

Extending the controller

This chapter covers

- Understanding the controller extensibility points
- Discovering the requirements for an action
- Using action selectors
- Creating custom action results
- Reducing controller complexity with action results

The ASP.NET MVC framework has a number of extensibility points built into the `ControllerBase` class, and this chapter will review the out-of-the-box functionality that uses these extensibility points. Additionally, we'll demonstrate how to use the extensibility points to reduce complexity in controllers.

The `ActionResult` is one of those extensibility points that can reduce an action's complexity. We'll cover how attributes placed on an action method are used to modify its behavior, including action selectors that can determine which action should be executed and action filters that can modify the model returned from an action.

Before covering the extensibility points of the `Controller` and `ControllerBase` base classes, it's important to learn that the controller is an extensibility point of its own. If your project requires additional flexibility that isn't supported out of the

box, you're not out of luck—the MVC Framework gives you full control to implement your own controller, which could act radically differently than the one provided in the framework.

9.1 Controller extensibility

The default controller implementation comes with some specific ideas about how action methods are selected, executed, and extended. This functionality comes from the `Controller` base class in the ASP.NET MVC framework, which is the default implementation of the `IController` interface.

`IController` is a simple interface that provides a single method, `Execute()`, and you could choose to implement it directly. By implementing this interface, you can still use the routing and controller factory functionality of the framework and push the rest of the framework to the side.

You can see the `IController` interface definition in figure 9.1.

A second extensibility option is available that isn't as lean as implementing `IController`. The framework contains a `ControllerBase` class that provides the most basic properties for managing `ViewData` and `TempData`. The `ControllerBase` class is listed in figure 9.2. It's a pretty minimal class but it 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, few developers and projects trade the productivity built into the framework's controller class for the power and extra work that's needed to implement their own `IController` implementation. The same goes for using the `ControllerBase` class. We needn't sacrifice productivity because a number of extensibility points are built into the `Controller` class. We'll cover them next.

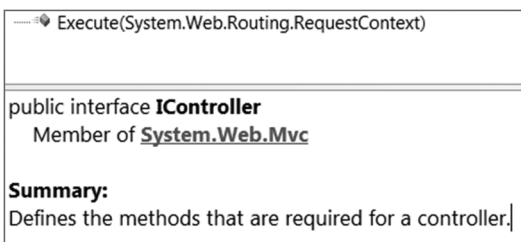


Figure 9.1 The `IController` interface exposes a single method, `Execute()`.

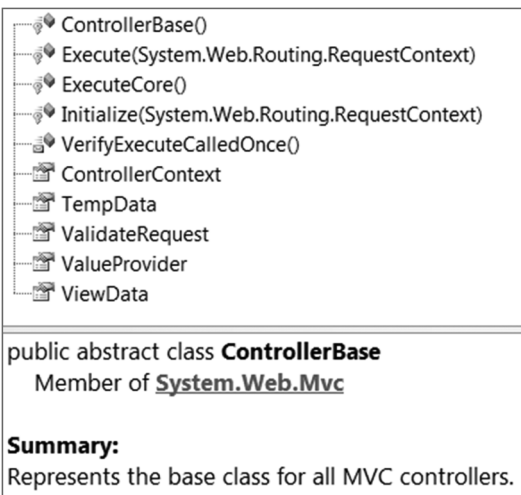


Figure 9.2 The `ControllerBase` class provides integration with routing as well as `HttpContext`.

9.2 Controller actions

Actions are the methods that control the main logic of each server request, but 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 Microsoft's ASP.NET MVC site (www.asp.net/mvc).

To be considered as an action, the method must meet the following requirements:

- It must be public.
- It can't be static.
- It can't be an extension method.
- It can't be a constructor, getter, or setter.
- It can't have open generic types.
- It can't be a method of the Controller base class.
- It can't be a method of the ControllerBase base class.
- It can't contain ref or out parameters.

If a method doesn't meet all these requirements, it isn't an action method.

Now that you can identify action methods, we'll discuss how to modify their behavior.

9.3 Action, authorization, and 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 action is executed. This is similar to aspect-oriented programming, which is a technique for applying cross-cutting concerns to 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's also possible to override methods on the Controller class itself.

Figure 9.3 shows the methods of `ActionFilterAttribute` that can be overridden to modify an action. This attribute implements the `IActionFilter` and `IResultFilter` interfaces, each of which provides different extensibility points.

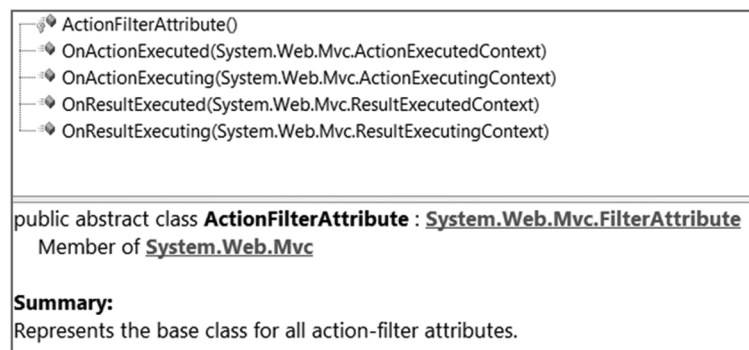


Figure 9.3
The action filter methods that can be overridden to modify an action.

