, such as form data, query string, routes, cookies, and so forth?" Enter the *ValueProvider* dictionary.

## The ValueProvider dictionary

In the *Controller* class, the *ValueProvider* property just provides a single container for input data collected from a variety of sources. By default, the ValueProvider dictionary is fed by input values from the following sources (in the specified order):

- 1. Child action values** Input values are provided by child action method calls. A child action is a call to a controller method that originates from the view. A child action call takes place when the view calls back the controller to get additional data or to demand the execution of some special task that might affect the output being rendered. I discuss child actions in Chapter 2, "ASP.NET MVC views."
- 2. Form data** Input values are provided by the content of the input fields in a posting HTML form. The content is the same as you would get through *Request.Form*.
- 3. Route data** Input values are provided by the content associated with parameters defined in the currently selected route.
- 4. Query string** Input values are provided by the content of parameters specified in the query string of the current URL.
- 5. Posted files** Input values are represented by the file or files posted via HTTP in the context of the current request.

The ValueProvider dictionary offers a custom programming interface centered on the *GetValue* method. Here's an example:

```
var result = ValueProvider.GetValue("data");
```

Be aware that *GetValue* doesn't return a *String* or an *Object* type. Instead, it returns an instance of the *ValueProviderResult* type. The type has two properties to actually read the real parameter value:

*RawValue* and *AttemptedValue*. The former is of type *Object* and contains the raw value as provided by the source. The *AttemptedValue* property, on the other hand, is a string and represents the result of an attempted type cast to *String*. Here's how to implement the *Echo* method by using *ValueProvider*:

```
public ActionResult Echo()
{
    var data = ValueProvider.GetValue("data").AttemptedValue ??
        (ValueProvider.GetValue("today").AttemptedValue ?? String.Empty);
    ...
}
```

*ValueProvider* is a bit more demanding than *Request* and *RouteData* when it comes to parameter names. If you mistype the case of a parameter, you'll get a null object back from *GetValue*. This leads to an exception if you then just read the value without checking the result object for nullness.

Finally, note that by default you won't get access to cookies through the *ValueProvider* dictionary. However, the list of value providers can be extended programmatically by defining a class that implements the *IValueProvider* interface.



**Note** The value-provider mechanism can be useful to retrieve some request data which is packed into a comfortable collection of values. Default value providers save you from the burden of looking into the *QueryString* or *Form* collection. What if you need to read data from a cookie or a request header? You can go the usual way and read the *Headers* or *Cookies* collection of the *Request* object and write the code that extracts individual values. However, if your application is extensively based on request headers or cookies, you might want to consider writing a custom value provider. It is not hard to find working examples of both from the community. You can find a good example of a value provider that exposes request headers at <http://blog.donnfelker.com/2011/02/16/asp-net-mvc-building-web-apis-with-headervalueprovider>.

## Producing action results

An action method can produce a variety of results. For example, an action method can just act as a web service and return a plain string or a JSON string in response to a request. Likewise, an action method can determine that there's no content to return or that a redirect to another URL is required. In these two cases, the browser will just get an HTTP response with no significant body of content. This is to say that one thing is producing the raw result of the action (for example, collecting values from the middle tier); it is quite another case to process that raw result to generate the actual HTTP response for the browser. The *ActionResult* class just represents the ASP.NET MVC infrastructure for implementing this programming aspect.

## Inside the *ActionResult* class

An action method typically returns an object of type *ActionResult*. The type *ActionResult* is not a data container, though. More precisely, it is an abstract class that offers a common programming interface to execute some further operations on behalf of the action method. Here's the definition of the *ActionResult* class:

```
public abstract class ActionResult
{
    protected ActionResult()
    {
    }

    public abstract void ExecuteResult(ControllerContext context);
}
```

By overriding the *ExecuteResult* method, a derived class gains access to any data produced by the execution of the action method and triggers some subsequent action. Generally, this subsequent action is related to the generation of some response for the browser.

## Predefined action result types

Because *ActionResult* is an abstract type, every action method is actually required to return an instance of a more specific type. Table 1-1 lists all predefined action result types.

**TABLE 1-1** Predefined *ActionResult* types in ASP.NET MVC

Type	Description
<i>ContentResult</i>	Sends raw content (not necessarily HTML) to the browser. The <i>ExecuteResult</i> method of this class serializes any content it receives.
<i>EmptyResult</i>	Sends no content to the browser. The <i>ExecuteResult</i> method of this class does nothing.
<i>FileContentResult</i>	Sends the content of a file to the browser. The content of the file is expressed as a byte array. The <i>ExecuteResult</i> method simply writes the array of bytes to the output stream.
<i>FilePathResult</i>	Sends the content of a file to the browser. The file is identified via its path and content type. The <i>ExecuteResult</i> method calls the <i>TransmitFile</i> method on <i>HttpResponse</i> .
<i>FileStreamResult</i>	Sends the content of a file to the browser. The content of the file is represented through a <i>Stream</i> object. The <i>ExecuteResult</i> method copies from the provided file stream to the output stream.
<i>HttpNotFoundResult</i>	Sends an HTTP 404 response code to the browser. The HTTP status code identifies a request that failed because the requested resource was not found.
<i>HttpUnauthorizedResult</i>	Sends an HTTP 401 response code to the browser. The HTTP status code identifies an unauthorized request.
<i>JavaScriptResult</i>	Sends JavaScript text to the browser. The <i>ExecuteResult</i> method of this class writes out the script and sets the content type accordingly.



