9

Extending the Controller

This chapter covers

* Understanding the Controller extensibility points
* Discovering the requirements for an Action
* Using Action Selectors
* Creating custom ActionResults
* Reducing Controller complexity with ActionResults

The ASP.NET MVC framework has a number of extensibility points built into the ControllerBase class. This chapter will review the out of the box functionality that uses these extensibility points. Additionally, we will demonstrate how to use the extensibility points to reduce complexity in your controllers. The ActionResult is one of those extensibility points, which can reduce an action's complexity. We will cover how attributes placed on an action method are used to modify its behavior. This includes action selectors which are used to determine which action should be executed and action filters which can modify the model returned from an action. Before covering the extensibility points of the Controller and ControllerBase base classes, it is important to learn that the controller concept is an extensibility point of its own. If your project requires additional flexibility that is not supported out of the box then you are not out of luck — the MVC Framework gives you full control to implement your own controller which could act radically different than the one provided in the framework.

9.1 The controller extensibility

The default controller implementation comes with some specific ideas for how action methods are selected, executed and extended. This functionality comes from the Controller base class in the framework. This class is the default implementation of the IController interface, which you could choose implement directly. This is a very simple interface which provides a single method, Execute(). By implementing this interface you can still use the routing and controller factory functionality of the framework and push the rest of framework to the side. You can see the IController interface definition in figure 9.1.

