# CHAPTER 9: TESTING ASP.NET MVC APPLICATIONS

The ASP.NET MVC framework has been developed to be testable out of the box. After having learned in the previous chapter what unit testing is and how to test applications, in this chapter you apply these concepts together with the ASP.NET MVC–specific features made to facilitate testing.

In the chapter, you learn:

- How to create a new testing project for an ASP.NET MVC application
- How to test controllers' actions and routes
- How the ASP.NET MVC framework facilitates testing
- How to test an end-to-end application
- A bit of TDD approach

## Creating a Test Project for ASP.NET MVC

Every ASP.NET MVC application should have a companion test project, so the project template for this process is super. When you create an ASP.NET MVC Web application, as soon as you click the OK button, another dialog pops up (see Figure 9-1), asking you whether you want to create a unit test project for the application. In this dialog, you can choose the name of the test project; by default, it's named like the MVC project name dot Tests (for example `mvcapplication.Tests`) and the testing framework you want to use to create the tests with. With the installation, there is only the Visual Studio Unit Test template, which uses MSTest, but you can add other testing frameworks if you like. (In Figure 9-1 MbUnit v3 is shown.)

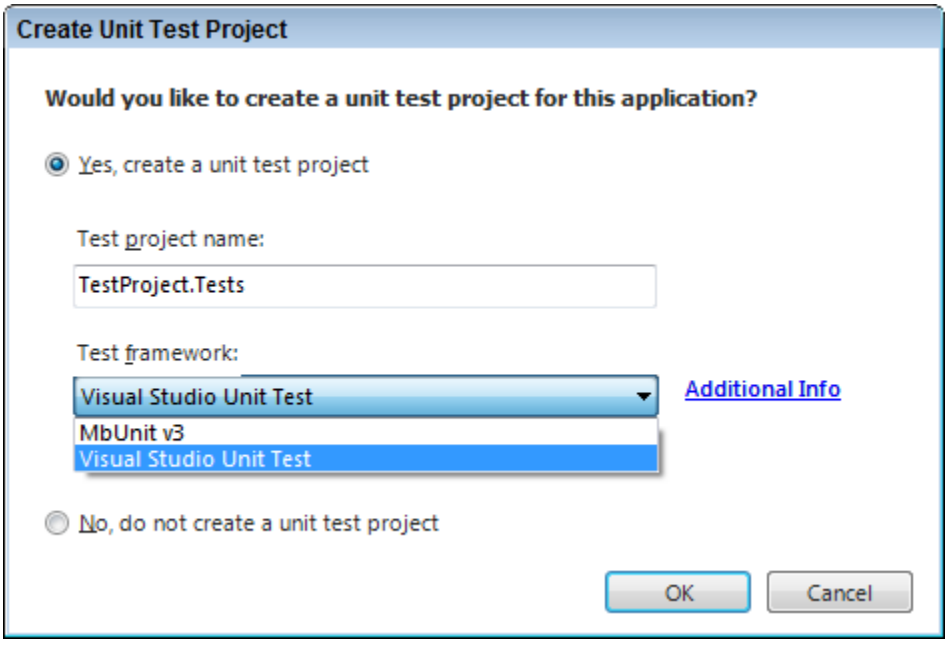


**Figure 9-1**

If you choose Yes, a new project will be created, with a tiny test class that tests the controller created by the project template: `HomeControllerTest`.

In the previous chapter, you read about mock objects and dependency injection, but to test the actions of a controller you often don't need to bring these concepts to the table. The ASP.NET MVC team did a very good job at removing many of the needless dependencies on the environment so that you can test actions just by instantiating the controller and calling the method. And this is true for most of the common scenarios.

In the next section, you'll read about these easily testable common scenarios and later, in the following ones, you'll see how the environment is abstracted so that you can also test the scenarios that interact directly with the HTTP runtime.

# Testing without Mocking

As you might remember from Chapter 8, a test is performed in four phases: Setup, Exercise, Verification, and Teardown. In that chapter you also read that mocking frameworks are starting to adopt a new syntax called Arrange, Act, Assert (AAA).

Given the rising popularity of the AAA syntax for mocking frameworks, and to make the symmetry even stronger, the Visual Studio project template creates a sample testing class and refers to the phases of unit testing using the AAA naming. (The Teardown phase is not included as this phase is usually in the `TearDown` method of the unit test class.)

But unlike mocking frameworks, in the context of test phases, the AAA naming doesn't bring in a different way of writing tests, but just different names for the various phases of unit tests. The AAA naming for test phases equates to the "traditional" test phases in the following way:

- Arrange = Setup
- Act = Exercise
- Assert = Verify

Since it's just a name shift, and since the Visual Studio project templates is using it, throughout the rest of this chapter, the new AAA naming will be used.

