Extracted from:

Test-Drive ASP.NET MVC

The Pragmatic Programmers

Covers ASP.NET MVC 2.0

Test-Drive ASP.NET MVC

Jonathan McCracken

Edited by Susannah Davidson Pfalzer

The best years of your life are the ones in which you decide your problems are your own. You do not blame them on your mother, the ecology, or the president. You realize that you control your own destiny.

► Albert Ellis

Chapter 5

Managing State and Files with Controllers

Controllers are the heavy lifters of the MVC framework. We use them to coordinate all activity between the user and the model. Up to this point we have implemented controllers, but we haven't covered all of their capabilities. There are a few tricks we have yet to learn that will help us develop web applications more efficiently. In this chapter, we will learn about the additional features of controllers: action filters, HttpSessionState, and file manipulation.

GetOrganized needs to restrict access to certain pages. Action filters will help us do that. We'll also need to keep track of what our users are doing while they are logged in by using HttpSessionState, and we'll see how to attach files to our Thoughts. We'll use MVCContrib,[1] an open source project that offers many enhancements to the MVC framework, to help improve the readability of our controller tests.

We have already learned that controllers direct models to views, but now we're going to look at how they can help us manage state and other external resources like files. Let's begin by looking at how controllers fit into the overall picture of the MVC framework.

5.1 Enabling Filters and Results with Controllers

Understanding how action filters and results work will help you see how controllers are created and how control is passed to them within MVC.

1. http://www.codeplex.com/mvccontrib

The controller is the entry point of our program and acts as the coordinator between the model and one or more views. From the request input from the browser, the controller takes the appropriate action. Each action interacts with a model and determines which view or other controller to send the response to. Since actions are where we spend a lot of time testing and coding, it is no surprise that MVC has built extension points into the framework in the form of action results and filters.

In this section, we'll learn how filters can help us roll up common functionality, such as security, into easy-to-apply labels to our actions. We'll also work through how action filters can make returning non-HTML resources such as files simpler than ever. Finally, we'll touch on how to leverage the HttpContext and store information from request to request.

Extending Actions with Action Filters

Action filters are C# attributes that you can apply to the controller or its actions. These filters perform an operation before the action is executed. Action filters are an example of *declarative programming*, aptly named after declaring what we want to accomplish up front.

We saw our first action filter, [HttpPost], in Section 3.3, *Creating a To-Do*, on page 58. We'll now see in depth how they work and when we use them. For example, an alternative to [HttpPost] is as follows:

Download controllers/ActionFilters.cs

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Create(Todo item)
{
   // create Todo here
}
```

In this case, the action filter is the [AcceptVerbs] attribute, and it's applied to the Create(Todo todo) action. Another example of an action filter is the [Authorize] attribute. You add this filter to controllers to restrict access so only users who are logged in can use it. An example of this is as follows:

Download controllers/ActionFilters.cs

```
[Authorize]
public ActionResult SecureMe()
{
  // users cannot get here unless they are logged in
}
```

Joe Asks...

When Do I Use Output Caching?

Caching at the output level is normally reserved for things like the web application's home page. Home pages generally don't change based on individual user information and so are safe to output cache. Caching keeps a copy of the output that was generated from the first request. That makes the second request render a lot faster. However, you don't want to use caching for every action. For example, if you have an action that uses specific information from a user's profile, then caching would show other users' information.

Output caching is not the only form of caching. Caching data is another way to speed up page load times. Once we start working with NHibernate in Chapter 8, *Persisting Your Models*, on page 174, we'll automatically get a certain level of data caching by default. If you want to really get serious about data caching, you should read more about the second-level caching available to NHibernate with the open source server memcached.*

*. http://www.cnblogs.com/RicCC/archive/2007/10/13/NHibernate-Memcached.html

Action filters contain generic behavior that you can reuse on multiple actions. For example, another useful action filter to be aware of is [HandleError], which redirects the user in case of exceptions. We'll talk more about that in Section 11.2, *Using an Action Filter to Handle Errors*, on page 240.

The action filter [OutputCache] (see the *Joe Asks...* on this page) caches requests and improves performance of frequently requested actions. In Chapter 11, *Security, Error Handling, and Logging*, on page 231, we'll create our own action filters by inheriting from the class FilterAction. In this chapter, we'll learn to apply the [Authorize] attribute in Section 5.2, *Logging In*, on page 102.

We can also author our own action filters. This is something that we'll do in Section 9.4, *Creating a Custom Action Filter*, on page 202. Writing custom action filters is another reason why MVC is such an extensible framework.

Action filters are just one piece of the puzzle. To round out our understanding of controllers, you need to know about ActionResults.

Directing to Different Content Types with ActionResults

As we learned in Section 3.2, *Our First Test*, on page 49, action results are the return type of controllers. So far we've learned how to render a view and redirect to another action. This covers two subtypes of ActionResult, the abstract class that all controllers use as their return type.

The ViewResult is a subtype we've used to direct to a view that displays information, such as the Index() action on the TodoController. Here is an example of an action returning a ViewResult:

Download controllers/ActionResults.cs