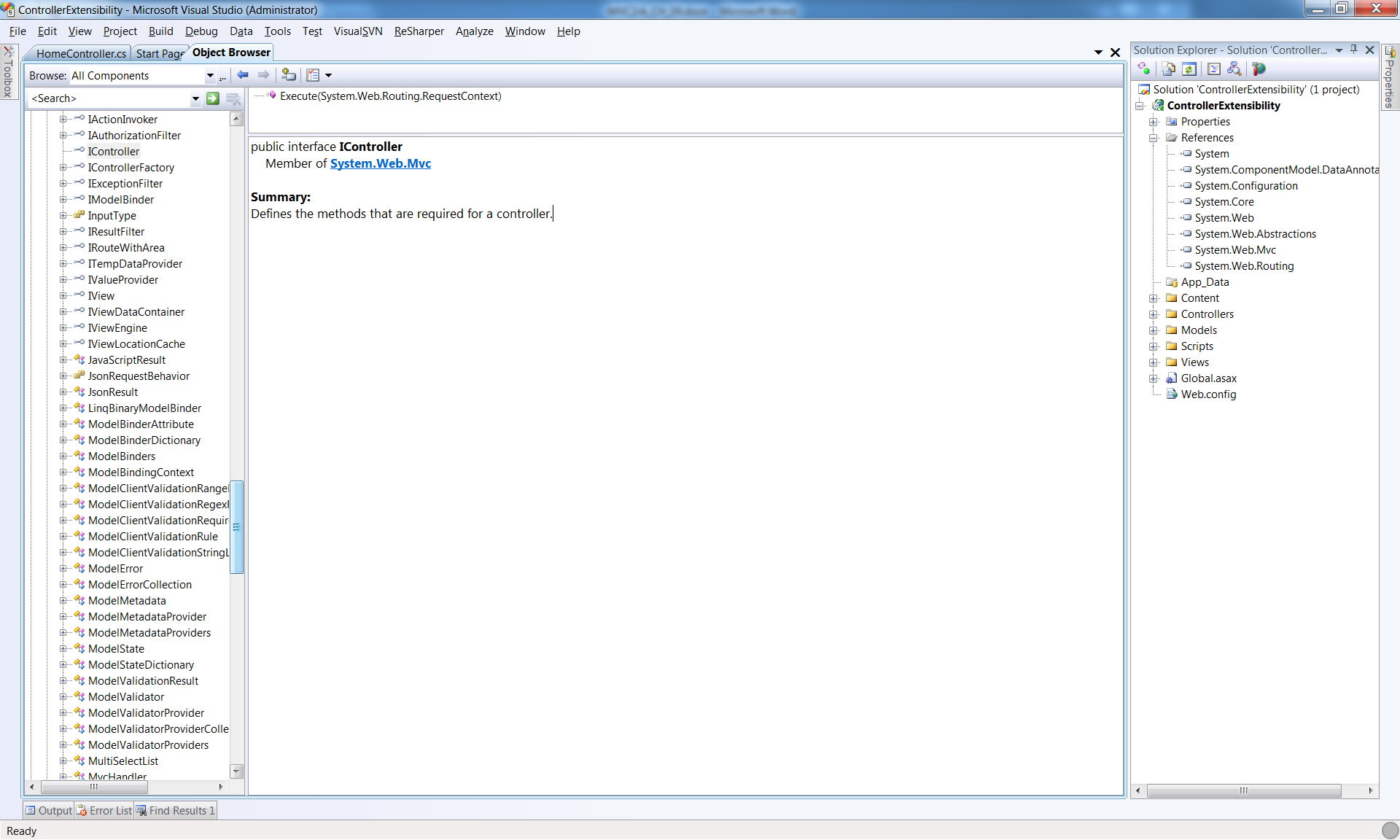
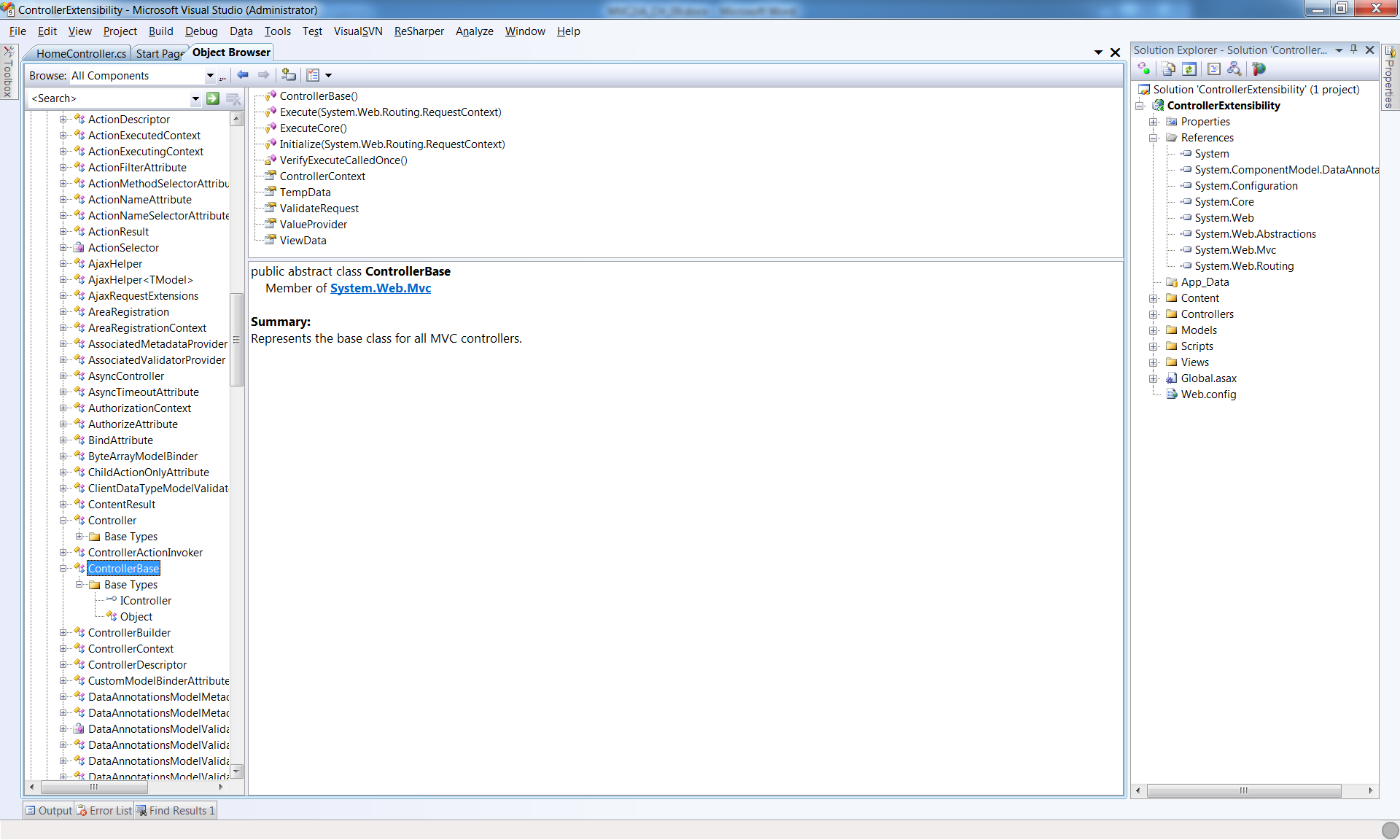


