ASP.NET MVC 4 IN ACTION

Jeffrey Palermo
Jimmy Bogard
Eric Hexter
Matthew Hinze
Jeremy Skinner

FOREWORD BY
Phil Haack

MANNING

ASP.NET MVC 4 in Action

Jeffrey Paleermo, Jimmy Bogard
Eric Hexter, Matthew Hinze
and Jeremy Skinner

Chapter 4

brief contents

4 Action-packed controllers

This chapter covers

- What makes a controller
- What belongs in a controller
- Manually mapping view models
- Validating user input
- Using the default unit test project

In the last couple of chapters, we've looked at the basics of creating a simple Guestbook application and at different options available for passing data to views. In this chapter, we'll finish off the Guestbook example by looking at controllers in a bit more detail. We'll explore what should (and shouldn't) be part of a controller and look at how to manually construct view models, validate simple user input, and write controller actions that don't use a view. This will give us a good set of building blocks for constructing the most common types of controller actions.

We'll also briefly introduce you to unit testing controller actions so you can verify that they're working correctly. We'll start off by looking at the default unit test project and then move on to creating unit tests for the `GuestbookController` that we've been working with in previous chapters.

But before we dive into these new concepts, let's quickly recap the purpose of controllers and actions.

4.1 Exploring controllers and actions

As you saw in chapter 2, a controller is one of the core components of an ASP.NET MVC application. It's a class that contains one or more public methods (actions) that correspond to a particular URL. These actions act as the "glue" in your applications, bringing together data from the model with the user interface of the application (the view), so it's important to understand how these actions work. In this section you'll gain a better understanding of how controller actions work as we briefly explore the anatomy of a controller and look at what should typically be part of a controller action.

But first, let's remind ourselves of what a controller action looks like. Here is the `Index` action of our `GuestbookController`.

Listing 4.1 The `Index` action of the `GuestbookController`

```
public class GuestbookController : Controller                    <-- Inherits from Controller
{
    private GuestbookContext _db = new GuestbookContext();

    public ActionResult Index()                                  <-- Exposes public action method
    {
        var mostRecentEntries = (from entry in _db.Entries
                                 orderby entry.DateAdded descending
                                 select entry).Take(20);

        var model = mostRecentEntries.ToList();
        return View(model);
    }
}
```

Our controller consists of a class that inherits from `Controller` and contains public methods that define actions. In chapter 2, we mentioned that all controllers have to inherit from the `Controller` base class, but this isn't strictly true—controllers don't have to inherit from `Controller`, but the framework requires that they must at least implement the `IController` interface or they won't be able to handle web requests.

In order to understand just how the framework decides whether a class should be treated as a controller, let's examine the `IController` interface in more detail.

4.1.1 IController and the controller base classes

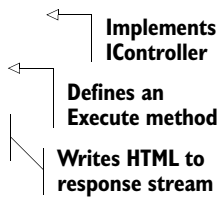
The `IController` interface defines the most basic element of a controller—a single method called `Execute` that receives a `RequestContext` object:

```
public interface IController
{
    void Execute(RequestContext requestContext);
}
```

The simplest type of controller could implement this interface and then write some HTML out to the response stream:

Listing 4.2 Implementing `IController` manually

```
public class SimpleController : IController
{
    public void Execute(RequestContext requestContext)
    {
        requestContext.HttpContext.Response
          .Write("<h1>Welcome to the Guest Book.</h1>");
    }
}
```



This controller implements the `IController` interface by defining an `Execute` method. Inside this method, we can directly access the `HttpContext`, `Request`, and `Response` objects. This way of defining controllers is very close to the metal but isn't very useful. We have no way to render views directly, and we end up mixing presentation concerns with controller logic by writing HTML directly from within the controller. Additionally, we bypass all of the framework's useful features, such as security (which we'll look at in chapter 8), model binding (chapter 10), and action results (chapter 16). We also lose the ability to define action methods—all requests to this controller are handled by the `Execute` method.

In reality, it's unlikely that you'll need to implement `IController` because it isn't particularly useful by itself (but the option is there if for some reason you need to bypass the majority of the framework). Usually you'll inherit from a base class instead. There are two to pick from—`ControllerBase` and `Controller`.

INHERITING FROM CONTROLLERBASE

The `ControllerBase` class directly implements `IController` but contains the infrastructure necessary for several of the features we've already looked at. For example, `ControllerBase` contains the `ViewData` property that you've seen can be used to pass data to a view. However, by itself `ControllerBase` still isn't very useful—there still isn't any way to render views or use action methods. This is where `Controller` comes in.

INHERITING FROM CONTROLLER

The `Controller` class inherits from `ControllerBase`, so it includes all of the properties `ControllerBase` defines (such as `ViewData`) but adds a significant amount of additional functionality. It contains the `ControllerActionInvoker`, which knows how to select a particular method to execute based on the URL, and it defines methods such as `View`, which you've already seen can be used to render a view from within a controller action. This is the class that you'll inherit from when creating your own controllers (and which we'll continue to use throughout the examples in this book). There typically isn't a reason or a benefit to directly inheriting from `ControllerBase` or `IController`, but it's useful to know that they exist because of the important part they play in the MVC pipeline.

Now that we've looked at how a class becomes a controller, let's move on to look at what makes an action method.