Following is the test method as created by the Visual Studio project template:

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

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

    // Assert
    // Here goes the verify phase
}
```

When testing an action, the first two phases are pretty simple: the setup (arrange) is just the instantiation of the controller, and the exercise (act) phase is just calling the action you want to test. The teardown phase,

most of the time, is implicitly performed by the garbage collector, so the only phase that needs your attention is the verification (or assert) one, where the test looks inside the result of the action to prove the correctness of the method.

The Assert phase is different, depending on what you want to test. You will learn about the different ways to do testing. You can split the outcomes of the controller actions in two main groups:

- The actions that render a view or other outputs
- The actions that send the request flow to another action, via a redirect

These two types of actions need different approaches.

# Testing Actions That Render Something

The first, and probably most used, type of actions sends some kind of output to the browser. This can be a view, a partial view, a direct text response, a JSON response, or other custom response types.

So, the first thing you have to verify is that the type returned by the action is the one you are expecting. Call the action method, which usually returns an object that inherits from `ActionResult`, and cast it to the specific type of result you are expecting, using the `as` statement.

The `as` statement behaves differently from the usual cast operation. If the cast cannot be completed, instead of raising an `InvalidCastException` it assigns a null value to the variable.

```
// Act
ActionResult actionResult = controller.Index();
// Assert

ViewResult result = actionResult as ViewResult;
Assert.IsNotNull(result,"The result is not a view result");
```

After you determine that the action is returning the correct kind of result, you have to verify the correctness of the data being passed to the view.

`ViewData` is a property of the result object, so verifying its content is pretty easy:

```
ViewDataDictionary viewData = result.ViewData;

Assert.AreEqual("Home Page", viewData["Title"]);
```

If the view data also contains a custom presentation model supplied to the view in a strongly typed manner, you can access it through the `Model` property.

```
MyCustomObject myObject = viewData.Model as MyCustomObject;

Assert.IsNotNull(myObject, "The custom object has not been passed");
Assert.AreEqual("Simone", myObject.Username);
```

This approach is similar to the one used to test the type of the response. First, you cast the model object to the expected type, then verify that it's not null, and finally check the value of the properties that need to be tested. This is pretty much all you have to do to verify that the data retrieved and manipulated inside the actions is the data you expected.

In case your action has logic that chooses the view to render based on certain conditions, you also have to test that the view that is going to be rendered is the correct one. To facilitate this scenario, the `ViewResult` and the `PartialViewResult` objects contain properties that you can access to check for these conditions:

- `ViewName`: The exact name of the view that is going to be rendered
- `MasterName`: The name of the master page

With this in place, if you want to test that the view that is going to be rendered is the view named "Index," you just need to write:

```
Assert.AreEqual("Index",result.ViewName);
```

The process is the same if you want to test the master page name (only if it has been set explicitly in the action method):

```
Assert.AreEqual("AdminMaster",result.MasterName);
```

In the following Try It Out, you test an ASP.NET MVC application that renders views.

type="activity"

## Try It Out Testing an Action That Renders a View

In this Try It Out section, you are going to write an application that renders different views based on the time of the day. The sample contains only the code of the controller and the tests. This is to show that you can start your application even before having created a view.

1. Create a new ASP.NET MVC Web application and create a testing project, selecting the Yes, create a unit test project option when asked.

2. Create a new file in the `Model` folder, name it **Time.cs**, and add the following code:

```
namespace TestingRenderView.Models
{
    public class Time
    {
        public int Hour { get; set; }
    }
}
```

3. Open the Controllers\HomeController.cs file, and add the following method:

```
public ActionResult DayPart(int hour)
{
    Time time = new Time {Hour = hour};
    if (hour <= 6 || hour >= 20)
        return View("NightTimeView", time);
    if (hour > 6 && hour < 20)
        return View("DayTimeView", time);
    return View("Index");
}
```

4. Now focus your attention on the other project, the testing one. Open the HomeControllerTest.cs file and write a new test method:

```
[TestMethod]
public void DayTimeView_Is_Rendered_At_Midday()
{

}
```

5. Inside this method start writing the setup of the test:

```
HomeController controller = new HomeController();
```

6. Then exercise the system under test:

```
ActionResult actionResult = controller.DayPart(12);
```

7. Finally, add the code to verify all the conditions. After adding the code the method will be:

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

    // Act
    ActionResult actionResult = controller.DayPart(12);

    // Assert
    ViewResult result = actionResult as ViewResult;
    Assert.IsNotNull(result,"Not a RenderView Result");
    Assert.AreEqual("DayTimeView", result.ViewName);
    ViewDataDictionary viewData = result.ViewData;
    Time model = viewData.Model as Time;
    Assert.IsNotNull(model,"Model is not the correct type");
    Assert.AreEqual(12, model.Hour);
```

```
}
```