Figure 9.1 The IController interface exposes a single method, Execute.

There is a second option that is available for extensibility that is not as lean as implementing IController. The framework contains a ControllerBase class which provides the most basic properties to manage ViewData and TempData. The ControllerBase is listed in Figure 9.2. This is a pretty minimalistic class which still lets you take advantage of some concepts that are shared with the View. Although the interface and base class extensibility points exist in the framework, most developers and projects do not trade the productivity built into the framework's controller class for the power and extra work that is needed to implement your own IController implementation. The same goes for using the ControllerBase class. We need not sacrifice productivity when there are a number of extensibility points built into the Controller class. These are what we will cover next.

Figure 9.2 The ControllerBase class provides integration with routing as well as HttpContext.

9.2 Controller actions

Actions are the methods that control the main logic of each server request. However, not all methods of a controller class qualify to be an action. The requirements for a method to be web-callable as an action method are well documented on <http://www.asp.net/mvc>. In order to be considered as an action, the method:

* Must be public
* Cannot be static
* Cannot be an extension method
* Cannot be a constructor, getter, or setter
* Cannot have open generic types
* Is not a method of the Controller base class
* Is not a method of the ControllerBase base class
* Cannot contain ref or out parameters

If a method does not meet all of these requirements, it is not an action method. Now that you can create and identify action methods, we will discuss how to modify their behavior.