4.1.2 What makes an action method

In chapter 2, you saw that action methods are public methods on a controller class. (Actually, the rules determining whether a method should be an action are a bit more complex than this—we'll explore this in chapter 16.) Typically an action method returns an instance of an `ActionResult` (such as `ViewResult` when you call `return View()`).

But they don't necessarily have to return an `ActionResult`. For example, an action could have a void return type and instead write out to the response stream directly (much like the `SimpleController` in listing 4.2):

```
public class AnotherSimpleController : Controller
{
    public void Index()
    {
        Response.Write("<h1>Welcome to the Guest Book.</h1>");
    }
}
```

The same result could be achieved by directly returning a snippet of HTML from the controller action:

```
public class AnotherSimpleController : Controller
{
    public string Index()
    {
        return "<h1>Welcome to the Guest Book.</h1>";
    }
}
```

This works because the `ControllerActionInvoker` ensures that the return value of an action is always wrapped in an `ActionResult`. If the action returns an `ActionResult` already (such as a `ViewResult`), then it is simply invoked. However, if the action returns a different type (in this case, a string) then the return value is wrapped in a `ContentResult` object that simply writes it out to the response stream. The end result is the same as using a `ContentResult` directly:

```
public class AnotherSimpleController : Controller
{
    public string Index()
    {
        return Content("<h1>Welcome to the Guest Book.</h1>");
    }
}
```

This means that for simple actions, you could render HTML markup directly to the browser without the need for a view. However, this isn't usually done in real-world applications. It's better to keep presentation separate from the controller by relying on views instead. This makes it easier to change the application's user interface without needing to change controller code.

In addition to rendering markup or returning a view, there are other types of action results available. For example, you can redirect the user's browser to another

page by returning a `RedirectToRouteResult` (which you used when calling the `RedirectToAction` method back in chapter 2, listing 2.7) or return other types of content such as JSON (which we'll explore in chapter 7 when we look at Ajax).

You can also prevent public methods on controllers from being actions by decorating them with the `NonActionAttribute`:

```
public class TestController : Controller
{
   [NonAction]
   public string SomePublicMethod()
   {
      return "Hello World";
   }
}
```

`NonActionAttribute` is an example of an *action method selector* that can be used to override the default behavior of matching a method name to an action name. `NonActionAttribute` is the simplest kind of selector, which simply excludes a method from being accessible via a URL. You already saw another example of an action selector in chapter 2—the `HttpPostAttribute` selector, which ensured that an action only responds to HTTP `POST` requests.

> **NOTE** It's a fairly rare occurrence to need to use the `NonActionAttribute`. If you find yourself with a public method on a controller that you don't want to be an action, it's probably a good idea to ask yourself whether the controller is the best place for it to be. If it's a utility method, it should probably be private instead. If the method has been made public for the sake of testability, then this might be an indication that it should be extracted to a separate class.

Now that we've looked briefly at what constitutes an action, you can see the different ways in which you can send content to the browser. As well as rendering views, you can also directly send content and perform other actions such as redirects. All of these techniques can be useful in your own applications.

Let's now look at what type of logic should be *inside* an action method.

4.2 What should be in an action method?

One of the major benefits of the MVC pattern is the separation of concerns that keeps user-interface and presentation logic away from application code, thereby making the application easier to maintain. But it can be easy to negate these benefits if you're not careful to keep your controllers lightweight and focused.

The controller should act as a coordinator—it shouldn't really contain any business logic but instead act as a form of translation layer that translates user input (from the view) into objects that can be used by the domain (where the business logic lives) and vice versa. This is shown in figure 4.1.

Let's look at two common examples of tasks performed by a controller—manually mapping view models and accepting user input. First, to show how to map view models, we'll take our guestbook example and add a new page that needs to display data in

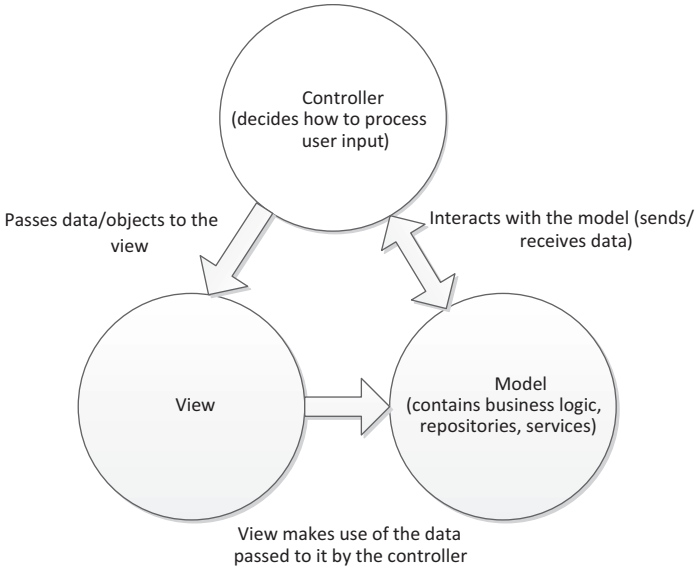


Figure 4.1 The controller coordinates between the model and the view.

a different format than how it's stored. Second, we'll add some validation to our page for adding entries to ensure that we can't store invalid data in our database. At the end of this section, you should have a basic understanding of how to build data structures specific for views (view models) and how to perform basic input validation.