8. Then add another test method, one that verifies the other possible outcome of the action:

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

    // Act
    ActionResult actionResult = controller.DayPart(0);

    // Assert
    ViewResult result = actionResult as ViewResult;
    Assert.IsNotNull(result, "Not a RenderView Result");
    Assert.AreEqual("NightTimeView", result.ViewName);
    ViewDataDictionary viewData = result.ViewData;
    Time model = viewData.Model as Time;
    Assert.IsNotNull(model, "Model is not the correct type");
    Assert.AreEqual(0, model.Hour);
}
```

Comp: Note Search and Replace codes below.

9. Now run the tests (command menu Test@@>Run@@>All Tests in Solution) and see them "*green*".

## HOW IT WORKS

In the first part of the example, a normal action has been created. Based on the hour supplied as parameter (in a real production environment this would have been passed as part of the URL and mapped through the Routing Engine), it renders a different view and passes it a custom model object (the `Time` object). To make sure that all the possible outcomes are tested, you need to write two tests: one with a parameter in the day range and the other in the night range. The first is called `DayTimeView_Is_Rendered_At_Midday` and the second one is `NightTimeView_Is_Rendered_At_Midnight`. They are the same method but have different parameters and expected values, so the explanation of the code will be based only on the day time one.

At Step 4 the test method is created, annotating the method with the `TestMethod` attribute, as required by the MSTest framework. Then an instance of the controller is created, and finally the SUT is exercised by calling the action method `DayPart` with the parameter 12 (midday).

But all the complexity lies in what you add during Step 7: all the assertions.

The action result is first cast to the expected result type (`ViewResult`), and you verify that the returned object is really as expected:

```
ViewResult result = actionResult as ViewResult;
Assert.IsNotNull(result,"Not a RenderView Result");
```

Then the name of the view is verified:

```
Assert.AreEqual("DayTimeView", result.ViewName);
```

And, finally, the test verifies that a custom model object of the correct type was returned and that the Hour is really set to the one passed as the parameter of the action method (in this sample, 12):

```
ViewDataDictionary viewData = result.ViewData;
Time model = viewData.Model as Time;
Assert.IsNotNull(model,"Model is not the correct type");
Assert.AreEqual(12, model.Hour);
```

The other test method is the same, just with `0` passed as the value of the parameter of the action and, obviously, with a different expected `ViewName` (`NightTimeView`).

It was not implemented in this sample, but usually it's a good idea to test the boundary conditions, to make sure that the inclusion of the limit has been implemented correctly. In the sample, the night goes from 8PM to 6AM, limits included, so you have also written two tests named something like `NightTimeView_Is_Still_Rendered_At_6AM` and `NightTimeView_Is_Already_Rendered_At_8PM`. The first passed 6 as value of the action and the second, 20.

These boundary tests are included in the code samples that come with the book, together with a full implementation of the Web application, with the two different views and the routing rules.

Place HIW end rule here

# Testing Actions That Redirect to Another Action

The other group of actions contains the ones that, instead of rendering something, redirect the flow of the application to another action or URL.

The test flow is the same as the previous one, although the details are slightly different: The type of the action result must be either `RedirectResult` or `RedirectToRouteResult`.

```
RedirectResult result = actionResult as RedirectResult;
Assert.IsNotNull(result, "The result is not a redirect");
```

Then, you have to verify that the processing is being redirected to the right URL.

If the result is a `RedirectResult`, this is pretty easy: Just check for the `Url` property.

```
Assert.AreEqual("/Home/NotLogged", result.Url);
```

If the action is returning a `RedirectToRouteResult`, things are a bit more complex. In this case you can check either the `RouteName` or the `RouteValues` collection, which is an instance of the usual `RouteValueDictionary` class. For example, if you want to verify that the flow is redirected to the action named "`Login`", this is the code to write:

```
Assert.AreEqual("Login",result.RouteValues["action"]);
```

Once you have verified that the request will be redirected to the correct action, the other element that needs to be tested is the `TempData` collection. This collection is not part of the result but is a property of the controller.