Type	Description
<i>JsonResult</i>	Sends a JSON string to the browser. The <i>ExecuteResult</i> method of this class sets the content type to the application or JSON and invokes the <i>JavaScriptSerializer</i> class to serialize any provided managed object to JSON.
<i>PartialViewResult</i>	Sends HTML content to the browser that represents a fragment of the whole page view. A partial view in ASP.NET MVC is a concept very close to a user control in Web Forms.
<i>RedirectResult</i>	Sends an HTTP 302 response code to the browser to redirect the browser to the specified URL. The <i>ExecuteResult</i> method of this class just invokes <i>Response.Redirect</i> .
<i>RedirectToRouteResult</i>	Like <i>RedirectResult</i> , it sends an HTTP 302 code to the browser and the new URL to which to navigate. The difference is in the logic and input data employed to determine the target URL. In this case, the URL is built based on action/controller pairs or route names.
<i>ViewResult</i>	Sends HTML content to the browser that represents a full page view.

Note that *FileContentResult*, *FilePathResult*, and *FileStreamResult* derive from the same base class: *FileResult*. You use any of these action result objects if you want to reply to a request with the download of some file content or even some plain binary content expressed as a byte array. *PartialViewResult* and *ViewResult* inherit from *ViewResultBase* and return HTML content. Finally, *HttpUnauthorizedResult* and *HttpNotFoundResult* represent two common responses for unauthorized access and missing resources. Both derive from a further extensible class *HttpStatusCodeResult*.

## The mechanics of executing action results

To better comprehend the mechanics of action result classes, let's dissect one of the predefined classes. I've chosen the *JavaScriptResult* class, which provides some meaningful behavior without being too complex. The *JavaScriptResult* class represents the action of returning some script to the browser. Here's a possible action method that serves up JavaScript code:

```
public JavaScriptResult GetScript()
{
    var script = "alert('Hello')";
    return JavaScript(script);
}
```

In the example, *JavaScript* is a helper method in the *Controller* class that acts as a factory for the *JavaScriptResult* object. The implementation looks like this:

```
protected JavaScriptResult JavaScript(string script)
{
    return new JavaScriptResult() { Script = script };
}
```

The *JavaScriptResult* class supplies a public property—the *Script* property—that contains the script code to write to the output stream. Here's its implementation:

```
public class JavaScriptResult : ActionResult
{
    public String Script { get; set; }

    public override void ExecuteResult(ControllerContext context)
    {
        if (context == null)
            throw new ArgumentNullException("context");

        // Prepare the response
        HttpResponseBase response = context.HttpContext.Response;
        response.ContentType = "application/x-javascript";
        if (Script != null)
            response.Write(Script);
    }
}
```

As you can see, the ultimate purpose of the *ActionResult* class is to prepare the *HttpResponse* object to return to the browser. This entails setting content type, expiration policies, and headers, as well as content.

## Returning HTML markup

Most of the time, requests are served by sending back HTML markup. Composing the HTML for the browser is the core of a web framework. In ASP.NET Web Forms, the task of composing HTML is done through the page. Developers create ASPX pages as a mix of a view template and a code-behind class. Both the action to grab results and the production of the actual response are blurred in a single run-time environment. In ASP.NET MVC, the production of the results is the responsibility of the action method; managing for the response to be composed and served is the responsibility of the framework. Finally, composing the HTML markup is the responsibility of yet another system component—the view engine.

I discuss view engines in Chapter 2, but for now it suffices to say that a view engine knows how to retrieve a view template for a given action and how to process that to a plain HTML stream that mixes template information and raw data. The view engine dictates the syntax of the view template (ASPX, Razor, and Spark to name a few); the developer dictates the format of the raw data to be merged into the view. Let's consider a sample action method returning HTML:

```
public ActionResult Index()
{
    return View(); // same as View("index");
}
```

The *View* method is a helper method responsible for creating a *ViewResult* object. The *ViewResult* object needs to know about the view template, an optional master view, and the raw data to be

incorporated into the final HTML. The fact that in the code snippet *View* has no parameters doesn't mean no data is actually passed on. Here's one of the signatures of the method:

```
protected ViewResult View(String viewName, String masterName, Object model)
```

By convention, the view template is a file named after the action name (*Index* in this case) and located in a specific folder. The exact location depends on the implementation of the currently selected view engine. By default, view templates are expected to be located in the Views folder in a directory that matches the name of the controller—for example, Views/Home. Be aware that you must maintain this directory structure when you deploy the site.