4.2.1 Manually mapping view models

In chapter 3, we looked at the concept of strongly typed views and at a view model—a model object that's been created solely for the purpose of displaying data on a screen. So far in our examples, we've been using the same class (`GuestbookEntry`) as both our domain model and as our view model—it represents the data stored in the database, and it also represents the fields in our user interface.

For very small applications like our guestbook, this is sufficient, but as applications grow in complexity, it often becomes necessary to separate the two when the structure of a complex user interface doesn't necessarily map directly to the structure of the model. Because of this, we need to be able to convert instances of our domain model into view models.

As an example, let's add a new page to our Guestbook application that displays a summary of how many comments have been posted by each user, as shown in figure 4.2.

To create this screen, we'll first need to create a view model that contains one property for each column—the user's name and how many comments they've posted:

```
public class CommentSummary
{
    public string UserName { get; set; }
    public int NumberOfComments { get; set; }
}
```

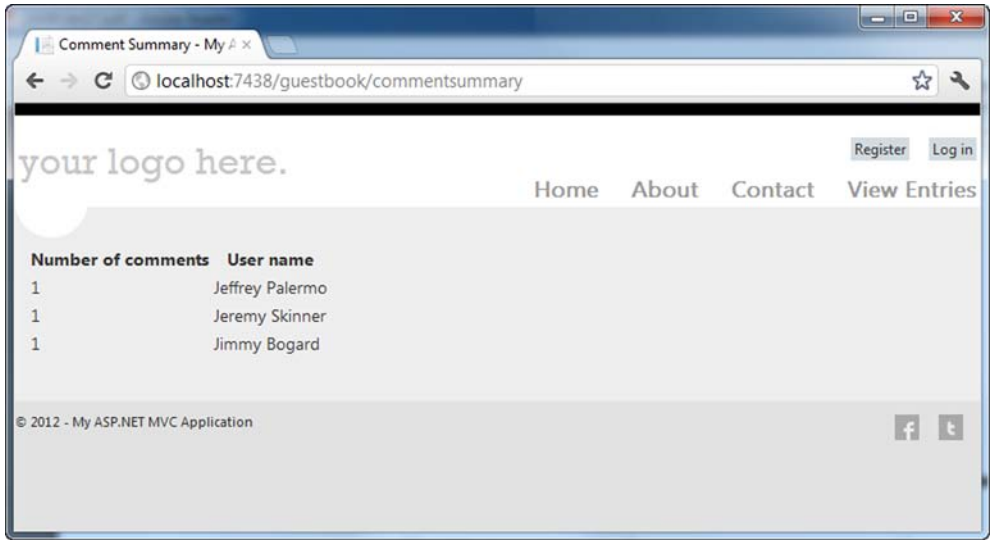


Figure 4.2 A simple comment summary screen

We now need to create a controller action (shown in listing 4.3) that will query the database to get the data necessary to display and then project it into instances of our `CommentSummary` class.

Listing 4.3 Projecting guestbook data into a view model

```
public ActionResult CommentSummary()
{
    var entries = from entry in _db.Entries
                  group entry by entry.Name
                  into groupedByName
                  orderby groupedByName.Count() descending
                  select new CommentSummary
                  {
                      NumberOfComments =
                           groupedByName.Count(),
                      UserName = groupedByName.Key
                  };

    return View(entries.ToList());
}
```

1 Retrieve guestbook data

2 Group data by username

3 Project into view model

4 Send view models to view

Here we're using LINQ to query our guestbook data 1 and group the comments by the name of the user that posted them 2. We then project this data into instances of our view model 3, which can then be passed to a view 4.

As the mapping logic here is fairly simple, keeping it in the controller action makes sense. But if the mapping became more complex (for example, if it required lots of data from many different sources in order to construct the view model), this

would be a good time to move the logic out of the controller action and into a separate, dedicated class to help keep our controller lightweight.

The corresponding view for our new screen is strongly typed and simply loops over the `CommentSummary` instances and displays them as rows in a table:

Listing 4.4 Displaying `CommentSummary` instances in a table

```
@model IEnumerable<Guestbook.Models.CommentSummary>

<table>
    <tr>
        <th>Number of comments</th>
        <th>User name</th>
    </tr>
    @foreach(var summaryRow in Model) {
        <tr>
            <td>@summaryRow.NumberOfComments</td>
            <td>@summaryRow.UserName</td>
        </tr>
    }
</table>
```

> **Automatically mapping view models**
>
> In addition to the manual projections we've shown here for mapping domain objects to view models, you could also make use of a tool, such as the open source AutoMapper, to achieve this with much less code. We'll look at how AutoMapper can be used with MVC projects in chapter 11.

We've only looked briefly at view models in this section, but we'll take a look at them in more detail in the next chapter, where we'll also explore the differences between view models and input models.

In addition to (simple) mapping operations, another common task for controller actions is performing validation on user input.