9.3 Action, Authorization, & Result Filters

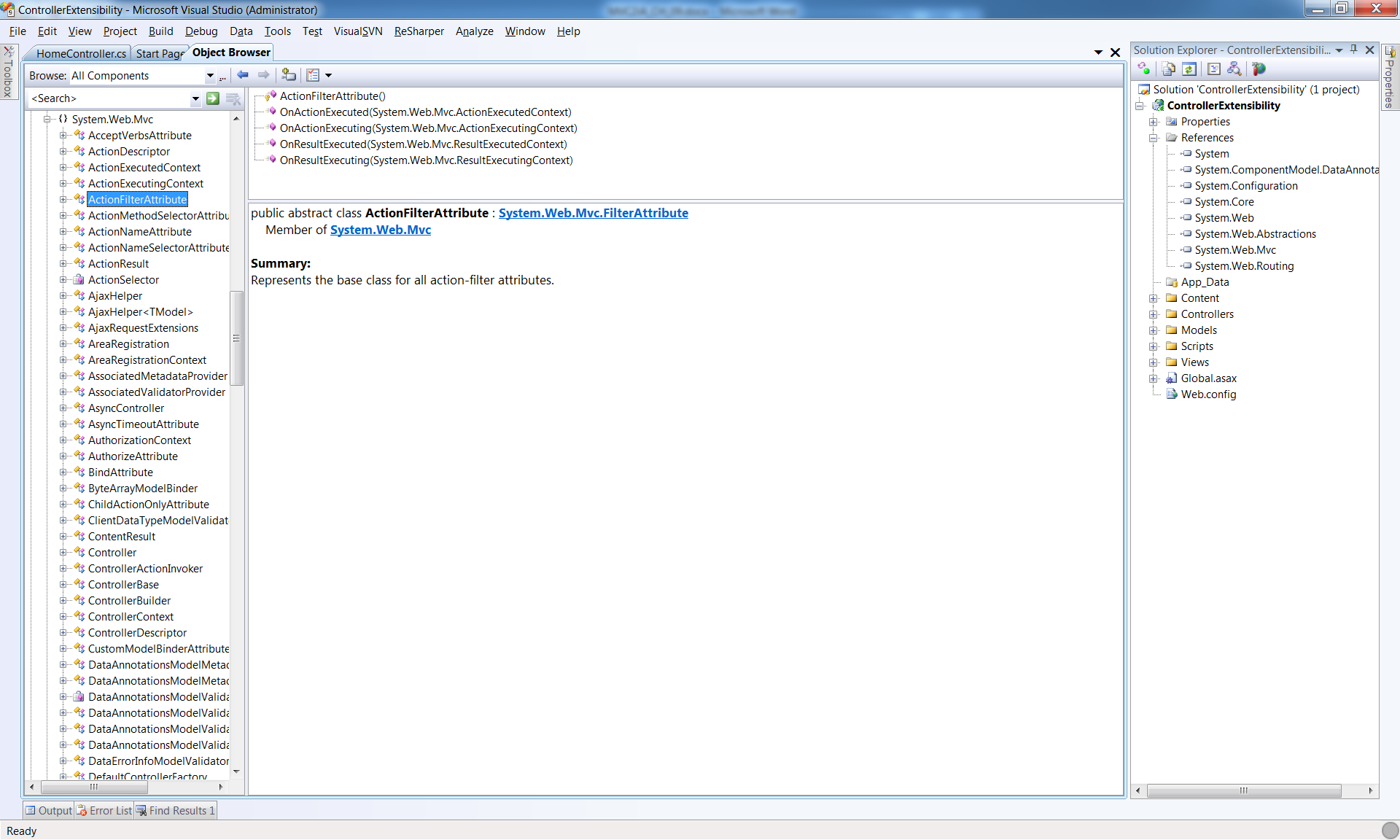
The first extensibility point of actions is through an ActionFilter. This extensibility point allows you to intercept the execution of an action and inject behavior before or after the execution of the action. This is very similar to aspect-oriented programming, which is a technique to apply cross cutting concerns across a code base without having lots of duplicate code to maintain. The easiest way to implement an action filter is to create a class that inherits from ActionFilterAttribute, although it is also possible to override methods on the Controller class itself. Figure 9.3 shows the methods that can be implemented to modify an action. This attribute actually implements the IActionFilter and IResultFilter interfaces, each of which provide different extensibility points.

Figure 9.3 The action filter extensibility point allows hooking several stages in the execution of the action and action result.

A new action filter that shipped with MVC 2 is the ChildActionOnlyAttribute. This filter implements the IAuthorizationFilter interface and is used by the framework to ensure that an action is only called from the RenderAction() method within a view. An action that has this attribute cannot be called through a top-level route and is not web callable.

Listing 9.1 Using the ChildActionOnlyAttribute

public class HomeController : Controller |A

{ |A

public ActionResult Index() |A

{ |A

return View(); |A

} |A

[ChildActionOnly] |B

public ActionResult ChildAction() |C

{ |C

return View(); |C

} |C

}

A- The HomeController has the default Action called Index.

B- The action filter is applied.

c- This action is not protected from being called directly from the web.

The code in listing 9.1 shows the ChildActionOnlyAttribute applied to the ChildAction method.

Listing 9.2 Calling a ChildAction from a View

<%Html.RenderAction("ChildAction"); %> |A

A - The execution of RenderAction method from within a View page.

The ChildActionOnlyAttribute attribute allows the method to be called by using RenderAction but not from a web browser using a direct url to the action.

How is account for filters in tests?

It may seem strange that the behavior defined in the attribute is called when the action is invoked. At runtime the method is not called directly; it is passed to the ControllerActionInvoker, which reads the action filters that are present on the controller and action. This is a nice extension point in the framework, as you are allowed to substitute your own IActionInvoker if you want to customize the semantics.