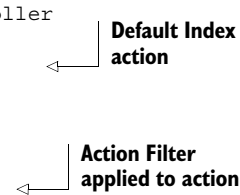
The new `ChildActionOnlyAttribute` action filter shipped with MVC 2. This filter implements the `IAuthorizationFilter` interface and is used by the framework to ensure that an action is only called from the `RenderAction()` method within a view. An action that has this attribute can't be called through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

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

Listing 9.1 Using the `ChildActionOnlyAttribute`

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View();
    }

    [ChildActionOnly]
    public ActionResult ChildAction()
    {
        return View();
    }
}
```



The `ChildActionOnly` attribute prevents the `ChildAction` method from being exposed as a web-callable action that can be invoked by a web browser. But it can still be invoked by making a call to `RenderAction` from within a view, as follows:

```
<%Html.RenderAction("ChildAction"); %>
```

Accounting 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 isn't called directly; it's 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, because you're allowed to substitute your own `IActionInvoker` if you want to customize the semantics.

During unit tests, you'll be calling action methods directly. None of the behavior defined in the action filters will be executed, so you should treat your tests as if the action filters were executed (for example, load any data into `ViewData` that would've been loaded by an action filter). To test whether filters such as `[Authorize]` or `[HttpPost]` have been applied, you can easily test for the existence of the attribute by using reflection.

Here's 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
    {
```

(continued)

```

        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;
    }
}

```

You can use this extension method 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 different from an action filter, but the two are often confused because they're 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 used to filter down the actions so that you can have an action for a specific scenario. The list in figure 9.4 shows the action selectors that come with the framework.

A common use for an action selector is to create an overloaded action to fulfill a route that differs only by the HTTP method that's sent to the web server. (Be aware that in this industry, the terms *HTTP method* and *HTTP verb* are used interchangeably.) A concrete example of this is to have two action methods named "Edit". One would










Derived Types of 'ActionMethodSelectorAttribute'	
 AcceptAjaxAttribute	(in Microsoft.Web.Mvc)
 AcceptVerbsAttribute	(in System.Web.Mvc)
 ActionMethodSelectorTest.SelectionAttributeController.MatchAttribute	(in System.Web.Mvc.Test)
 AsyncActionMethodSelectorTest.SelectionAttributeController.MatchAttribute	(in System.Web.Mvc.Async.Test)
 HttpDeleteAttribute	(in System.Web.Mvc)
 HttpGetAttribute	(in System.Web.Mvc)
 HttpPostAttribute	(in System.Web.Mvc)
 HttpPutAttribute	(in System.Web.Mvc)
 NonActionAttribute	(in System.Web.Mvc)

Figure 9.4 Action selectors in ASP.NET MVC

have the `HttpGetAttribute` applied and would render an edit form to the browser, and the other would have the `HttpPostAttribute` applied and would take a view model as a parameter. This simplifies the code in the view 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.

9.5 *Using action results to reduce complexity*

Custom action results can be used to remove code that's duplicated across methods and to extract 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. Listing 9.2 demonstrates how to take the logic for creating a comma-separated value (CSV) file from a collection of objects and encapsulate it within an action result.

Listing 9.2 The `CsvActionResult` class

```
public class CsvActionResult : ActionResult
{
    public IEnumerable ModelListing { get; set; }
    public CsvActionResult(IEnumerable modelListing)
    {
        ModelListing = modelListing;
    }
    public override void ExecuteResult(
        ControllerContext context)
    {
        byte[] data = new CsvFileCreator()
            .AsBytes(ModelListing);

        var fileResult = new FileContentResult(
            data, "text/csv")
        {
            FileName = "CsvFile.csv";
        }
        fileResult.ExecuteResult(context);
    }
}

public class CsvFileCreator
{
    public byte[] AsBytes(IEnumerable modelList)
    {
        StringBuilder sb = new StringBuilder();
        BuildHeaders(modelList, sb);
        BuildRows(modelList, sb);
        return sb.AsBytes();
    }
}
```

Stores data to render (points to `ModelListing`)

Takes data to render (points to `CsvActionResult` constructor)

Creates output (points to `ExecuteResult` method)

Converts data to byte array (points to `AsBytes` method)

Builds header row for CSV file (points to `BuildHeaders` method)