4.2.2 Input validation

Back in chapter 2, we looked at an example of accepting user input in the `Create` action of our `GuestbookController`:

```
[HttpPost]
public ActionResult Create(GuestbookEntry entry)
{
    entry.DateAdded = DateTime.Now;

    _db.Entries.Add(entry);
    _db.SaveChanges();
    return RedirectToAction("Index");
}
```

This action simply receives the input posted from the New Comment page in the form of a `GuestbookEntry` object (which has been instantiated by MVC's model-binding

process), sets the date, and then inserts it into the database. Although this works fine, it isn't really the best approach—we don't have any validation. As it is at the moment, a user can submit the form without entering their name or a comment. Let's improve on this by adding some basic validation.

The first thing we'll do is annotate the `Name` and `Message` properties of our `GuestbookEntry` class with `Required` attributes.

Listing 4.5 Applying validation attributes

```
public class GuestbookEntry
{
    public int Id { get; set; }

    [Required]
    public string Name { get; set; }

    [Required]
    public string Message { get; set; }

    public DateTime DateAdded { get; set; }
}
```

(1) Mark property as required

The `Required` attribute (1) resides in the `System.ComponentModel.DataAnnotations` namespace and provides a way to validate particular properties of an object. (There are several other attributes in this namespace too, such as `StringLengthAttribute`, which validates the maximum length of a string—we'll look at these validation attributes in more detail in chapter 6.)

Once annotated, MVC will automatically validate these properties when the `Create` action is invoked. We can check whether validation has succeeded or failed by checking the `ModelState.IsValid` property and then making a decision about what to do if validation fails. Here is the updated version of our `Create` action:

Listing 4.6 Checking whether validation succeeded

```
[HttpPost]
public ActionResult Create(GuestbookEntry entry)
{
    if (ModelState.IsValid)
    {
        entry.DateAdded = DateTime.Now;

        _db.Entries.Add(entry);
        _db.SaveChanges();
        return RedirectToAction("Index");
    }

    return View(entry);
}
```

(1) Check if validation succeeded

(2) Re-render form on failure

This time, instead of simply storing the new entry in the database, we first check whether `ModelState.IsValid` returns `true` (1). If it does, we continue to save the new entry as before. However, if it failed, we instead re-render the `Create` view, which allows the user to correct any problems before submitting again (2).

NOTE Keep in mind that calling `ModelState.IsValid` does not actually perform validation; it only checks to see whether validation has already succeeded or failed. The validation itself occurs just before the controller action is invoked.

We can display the error messages generated by the validation failure in our view by calling the `Html.ValidationSummary` method.

Listing 4.7 Displaying error messages in a view

```
@Html.ValidationSummary()

@using(Html.BeginForm()) {
    <p>Please enter your name: </p>
    @Html.TextBox("Name")

    <p>Please enter your message: </p>
    @Html.TextArea("Message", new{rows=10,cols=40})

    <br /><br />
    <input type="submit" value="Submit Entry" />
}
```

1 **Display error message summary**

2 **Build input fields using helpers**

In addition to calling the `ValidationSummary` method at the top of the view 1, note that we're also now using MVC's HTML helpers to generate the text inputs on our page 2 (back in chapter 2, we manually wrote the appropriate markup for the `input` and `textarea` elements). One advantage of using these helpers is that MVC will automatically detect the validation error messages (because the elements have the same name as the invalid model properties) and apply a CSS class that can be used to indicate that the field has an error. In this case, because our application is based on the default MVC project template, the invalid fields appear with a light red background, as shown in figure 4.3.

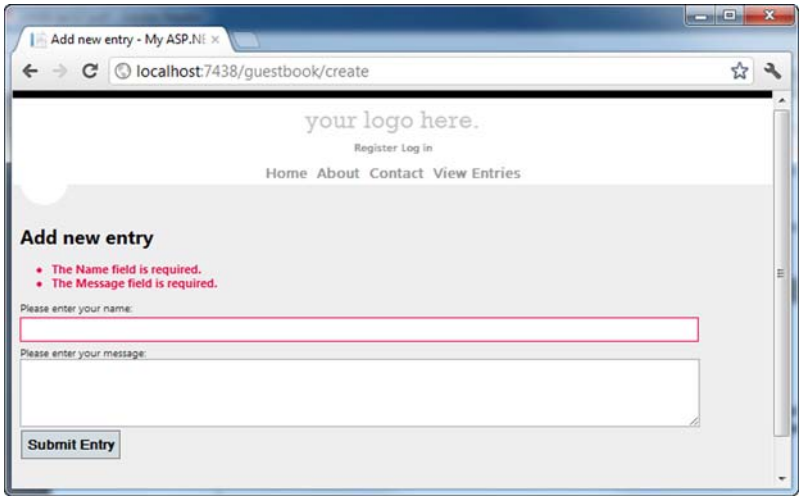


Figure 4.3 Displaying error messages and highlighting invalid fields

The error messages you see in figure 4.2 are ASP.NET MVC's default error messages for required fields. We can override these messages and use our own by modifying the `Required` attribute declaration to include a custom message:

```
[Required(ErrorMessage = "Please enter your name")]
```

Alternatively, if you don't want to hard-code the message and instead want to rely on .NET's support for localization through resource files, you could specify the resource name and resource type:

```
[Required(ErrorMessageResourceType = typeof(MyResources),
     ErrorMessageResourceName = "RequiredNameError")]
```

We've now looked at a couple of common scenarios for controller actions. You've seen that there's often a need to take data from the model and project it into a different shape to render a view. You've also seen that you should validate your input to make sure you don't end up with bad data in your database. But how do you know that your controller actions are working correctly? It can be easy to accidentally introduce bugs, and manually testing every controller action can be a time-consuming process. This is where automated testing comes in. In the next section, we'll talk about one form of automated testing—unit testing—and how you can use this to ensure that your controller actions do what you expect.