During unit tests, you will be calling action methods directly. None of the behavior defined in the action filters will be executed. Thus, you should treat your tests as if the action filters were executed (for example, load any data into ViewData that would have been loaded by an action filter). To test whether filter such as [Authorize] or [HttpPost] have been applied, you can easily test for the existence of the attribute by using reflection. Here is a class that can help you simplify the reflection code required to get attributes.

public static class ReflectionExtensions

{

public static TAttribute GetAttribute<TAttribute>(

this MemberInfo member) where TAttribute : Attribute

{

var attributes = member.GetCustomAttributes(typeof (TAttribute), true);

if (attributes != null && attributes.Length > 0)

return (TAttribute)attributes[0];

return null;

}

public static bool HasAttribute<TAttribute>(

this MemberInfo member) where TAttribute : Attribute

{

return member.GetAttribute<TAttribute>() != null;

}

}

The usage of this extension method is as follows:

type.GetMethod("Index").HasAttribute<AcceptVerbsAttribute>()…

The extension method accepts the attribute type as a generic parameter and then ensures that the method in question is marked with that attribute.

9.4 Action Selectors

The next extensibility point is the ActionMethodSelector. An action selector is very different from an action filter. The two are often confused because they are both applied to action methods by using attributes. The action selector is used to control which action method is selected to handle a particular route. There are a number of built in action selectors, each one is used to filter down the actions so that you can have an action for a very specific scenario. The list in Figure 9.4 shows the action selectors that come with the framework. The common use for an action select is to create an overloaded action to fulfill a route that differs only by the HTTP verb that is sent to the web server. A concrete example of this is to have two action methods named "Edit". One has the HttpGetAttribute applied and would render an edit form to the browser. The other overload for the edit method would have the HttpPostAttribute applied to it and it would take a view model as a parameter. By doing this, the code in the view is simplified because the form from the first action is posted to the same URL. Essentially, the HTTP Method is used to differentiate which overload should be invoked.

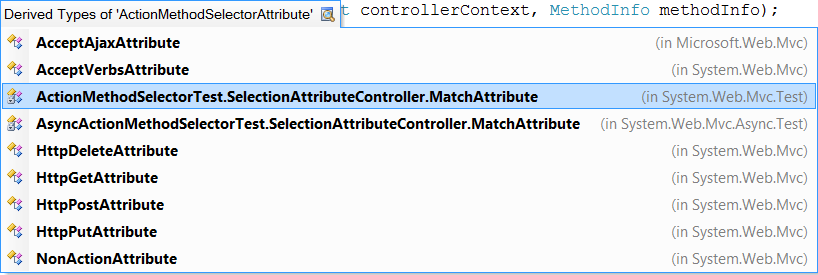


Figure 9.4 Action Selector

9.5 Using action results to reduce complexity

Custom action results can be used to remove code that is duplicated across methods as well as extracting dependencies that can make an action difficult to test. A great way to use a custom action result is to compose functionality on top of an out-of-the-box ActionResult, like the ViewResult or RedirectResult.

9.5.1 Removing duplication with an action result

To remove the duplication in multiple similar action methods, you can extract the majority of the code and move it into an action result. The sample below demonstrates how to take the logic for creating a comma separated values (CSV) file from a collection of objects and encapsulate it within an action result.

Listing 9.3 the CsvActionResult class

public class CsvActionResult : ActionResult

{

public IEnumerable ModelListing { get; set; } |1

public CsvActionResult(IEnumerable modelListing) |2

{ |2

ModelListing = modelListing; |2

} |2

public override void ExecuteResult(ControllerContext context) |3

{

byte[] data = new CsvFileCreator().AsBytes(ModelListing); |3

var fileResult = new FileContentResult(data, "text/csv") |3

{ |3

FileDownloadName = "CsvFile.csv"; |3

} |3

fileResult.ExecuteResult(context); |3

}

}

public class CsvFileCreator

{

public byte[] AsBytes(IEnumerable modelList) |4

{

StringBuilder sb = new StringBuilder(); |4

BuildHeaders(modelList, sb); |5

BuildRows(modelList, sb); |6

return sb.AsBytes();

}

private void BuildHeaders(IEnumerable modelList, StringBuilder sb) |5

{

foreach (PropertyInfo property in modelList.GetType().GetElementType().GetProperties())

{

sb.AppendFormat("{0},",property.Name);

}

sb.NewLine();

}

private void BuildRows(IEnumerable modelList, StringBuilder sb) |6

{

foreach (object modelItem in modelList)

{

BuildRowData(modelList, modelItem, sb);

sb.NewLine();