Builds rows of CSV file (points to `BuildRows` method)

```

    }

    private void BuildHeaders(
        IEnumerable modelList, StringBuilder sb)
    {
        foreach (PropertyInfo property in
            modelList.GetType().GetElementType().GetProperties())
        {
            sb.AppendFormat("{0},{0}", property.Name);
        }
        sb.NewLine();
    }

    private void BuildRows(
        IEnumerable modelList, StringBuilder sb)
    {
        foreach (object modelItem in modelList)
        {
            BuildRowData(modelList, modelItem, sb);
            sb.NewLine();
        }
    }

    private void BuildRowData(
        IEnumerable modelList, object modelItem,
        StringBuilder sb)
    {
        foreach (PropertyInfo info in
            modelList.GetType().GetElementType().GetProperties())
        {
            object value = info.GetValue(modelItem, new object[0]);
            sb.AppendFormat("{0},{0}", value);
        }
    }
}

```

Builds header row for CSV file

Builds rows of CSV file

Listing 9.2 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's streamed to the user's browser.

Listing 9.3 shows how clean the `ExportUsers` action is as a result of moving the logic to create the CSV file into the `CsvActionResult` action result.

Listing 9.3 The simplified action method that uses `CsvActionResult`

```

public ActionResult ExportUsers()
{
    IEnumerable<User> model = UserRepository.GetUsers();
    return new CsvActionResult(model);
}

```

We've seen that most developers will first lean toward putting this type of logic into the action, which means the action method is hard to test and contains logic that may be duplicated in other action methods in the application. Duplication in code is something you want to reduce so that maintaining your code base is easier.

The action method code for rendering the `CsvActionResult` is now clean and easy to understand, and the simple act of abstracting the logic and putting it into an action result allows for some reuse. It's now pretty trivial to add more CSV exports to the application because the logic is in an action result.

9.5.2 *Using action results to abstract hard-to-test dependencies*

Another great use for action results is to abstract hard-to-test dependencies. Although the MVC Framework gives you a lot of control when using the framework and creating controllers, there are still some features of ASP.NET that 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, you ensure that the actions become significantly easier to unit-test. That's because when you unit-test an action, you assert the type of action result that the action returns and the state of the action result. The `Execute` method of the action result isn't executed as part of the unit test.

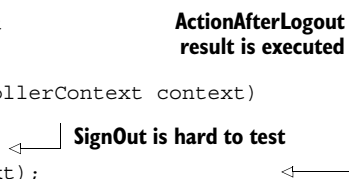
Listing 9.4 shows a `LogoutActionResult` that encapsulates the hard-to-test `FormsAuthentication.SignOut` method.

Listing 9.4 Moving hard-to-test code into an `ActionResult`

```
public class LogoutActionResult : ActionResult
{
    public RedirectToRouteResult ActionAfterLogout {
        get; set; }

    public LogoutActionResult(RedirectToRouteResult actionAfterLogout)
    {
        ActionAfterLogout = actionAfterLogout
    }

    public override void ExecuteResult(ControllerContext context)
    {
        FormsAuthentication.SignOut();
        ActionAfterLogout.ExecuteResult(context);
    }
}
```

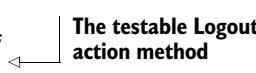


The diagram illustrates the execution flow within the `LogoutActionResult` class. A vertical line on the right side of the code block represents the execution path. An arrow points from the `ExecuteResult` method call to the `FormsAuthentication.SignOut()` call. Another arrow points from the `FormsAuthentication.SignOut()` call to the `ActionAfterLogout.ExecuteResult(context)` call. A third arrow points from the `ActionAfterLogout.ExecuteResult(context)` call to the `ActionAfterLogout` property access. A final arrow points from the `ActionAfterLogout` property to its `ExecuteResult` method. Annotations with arrows point to these steps: "SignOut is hard to test" points to `FormsAuthentication.SignOut()`, and "ActionAfterLogout result is executed" points to the `ActionAfterLogout.ExecuteResult(context)` call.

Listing 9.4 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`, as in listing 9.5, and the testing of that method doesn't 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 set up the redirect.

Listing 9.5 Action method that uses the `LogoutActionResult`

```
public ActionResult Logout()
{
    var redirect = RedirectToAction("Index", "Home");
    return new LogoutActionResult(redirect);
}
```



The diagram shows the execution flow for the `Logout` action method. An arrow points from the `Logout` method to the `RedirectToAction` call. Another arrow points from the `RedirectToAction` call to the `LogoutActionResult` constructor call. An annotation "The testable Logout action method" points to the `Logout` method.

Listing 9.5 shows that the Logout action method returns the new LogoutActionResult method. The constructor parameter to the LogoutActionResult is a RedirectToActionResult 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 to tweak the framework easily. The IController interface provides the most control, but the various controller base classes offer some useful but flexible capabilities.

Actions help you easily break down basic functions of a single controller, and action filters provide hooks for inserting code before or after action execution. Action selectors help you supply 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 help you 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've seen some advanced controller extensibility seams, the next chapter will walk you through advanced view techniques.