4.3 Introduction to unit testing

In this section, we'll take a brief look at testing controllers. Of all the different types of automated testing, we're concerned with only one type at this point: unit testing.

Unit tests are small, scripted tests, usually written in the same language as the production code. They set up and exercise a single component's function in isolation from the rest of the system in order to verify that it's working correctly. As the application grows, the number of unit tests increases too. It's common to see applications with hundreds or even thousands of tests that can be executed at any time to verify that bugs haven't been accidentally introduced into a codebase.

To ensure that unit tests run quickly, it's important that they don't call out of process. When unit testing a controller's code, any dependencies should be simulated so the only production code running is the controller itself. For this to be possible, it's important that controllers be designed in such a way that any external dependencies can be easily swapped out (such as database or web service calls).

In order to effectively test our `GuestbookController`, we'll need to make a few modifications to allow for testability, but before we do this, let's take a look at the default unit testing project that's part of ASP.NET MVC.

4.3.1 Using the provided test project

By default, when you create a new ASP.NET MVC project, Visual Studio provides an option for creating a unit test project (which you saw briefly in chapter 2 and is shown in figure 4.4).

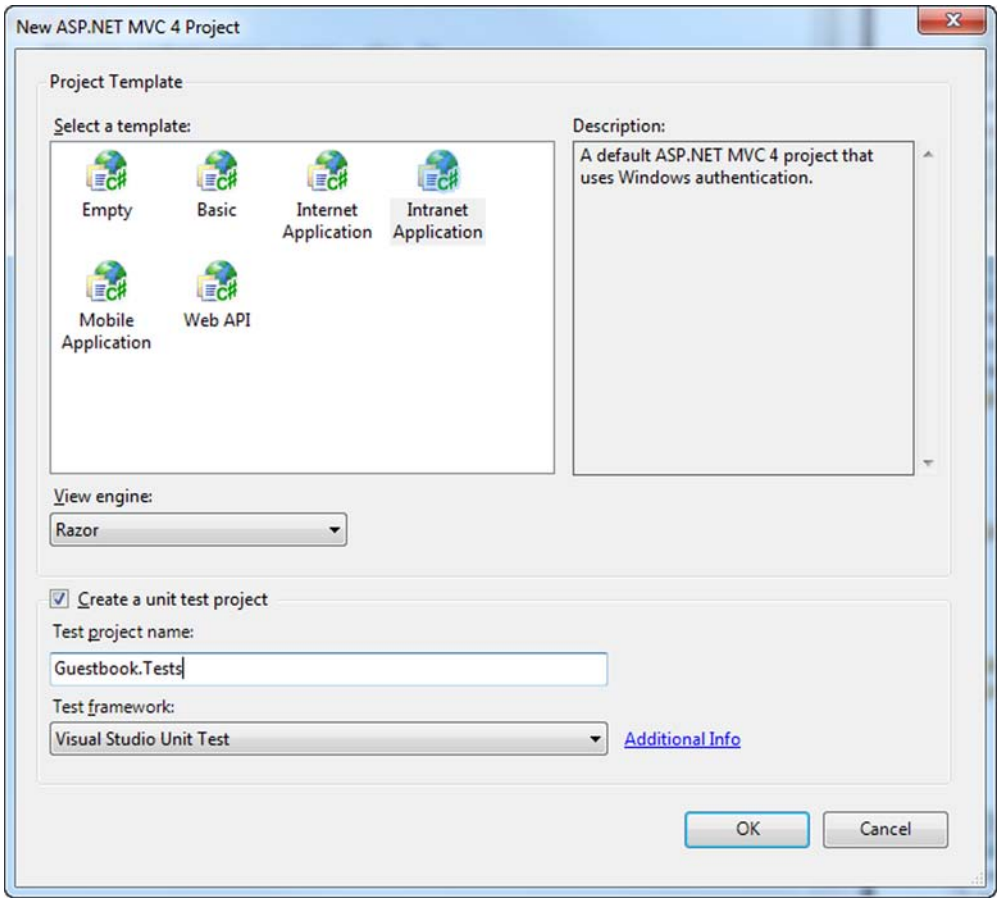


Figure 4.4 Optionally creating a unit test project

If you opt in to creating the unit test project, Visual Studio generates one using the Visual Studio Unit Testing Framework. The unit test project contains a couple of sample tests that can be found in the `HomeControllerTest` class, as shown in listing 4.8.

> **NOTE** Although the unit test project uses the Visual Studio Unit Testing Framework (MSTest) by default, it's possible to extend this dialog box to use other unit testing frameworks, such as NUnit, MbUnit or xUnit.net. In practice, using NuGet to add other test frameworks is simpler than extending this dialog.