The extension of the view template file also depends on the implementation of the view engine. For the two predefined view engines you get with ASP.NET MVC, the extensions are *.aspx* if you opt for the ASPX view engine and *.cshtml* (or *.vbhtml*) if you opt for the Razor view engine. (I provide more details about this in Chapter 2.)

## Returning JSON content

ASP.NET MVC lends itself very well to implementing simple web services to be called back from jQuery snippets in an Ajax context. All you need to do is set one or more action methods to return JSON strings instead of HTML. Here's an example:

```
public JsonResult GetCustomers()
{
    // Grab some data to return
    var customers = _customerRepository.GetAll();

    // Serialize to JSON and return
    return Json(customers);
}
```

The *Json* helper method gets a plain .NET object and serializes it to a string by using the built-in *JavaScriptSerializer* class.



**Note** What if your controller action method doesn't return *ActionResult*? First and foremost, no exceptions are raised. Quite simply, ASP.NET MVC encapsulates any return value from the action method (numbers, strings, or custom objects) into a *ContentResult* object. The execution of a *ContentResult* object causes the plain serialization of the value to the browser. For example, an action that returns an integer or a string will get you a browser page that displays data as-is. On the other hand, returning a custom object displays any string resulting from the implementation of the object's *ToString* method. If the method returns an HTML string, any markup will not be automatically encoded and the browser will likely not properly parse it. Finally, a void return value is actually mapped to an *EmptyResult* object whose execution causes a no-op.

## Asynchronous operations within a controller

The primary purpose of a controller is to serve the needs of the user interface. Any server-side functions you need to implement should be mapped to a controller method and triggered from the user interface. After performing its own task, a controller's method selects the next view, packs some data, and instructs it to render.

This is the essence of the controller's behavior. However, other characteristics are often required in a controller, especially when controllers are employed in large and complex applications with particular needs, such as long-running requests. In earlier versions of ASP.NET MVC, you had to follow a specific pattern to give controller methods an asynchronous behavior. Starting with the .NET Framework 4.5, you can take advantage of the new *async/await* language facilities and the underlying .NET machinery. Here's the way in which you write a controller class with one or more asynchronous methods:

```
public class HomeController : AsyncController
{
    public async Task<ActionResult> Rss()
    {
        // Run the potentially lengthy operation
        var client = new HttpClient();
        var rss = await client.GetStringAsync(someRssUrl);

        // Parse RSS and build the view model
        var model = new HomeIndexModel();
        model.News = ParseRssInternal(rss);
        return model;
    }
}
```

The code looks as if it were written to be synchronous—you don't care about callbacks. In the end, though, it is highly readable and runs asynchronously thanks to the syntactic sugar added by the C# compiler when you use *async/await* keywords.

## Summary

---

Controllers are the heart of an ASP.NET MVC application. Controllers mediate between the user requests and the capabilities of the server system. Controllers are linked to user-interface actions and are in touch with the middle tier. Controllers order the rendering of the page but don't run any rendering tasks themselves. This is a key difference from ASP.NET Web Forms. In a controller, the processing of the request is neatly separated from the display. In Web Forms, on the other hand, the page-processing phase incorporates both the execution of some tasks and the rendering of the response.

Although based on a different syntax, controller methods are not much different from the post-back event handlers you have in ASP.NET Web Forms. In this regard, a controller class plays the same role of a code-behind class in Web Forms. The controller as well as a Web Forms code-behind class belongs to the presentation layer. For this reason, you should pay a lot of attention to how you code the behavior of the various action methods. Keep in mind that in ASP.NET MVC, any layering in the building of the solution is also up to you.

In this chapter, I skipped over all the details about how you add behavior to a controller method. I focused on an overview of what comes before and what comes after. In Chapter 2, I delve deeper into what comes after; therefore, the focus will be on views, view engines, and the generation of the markup. Then, in Chapter 3, I discuss model binding and what happens before the behavior of an action method comes into play. And in Chapter 7, I come back to this topic with some design considerations on how to structure methods in a controller class.

This has been just the first pass on controllers. A lot more must be said and learned.

## Microsoft Press Ebooks—Your bookshelf on your devices!



When you buy an ebook through [oreilly.com](http://oreilly.com) you get lifetime access to the book, and whenever possible we provide it to you in five, DRM-free file formats—PDF, .epub, Kindle-compatible .mobi, Android .apk, and DAISY—that you can use on the devices of your choice. Our ebook files are fully searchable, and you can cut-and-paste and print them. We also alert you when we’ve updated the files with corrections and additions.

Learn more at [ebooks.oreilly.com](http://ebooks.oreilly.com)

You can also purchase O’Reilly ebooks through the iBookstore, the [Android Marketplace](http://AndroidMarketplace), and [Amazon.com](http://Amazon.com).

# O'REILLY®

Spreading the knowledge of innovators

[oreilly.com](http://oreilly.com)