```
public ActionResult Display()
{
  return View(ModelToDisplay);
}
```

What's interesting here is we've introduced a shortcut for how to set a model that is different than covered in the previous chapters. Instead of ViewData.Model = ModelToDisplay, we can accomplish the same thing with View(ModelToDisplay).

The second subtype of ActionResult is the RedirectToRouteResult. This result redirects control to another action or a different controller altogether. Here is a sample of using RedirectToAction():

Download controllers/ActionResults.cs

```
public ActionResult Redirector()
{
  return RedirectToAction("DifferentAction","DifferentController");
}
```

Notice that the first argument of RedirectToAction(string action, string controller) is the action we want to send the control to, and the second is the controller. We can omit the second argument if we want to stay within the same controller, like going from Create() to Index() on the successful creation of a model. In both cases, we never create an action result directly using return new RedirectToRouteResult{ //...}. Instead, we use methods that are available from the base Controller, which all controllers inherit from.

MVC offers many other types of action results. The most useful action results are as follows:

- JsonResult: Returns models in JavaScript Object Notation (JSON)
- ContentResult: Returns plain text
- FilePathResult: Returns a file from a path on the server
- FileStreamResult: Returns a file from a stream

Joe Asks...

When Would I Use JSON?

JSON is a helpful format to allow JavaScript libraries such as jQuery to manipulate objects, as opposed to using XML or HTML. Using JSON allows us to focus on becoming proficient in JavaScript, instead of having to master XPath. An object like Thought renders in JSON as follows:

```
var thought = {
        "Id": 0,
        "Topic": {
                "Id": 1,
                "Color": {
                        "R": 255,
                        "G": 0,
                        "B": 0,
                        "A": 255,
                        "IsKnownColor": true,
                        "IsEmpty": false,
                        "IsNamedColor": true,
                        "IsSystemColor": false,
                        "Name": "Red"
                },
                "Name": "Work"
        },
        "Name": "Learn C# 3.5"
}
```

Accessing the object within JavaScript with JSON becomes much easier with thought.Topic.Color, instead of using XPath to access the same field like this:

```
document.evaluate("thought/Topic/Color", document, null,
      XPathResult.ANY_TYPE, null)
```

JsonResult is useful for Ajax requests and often works with the ContentResult (these action results will be covered in Chapter 7, *Composing Views with Ajax and Partials*, on page 153). Most Internet applications need to allow users to download files, and MVC does this with FilePathResult and FileStreamResult. FilePathResult uses files located on the file system of the web server; we will use them in Section 5.5, *Manipulating Files*, on page 121. FileStreamResult takes any output stream, such as streaming a file out of the database.

Some action results are useful in specific situations but not used as frequently:

- JavascriptResult: Returns a JavaScript file
- FileContentResult: Returns a file as binary data
- EmptyResult: Returns nothing

JavascriptResult can aggregate all our application's JavaScript into one simple file reference. This is handy if there are a lot of jQuery plug-ins, each having a separate JavaScript file. Instead of having to add a large head section to your HTML file, imagine just using one with the help of this action result.

FileContentResult is a specialized kind of file return where you must return a Byte array, perhaps for graphics manipulation. Generally, we're better off using the FileStreamResult because it is faster and friendlier to our web server's memory usage.

EmptyContent is for rare situations where you would like to return something directly to the HttpResponse, such as when serving some static HTML fragment. Doing this short-circuits the normal pattern of MVC and is not recommended unless there is absolutely no alternative.

Although these are the default action results that come with MVC, the reason that the framework designers created ActionResult as an abstract class was for extensibility by developers like us. In Chapter 10, *Building RESTful Web Services*, on page 212, we'll see how to use MVCContrib's XmlResult to return XML from an action.

IControllerFactory: Where Controllers Are Born

It's helpful to understand how web requests are processed, because it puts the IControllerFactory in the context of the request pipeline. When a request comes in from the browser, it is handed off to the web server and passed through the UrlRoutingModule.

This module creates an MVCRouteHandler where the request gets executed in the method ProcessRequest(). This method gets a reference to the IControllerFactory interface responsible for creating an instance of our controller and calling the controller's Execute() method. Execute() is where our actions are called.

The other main function of the MVCRouteHandler is that it hooks up the HttpSessionState so we can use it between requests to store information

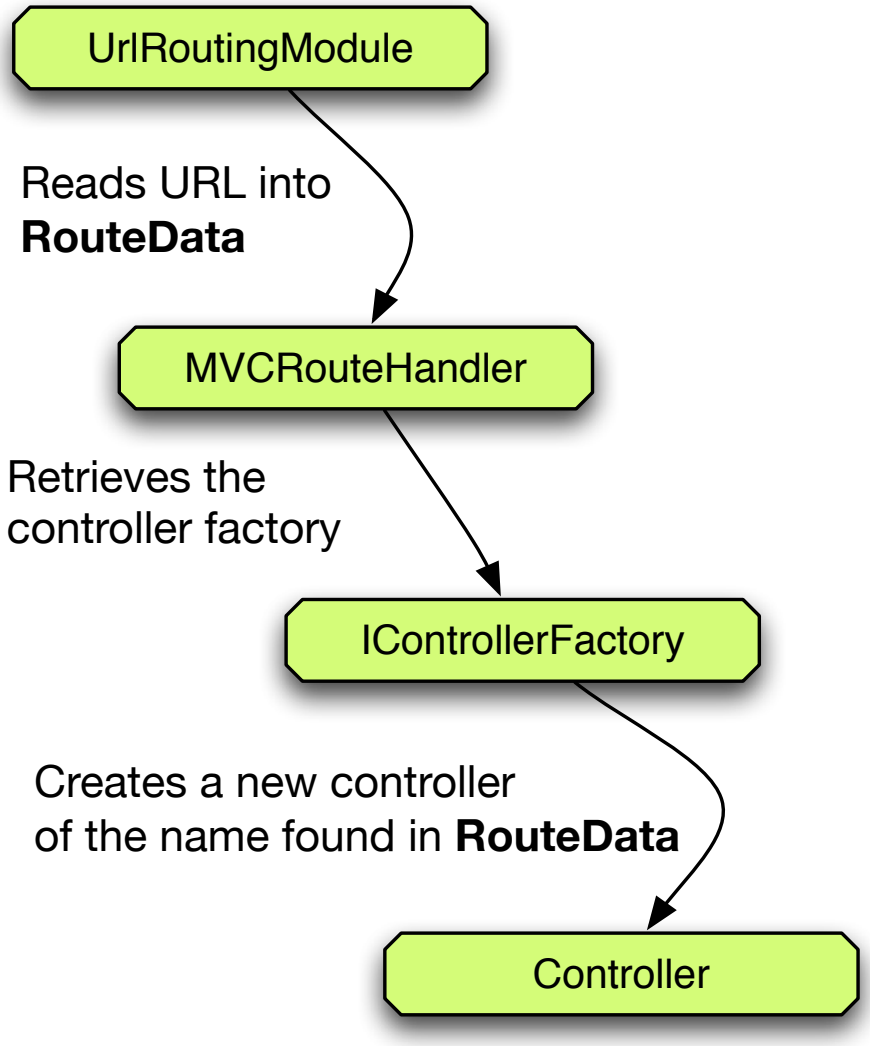

Figure 5.1: Controllers are created by the IControllerFactory based on a name inside the URL.

unique to each user. Controllers populate and manipulate HttpSessionState. These types of interactions between controllers and HttpSessionState will be covered in Section 5.4, *Storing Information in Memory*, on page 111 (see Figure 5.1).

The birthplace of a controller is the DefaultControllerFactory, which implements the IControllerFactory interface. The factory knows what type of controller to create using the RouteData class, which was set when the UrlRoutingModule ran earlier. For example, http://localhost/Todo parses Todo to create a new instance of type TodoController.

This factory interface allows us to extend how controllers are built; this extensibility helps us implement dependency injection. We'll cover this topic in Section 9.2, *Using Inversion of Control with the IControllerFactory*, on page 194 when we use MVCContrib to inject the database layer into controllers.

The term Inversion of Control (IoC) is mentioned a lot these days; it has become synonymous with dependency injection. We pass the behavior to the constructor, effectively "injecting" it into the object, as in new SomeController(IDatabaseConnector connector). The actual logic for the database is injected into SomeController. This helps us test each class separately.

Let's look at how action filters help add a user login to our application.

5.2 Logging In

We will apply some of this theory back into our code. Every web application has a login page, unless you have a central authorization service such as Active Directory handling that for you. For GetOrganized, we don't want other users to access our thoughts and to-do lists (like our boss finding out we are planning on asking for a well-deserved raise). For these next few features, we'll work with action filters as well as the standard Microsoft Membership API to implement the feature of logging in.

Out-of-the-Box Authentication

MVC gives us rudimentary security installed by default in the AccountController. We can administer all this from the ASP.NET Web Site Administration Tool. We'll set up our database so that we can save user information permanently, and we'll also learn how to use the action filter [Authorize] to prompt users to log in on secured pages.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Home Page for Test-Drive ASP.NET MVC
http://pragprog.com/titles/jmasp
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/jmasp.

Contact Us

Online Orders:	www.pragprog.com/catalog
Customer Service:	support@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com
Contact us:	1-800-699-PROG (+1 919 847 3884)