Listing 4.8 Default sample tests for the `HomeController`

```
[TestClass]
public class HomeControllerTest
{
  [TestMethod]
```

```
  public void Index()
  {
    // Arrange
    HomeController controller = new HomeController();

    // Act
    ViewResult result = controller.Index()
       as ViewResult;

    // Assert
    Assert.AreEqual("Modify this template to jump-start",
       result.ViewBag.Message);
  }

  [TestMethod]
  public void About()
  {
    // Arrange
    HomeController controller = new HomeController();

    // Act
    ViewResult result = controller.About()
        as ViewResult;

    // Assert
    Assert.IsNotNull(result);
  }
}
```

1 Instantiate controller

2 Exercise action method

3 Assert results

These default tests exercise the two action methods available on the default `HomeController` class created with new MVC projects.

Each test has three phases—*arrange*, *act*, and *assert*. The first test instantiates the `HomeController` 1 (this is the "arrange"), invokes its `Index` method (the "act") to retrieve a `ViewResult` instance 2, and then asserts that the action passed the correct message into the `ViewBag` by calling the static `Assert.AreEqual` method to compare the message in the `ViewBag` with the expected message 3. The test for the `About` action is even simpler as it simply checks that a `ViewResult` was returned from the action.

If we run these tests using Visual Studio's built-in unit test runner, you'll see that both pass, as shown in figure 4.5.

However, these tests aren't particularly good examples of how to write unit tests for your controllers, because the default `HomeController` doesn't contain any real interaction logic. Let's instead look at how we could write some tests for a couple of the actions in our `GuestbookController`.

4.3.2 Testing the GuestbookController

One of the issues with the current implementation of the `GuestbookController` is that it directly instantiates and uses the `GuestbookContext` object, which in turn accesses the database. This means that it isn't possible to test the controller without also having a database set up and correctly populated with test data, which is an integration test rather than a unit test.

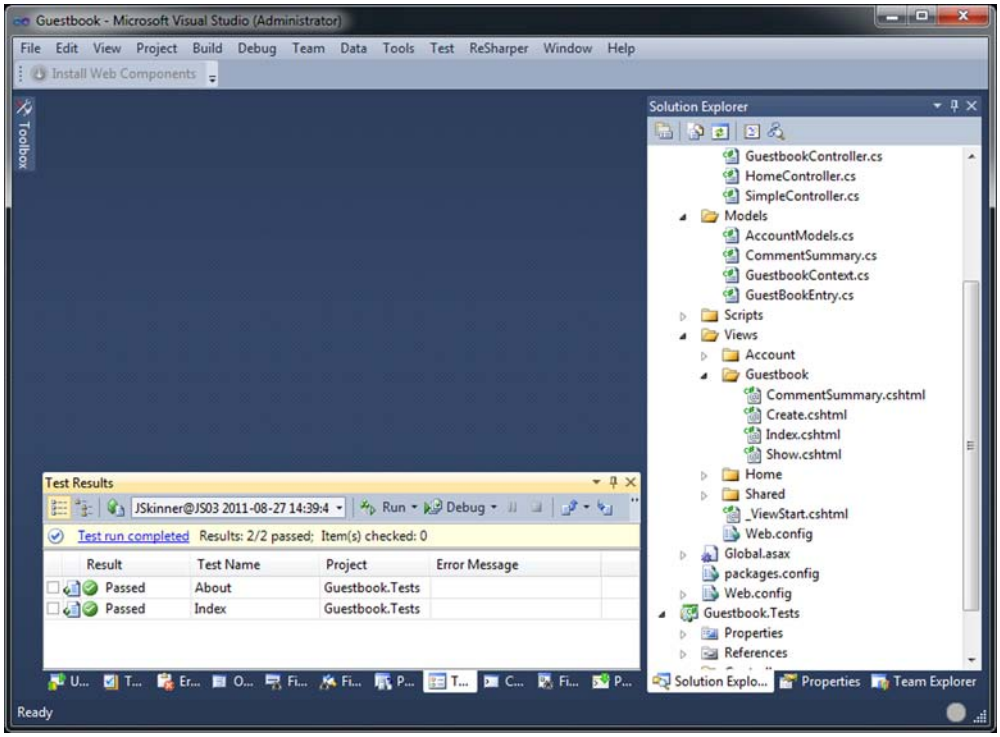


Figure 4.5 Running MSTest unit tests inside Visual Studio

Although integration testing is very important to ensure that the different components of an application are interacting with each other correctly, it also means that if we're only interested in testing the logic within the controller, we have to have the overhead of making database connections for every test. For a small number of tests this might be acceptable, but if you have hundreds or thousands of tests in a project, it will significantly slow down the execution time if each one has to connect to a database. The solution to this is to decouple the controller from the `GuestbookContext`.

