9

Controller, Actions, & ActionResults , Action Selectors

This chapter covers

* Understanding the Controller extensibility points
* Discover 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 Actions complexity. We will cover how attributes placed on a Action method are used to modify its behavior. This includes Action Selectors which are used to determine which Action should be executed, Action Filters which can modify the Model which is returned from an Action. Before covering the extensibility points of the Controller base class, it is important to learn that the Controller concept is an extensibility point of its own. If your project needed some concept that just could not fit into the existing Action extensibility points you are not out of luck, the MVC Framework gives you the 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 concept of the Controller comes with some specific ideas of how Actions are selected, executed and extended. This functionality comes from the Controller base class in the framework. The framework does allow for an extensibility point that sits in front of the ControllerBase class. This is the IController interface. 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.



Figure 9.1 The IController interface

There is a second option that is available for extensibility that is not as lean as implementing IController. The Framework contains a ControllerBase 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 Frameworks 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, why trade productivity when there are a number of extensibility points built into the Controller class, which we will cover next.

Figure 9.2 The ControllerBase class

9.2 Controller Actions

As we learned earlier in chapter 4, Actions are the place where your code lives to control the main logic of each server request. First it is important to know the method signature requirements for an method to be an Action in a class that inherits from Controller.

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 an Action the method:

* Must be public
* Cannot be a static method
* 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

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 ActionFilter is to inherit from the ActionFilterAttribute. Figure 9.3 shows the methods that can be implemented to modify an action. This attribute actually implements the IActionFilter and IResultFilter interfaces, each allow for a different entry points for your extension.

Figure 9.3 The Action Filter extensibility

A new ActionFilter which shipped with MVC 2 is the ChildActionOnlyAttribute. This filter implements the IAuthorizationFilter interface and is used by the Framework to ensure that a Action is only called from the RenderAction method from within a View. An Action that has this attribute cannot be called through a 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. This attribute allows the method to be called from a RenderAction but not from a web browser using a direct url to the Action.

Listing 9.2 Calling a ChildAction from a View

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

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

How is code in the action filter called?

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). For things like [Authorize] or [AcceptVerbs(HttpVerbs.POST)] you can easily test the existence of the attribute with 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;

 }

}

Usage is simple:

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

9.4 Action Selectors

The next extensibility point is the ActionSelector. An ActionSelector is very different from an ActionFilter. The two are often confused because they are both applied to Action methods by using attributes. The ActionSelector is used to modify how an Action is selected to fulfill a route. There are a number of built in ActionSelectors, 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 ActionSelectors 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 webserver. A concrete example of this is to have two Action methods named Edit. One has the HttpGetAttribute applied, this action would render an edit form to the browser. The overload for this 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 form is simplified because the form from the first Action is posted to the same url. It only differs by the HttpVerb.



Figure 9.4 Action Selectors

9.5 Using ActionResults to reduce complexity.

We have covered how to return an ActionResult or an object that derives from an ActionResult in chapter 4. Using a custom ActionResult can provide the following benefits. They can be used to remove code that is duplicated across methods. They can also be used to extract dependencies that are difficult to test. A great way to use a custom ActionResult 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.

A great way to remove duplication in multiple Controller Action methods is to extract a majority of that code and move it into an ActionResult. The sample below demonstrates that by putting the logic to create a comma separated values (csv) file into an ActionResult. This example show the ActionResult that can take an existing model that implements IEnumerable (meaning it is a list of items) and dynamically determines the field names and formats the values.

Listing 9.3 the CsvActionResult class

public class CsvActionResult : ActionResult

{

 public IEnumerable ModelListing { get; set; } |A

 public CsvActionResult(IEnumerable modelListing) |B

 { |B

 ModelListing = modelListing; |B

 } |B

 public override void ExecuteResult(ControllerContext context) |C

 {

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

 new FileContentResult(data, "text/csv").ExecuteResult(context); |C

 }

}

public class CsvFileCreator

{

 public byte[] AsBytes(IEnumerable modelList) |D

 {

 StringBuilder sb = new StringBuilder(); |D

 BuildHeaders(modelList, sb); |E

 BuildRows(modelList, sb); |F

 return sb.AsBytes(); |G

 }

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

 {

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

 {

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

 }

 sb.NewLine();

 }

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

 {

 foreach (object modelItem in modelList)

 {

 BuildRowData(modelList, modelItem, sb);

 sb.NewLine();

 }

 }

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

 {

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

 {

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

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

 }

 }

}

A - Shows the property that stores the IEnumerable Model which is the data for the Csv File.

B - Shows the constructor which takes the model as the only parameter, it is than stored as a property on the class.

C - Shows the ExecuteResult method which will be called by the runtime to execute the ActionResult. This method ties together the IEnumerable model and passes it to the CsvFileCreator.

D - The AsBytes method is the entry point into the CsvFileCreator class. The first thing this method does is it creates a StringBuilder. It than goes on to orchestrate the other actions which need to be addressed to create the Csv file.

E - This code creates the Header row for the file. This uses some reflection to determine the name of all of the models fields, and then it concatenates them into a properly formatted Csv header.

F -

This listing shows how a call to the CsvFileCreator class has been moved into an custom ActionResult called CsvActionResult. This ActionResult is only responsible for executing the CsvFileCreator and 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 comma separated list file into an ActionResult. 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 will 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 csv FileResult is now clean and easy to understand. The simple act of abstracting the logic and putting it into an ActionResult, 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 ActionResult.

9.5.2 Using ActionResult to abstract hard to test dependencies.

Another great use for creating ActionResults 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 mock in a test. By taking that hard to test code out of an Action and putting it into the Execute method of an ActionResult 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 ActionResult that the Action return and state of the ActionResult. The execute method of the ActionResult is not executed as part of the unit test.

Queballs in text

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 - The out of the box ActionResult that can be unit tested

B - The constructor is used to set the ActionAfterLogout

C - The SignOut is the hard to test dependency

D - The ActionAfterLogout is executed.

Listing 9.5 shows how moving the FormsAuthentication.SignOut call from an Action and into the ActionResult, abstracts that line of code and prevents it from executing during the execution of the Action. This allows an Action to return a LogoutActionResult and the testing of that method does not have to deal with the dependency of the FormsAuthentication object. 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 Loutout 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 HomeController.Index Action.

Summary

The advanced controller extensibility points shown in this chapter allow you the ability to tweak the framework easily to form to your project. 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.