}

}

private void BuildRowData(IEnumerable modelList, object modelItem, StringBuilder sb) |6

{

foreach (PropertyInfo info in modelList.GetType() .GetElementType() .GetProperties())

{

object value = info.GetValue(modelItem, new object[0]);

sb.AppendFormat("{0},", value);

}

}

}

1 - Data to be rendered as CSV

2 - Constructor takes data to render

3 - Create output

4 - Converts data to byte array

5 - Builds header row for CSV file

6 - Builds rows of the CSV file

This listing shows how a call to the CsvFileCreator class has been moved into a custom action result called CsvActionResult. This action result is then responsible for instantiating and executing the CsvFileCreator as well as setting the appropriate content type for the file that is streamed to the users browser.

Listing 9.4 The simplified Action method that uses the CsvActionResult

public ActionResult ExportUsers()

{

IEnumerable<User> model = UserRepository.GetUsers();

return new CsvActionResult(model);

}

This listing shows how clean the ExportUsers action is as a result of moving the logic to create the CSV file into an action result. We have seen that most developers will first lean to putting this type of logic into the action which means the action method is hard to test and contains logic which may be duplicated in other action methods in the application. Duplication in code is something that you want to reduce so that maintenance of your codebase is easier.

The action method code for rendering the CsvFileResult is now clean and easy to understand. The simple act of abstracting the logic and putting it into an action result, allows for some reuse in your application as well. It is now pretty trivial to add more CSV exports to the application because the logic was put into an action result.

9.5.2 Using ActionResult to abstract hard to test dependencies

Another great use for creating action results is to abstract hard to test dependencies. While the MVC Framework gives you a lot of control around using the framework and creating controllers, there are still some features of ASP.NET which are difficult to simulate in a test. By taking that hard-to-test code out of an action and putting it into the Execute method of an action result, the actions become significantly easier to unit test. The reason for this is that when you unit test an action, you assert the type of action result that the action return and state of the action result. The execute method of the action result is not executed as part of the unit test.

Listing 9.5 Moving hard to test code into an ActionResult

public class LogoutActionResult : ActionResult

{

public RedirectToRouteResult ActionAfterLogout { get; set; } |A

public LogoutActionResult(RedirectToRouteResult actionAfterLogout) |B

{

ActionAfterLogout = actionAfterLogout |B

}

public override void ExecuteResult(ControllerContext context)

{

FormsAuthentication.SignOut(); |C

ActionAfterLogout.ExecuteResult(context); |D

}

}

A - RedirectToRouteResult is testable

B - Constructor sets ActionAfterLogout

C - SignOut is hard to test

D - ActionAfterLogout result is executed.

Listing 9.5 shows how moving the FormsAuthentication.SignOut() call from an action and into the action result, abstracts that line of code and prevents it from executing from within the action method. This allows an action to return a LogoutActionResult and the testing of that method does not have to deal with calls to the FormsAuthentication class. The test can just assert that the LogoutActionResult was returned from the action. The test can also assert the values in the RedirectToRouteResult to make sure that the action correctly setup the redirect.

Listing 9.6 Action method that uses the LogoutActionResult

public ActionResult Logout()

{

return new LogoutActionResult(RedirectToAction("Index","Home")); |A

}

A - The testable Logout action method.

Listing 9.6 shows that the Logout action method returns the new LogoutActionResult method. The constructor parameter to the LogoutActionResult is a RedirectToAction result that will redirect the browser to the Index action on the HomeController.

9.6 Summary

The advanced controller extensibility points shown in this chapter allow you the ability to tweak the framework easily. The IController interface allows the most control, but the various controller base classes provide some very useful but flexible capabilities. Actions help you easily break down basic functions of a single controller. Action filters provide hooks for inserting code before or after action execution. Action selectors help you provide hints to the action invoker about which action should be selected for execution, and action results help encapsulate repetitive rendering logic.

The examples demonstrated in this chapter will allow you to get the most from your controllers and allow cross cutting concerns to be easily applied throughout your application and reduce code duplication. Both of these should enable better application maintenance. Now that we have shown some advanced controller extensibility seams, the next chapter will walk you through advanced view techniques.