Instead of accessing the `GuestbookContext` directly, we could introduce a *repository* that provides a gateway for performing data-access operations on our `GuestbookEntry` objects. We'll begin by creating an interface for our repository:

```
public interface IGuestbookRepository
{
    IList<GuestbookEntry> GetMostRecentEntries();
    GuestbookEntry FindById(int id);
    IList<CommentSummary> GetCommentSummary();
    void AddEntry(GuestbookEntry entry);
}
```

This interface defines four methods that correspond to the four queries that we currently have in our `GuestbookController`. We can now create a concrete implementation of this interface that contains the query logic:

Listing 4.9 The `GuestbookRepository`

```
public class GuestbookRepository : IGuestbookRepository
{
    private GuestbookContext _db = new GuestbookContext();

    public IList<GuestbookEntry> GetMostRecentEntries()
    {
      return  (from entry in _db.Entries
          orderby entry.DateAdded descending
          select entry).Take(20).ToList();
    }

    public void AddEntry(GuestbookEntry entry)
    {
        entry.DateAdded = DateTime.Now;

        _db.Entries.Add(entry);
        _db.SaveChanges();

    }

    public GuestbookEntry FindById(int id)
    {
        var entry = _db.Entries.Find(id);
        return entry;
    }

    public IList<CommentSummary> GetCommentSummary()
    {
        var entries = from entry in _db.Entries
                      group entry by entry.Name into groupedByName
                      orderby groupedByName.Count() descending
                      select new CommentSummary
                      {
                          NumberOfComments = groupedByName.Count(),
                          UserName = groupedByName.Key
                      };
        return entries.ToList();
    }
}
```

1 Implements the interface

The concrete `GuestbookRepository` class implements our new interface by providing implementations of all of its methods. We're using the same query logic that we'd previously placed in the controller, but we've now encapsulated our queries in one place. The controller itself can now be modified to use the repository rather than the `GuestbookContext` directly.

Listing 4.10 Using the repository in the `GuestbookController`

```
public class GuestbookController : Controller
{
    private IGuestbookRepository _repository;

    public GuestbookController()
    {
        _repository = new GuestbookRepository();
```

1 Stores repository in field

2 Creates default repository

```
    }

    public GuestbookController(
       IGuestbookRepository repository)
    {
        _repository = repository;
    }

    public ActionResult Index()
    {
        var mostRecentEntries = _repository.GetMostRecentEntries();
        return View(mostRecentEntries);
    }

    public ActionResult Create()
    {
        return View();
    }

    [HttpPost]
    public ActionResult Create(GuestbookEntry entry)
    {
        if (ModelState.IsValid)
        {
           _repository.AddEntry(entry);
            return RedirectToAction("Index");
        }

        return View(entry);
    }

    public ViewResult Show(int id)
    {
        var entry = _repository.FindById(id);

        bool hasPermission = User.Identity.Name == entry.Name;

        ViewBag.HasPermission = hasPermission;

        return View(entry);
    }

    public ActionResult CommentSummary()
    {
        var entries = _repository.GetCommentSummary();
        return View(entries);
    }
}
```

(3) **Allows repository to be injected**

Rather than instantiating the `GuestbookContext`, we now store an instance of our repository within a field (1). The controller's default constructor (which will be invoked by the MVC framework when we run the application) populates the field with the default implementation of the repository (2). We also have a second constructor (3), which allows us to provide our own instance of the repository rather than the default. This is what we'll use in our unit tests to pass in a fake implementation of the repository. Finally, the actions in our controller now use the repository to perform data access rather than executing LINQ queries directly.

NOTE Although we've moved the querying logic out of the controller, it's still important that the query itself should be tested. However, this would not be part of a unit test but rather an integration test that exercises the concrete repository instance against a real database.

Dependency injection

The technique of passing dependencies into the constructor of an object is known as dependency injection. However, we've been performing the dependency injection manually by including multiple constructors in our class. In chapter 18, we'll look at how we can use a dependency injection container to avoid the need for multiple constructors. More information about dependency injection can also be found in the book Dependency Injection in .NET by Mark Seemann (http://manning.com/seemann/) as well as in numerous online articles, such as "Inversion of Control Containers and the Dependency Injection Pattern" by Martin Fowler (http://martinfowler.com/articles/injection.html).

At this point, we're able to test our controller actions in isolation from the database, but to achieve this we'll need a fake implementation of our `IGuestbookRepository` interface that doesn't interact with the database. There are several ways to achieve this— we could create a new class that implements this interface but performs all operations against an in-memory collection (shown in listing 4.11), or we could use a *mocking framework* such as moq or Rhino Mocks (both of which can be installed via NuGet) to automatically create the fake implementations of our interface for us.

Listing 4.11 A fake implementation of `IGuestbookRepository`

```
public class FakeGuestbookRepository : IGuestbookRepository
{
    private List<GuestbookEntry> _entries
       = new List<GuestbookEntry>();

    public IList<GuestbookEntry> GetMostRecentEntries()
    {
        return new List<GuestbookEntry>
        {
            new GuestbookEntry
            {
                DateAdded = new DateTime(2011, 6, 1),
                Id = 1,
                Message = "Test message",
                Name = "Jeremy"
            }
        };
    }

    public void AddEntry(GuestbookEntry entry)
    {
        _entries.Add(entry);
    }
```

1 List used for storage

```
    public GuestbookEntry FindById(int id)
    {
        return _entries.SingleOrDefault(x => x.Id == id);
    }

    public IList<CommentSummary> GetCommentSummary()
    {
        return new List<CommentSummary>
        {
            new CommentSummary
            {
                UserName = "Jeremy", NumberOfComments = 1
            }
        };
    }
}
```

The fake implementation of our repository exposes the same methods as the real version, except internally it simply makes use of an in-memory collection (1) and both the `GetCommentSummary` and `GetMostRecentEntries` methods return canned responses (they always return the same fake data).

As our controller contains several actions, there are potentially quite a few tests that we could write. The following listing shows a couple of tests for the `Index` action:

Listing 4.12 Testing the `Index` action

```
[TestMethod]
public void Index_RendersView()
{
    var controller = new GuestbookController(
       new FakeGuestbookRepository());
    var result = controller.Index() as ViewResult;
    Assert.IsNotNull(result);
}

[TestMethod]
public void Index_gets_most_recent_entries()
{
    var controller = new GuestbookController(
       new FakeGuestbookRepository());
    var result = (ViewResult)controller.Index();
    var guestbookEntries = (IList<GuestbookEntry>) result.Model;
    Assert.AreEqual(1, guestbookEntries.Count);

}
```

(1) Pass fake repository to controller

The first of our tests invokes the `Index` action and simply asserts that it renders a view (much like the tests for the `HomeController`). The second test is slightly more complex—it asserts that a list of `GuestbookEntry` objects was passed to the view (if you remember, the `Index` action invokes the `GetMostRecentEntries` method of our repository).

Both tests make use of the fake repository (1). By passing it to the controller's constructor, we ensure that the controller uses our fake set of in-memory data rather than connecting to the real database.

Unit testing vs. TDD

The examples in this section have followed a fairly traditional unit testing approach, where the tests have been written after the code in order to validate its behavior. If we were using TDD (test-driven development), both the tests and the code would be written in small iterations: first write a failing test, then the code to make it pass. This usually means that much less time is spent debugging code, because it leads to a workflow in which you are constantly creating small working chunks.

In this section, you saw that you can use unit testing to verify that your controller actions are doing what you expect them to. We wrote some tests to verify that a couple of the actions in the `GuestbookController` did what we expected, but we also saw that we had to make some changes to the controller in order for it to be easily unit-testable. If you design your applications with testability in mind, this will avoid the need to perform subsequent refactorings for testability.

4.4 Summary

In this chapter, we looked in more detail at controllers in the context of our example Guestbook application. You saw that there are several ways to indicate that a class is a controller, although most of the time you'll inherit from the `Controller` base class. You also saw that controller actions don't have to return views—there are many other types of `ActionResults` available, and you can even render content directly from an action. From this you can see that controller actions aren't limited to just rendering views and that you can customize your controller actions to return the type of content that you need for a particular scenario. You can even create your own custom action results if you need to send a response from a controller action that the framework doesn't support by default (we'll look at this in chapter 16).

Following this, we looked at some operations that would typically be part of a controller action, such as mapping view models and validation. Both of these are common scenarios that you'll typically end up doing very often in your applications, so it's important to understand how to do them. We'll dig into both of these topics in more detail later—we'll cover many options available for validation in chapter 6, and mapping view models is the subject of the next chapter.

Finally, we looked at the default unit testing project and at how you can perform assertions on the results of controller actions to make sure a controller action is working correctly.

We've now finished the introductory part of the book—in the next part, we'll move away from the Guestbook application that we've used so far and begin to focus on more advanced topics related to ASP.NET MVC development. We'll begin by exploring the topic of view models, which we mentioned briefly in this chapter, in more detail.

ASP.NET/WEB DEVELOPMENT

ASP.NET MVC 4 IN ACTION

Palermo • Bogard • Hexter • Hinze • Skinner

Free eBook
SEE INSERT

ASP.NET MVC provides the architecture needed to separate an application's logic and its UI. Because each component's role is well defined, MVC applications are easy to test, maintain, and extend. The latest version, ASP.NET MVC 4, takes advantage of .NET 4 and includes powerful features like the Razor view engine, Web Matrix helpers, and enhanced extensibility.

ASP.NET MVC 4 in Action is a hands-on guide that shows you how to apply ASP.NET MVC effectively. After a high-speed ramp up, this thoroughly revised new edition explores each key topic with a self-contained example so you can jump right to the parts you need. Based on thousands of hours of real-world experience, the authors show you valuable high-end techniques you won't find anywhere else. Written for developers, the book arms you with the next-level skills and practical guidance to create compelling web applications.

What's Inside

- Complete coverage of ASP.NET MVC 4
- The new Web API
- Full-system testing

You need some knowledge of ASP.NET and C#, but no prior ASP.NET MVC experience is assumed.

Jeffrey Palermo, Jimmy Bogard, Eric Hexter, Matthew Hinze, and **Jeremy Skinner** are all ASP.NET MVPs, ASP Insiders, and early adopters of ASP.NET MVC.

To download their free eBook in PDF, ePub and Kindle formats, owners of this book should visit manning.com/ASP.NETMVC4inAction

"Guides you through the inner workings of ASP.NET MVC."
—From the Foreword by Phil Haack, GitHub

"A brilliant book for a great framework."
—Jonas Bandi, TechTalk

"A complete guide, with established open source tools."
—Apostolos Mavroudakis, UBS AG

"A great addition to a great series of books."
—Paul Stack, Toptable.com

"Practical web application construction for the pragmatic practitioner."
—Arun Noronha
Guardian Protection Services

ISBN 13: 978-1-617290-41-1
ISBN 10: 1-617290-41-6
54999
9 781617 290411

MANNING $49.99 / Can $52.99 [INCLUDING eBOOK]