```
Assert.AreEqual("Simone", controller.TempData["Name"]);
```

Now that you know about testing controller actions, in the next Try It Out section, which builds on the previous one, you put everything together inside the same application.

type="activity"

## Try It Out: Testing for both Render and Redirect

In the previous Try It Out, the code didn't handle very well the case when the user specifies a value that is not a valid hour of the day, for example 44. In this example, you enhance the code of the application so that it redirects the user to a specific view if the value of the hour is not valid.

1. Building on the previous Try It Out sample, in the `Controllers\HomeControllers.cs` file replace the `DayPart` action with the following one:

```
public ActionResult DayPart(int hour)
{
    if(hour <0 || hour > 24)
    {
        TempData["Hour"] = hour;
        return RedirectToAction("NotValid");
    }
    Time time = new Time { Hour = hour };
    if (hour <= 6 || hour >= 20)
        return View("NightTimeView", time);
    return View("DayTimeView", time);
}
```

2. Then, add to the same controller the action that handles the error situation:

```
public ActionResult NotValid()
{
    ViewData["UserSuppliedTime"] = TempData["Hour"];
    return View("NotAValidHour");
}
```

3. Finally, add to the `HomeControllerTest` class the test method for the new “not valid” scenario:

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

    // Act
    ActionResult actionResult = controller.DayPart(44);

    // Assert
    RedirectToRouteResult result = actionResult as RedirectToRouteResult;
    Assert.IsNotNull(result, "Not a RedirectToRouteResult Result");
    Assert.AreEqual("NotValid", result.RouteValues["Action"]);
    Assert.AreEqual(44, controller.TempData["Hour"]);
}
```

4. The complete test class should now look like Listing 9-1:

***Listing 9-1: The complete test class***

```
using System.Web.Mvc;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using RenderAndRedirect.Controllers;
using RenderAndRedirect.Models;

namespace RenderAndRedirect.Tests.Controllers
{
    /// <summary>
    /// Summary description for HomeControllerTest
    /// </summary>
    [TestClass]
    public class HomeControllerTest
    {
```

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

    // Act
    ActionResult actionResult = controller.DayPart(12);

    // Assert
    ViewResult result = actionResult as ViewResult;
    Assert.IsNotNull(result, "Not a RenderView Result");

    Assert.AreEqual("DayTimeView", result.ViewName);
    ViewDataDictionary viewData = result.ViewData;
    Time model = viewData.Model as Time;
    Assert.IsNotNull(model, "Model is not the correct type");
    Assert.AreEqual(12, model.Hour);
}

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

    // Act
    ActionResult actionResult = controller.DayPart(0);

    // Assert
    ViewResult result = actionResult as ViewResult;
    Assert.IsNotNull(result, "Not a RenderView Result");
    ViewDataDictionary viewData = result.ViewData;
    Assert.AreEqual("NightTimeView", result.ViewName);
    Time model = viewData.Model as Time;
    Assert.IsNotNull(model, "Model is not the correct type");
    Assert.AreEqual(0, model.Hour);
}

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

    // Act
    ActionResult actionResult = controller.DayPart(44);

    // Assert
    RedirectToRouteResult result = actionResult
            as RedirectToRouteResult;
```

```
            Assert.IsNotNull(result, "Not a RedirectToRouteResult
Result");
            Assert.AreEqual("NotValid", result.RouteValues["Action"]);
            Assert.AreEqual(44, controller.TempData["Hour"]);
        }
    }
}
```

5. Now run the tests, including the one you created in the previous Try It Out to confirm that they all pass.

## How It Works

In the previous Try It Out, the action didn't handle invalid hours very well. In this new version of the action, the code first checks whether the hour supplied is positive and less than 24. If it's not, the code redirects the user to another action, called `NotValid`, and passes the original value to it. This is done using the `TempData` collection.

Notice that the implementation of the part of the day has been changed a bit, but the tests you put in place will tell you whether you broke something, so you can do your refactoring free from worry.

The action that handles the error condition just takes the original value from the `TempData` collection and stores it inside the `ViewData` collection and then renders the view with the error message.

The new test method, called `Not_Valid_Hours_Are_Redirected_To_NotValid_Action`, calls the action to test with an invalid parameter (`44` in this sample) and then verifies that everything is as expected.

As before, first the type of the action result is tested for correctness:

```
RedirectToRouteResult result = actionResult as RedirectToRouteResult;
Assert.IsNotNull(result, "Not a RedirectToRouteResult Result");
```

Then the code proves the correctness of the action to which the flow will be redirected:

```
Assert.AreEqual("NotValid", result.RouteValues["Action"]);
```

And, finally, the value inside the `TempData` is tested:

```
Assert.AreEqual(44, controller.TempData["Hour"]);
```

Place HIW end rule here

type="warning"

> Most of the parameters inside the action results, such as the `ViewName` or the name of the controller inside the `RouteValueDictionary`, are available to use inside a test method only if they are explicitly set by the developer. If you call the `View()` method without parameters, the `ViewName` will be empty. So remember this when testing actions that rely on the default values for the view name: Either test it to see if it is empty or explicitly set the name of the view.

# System.Web.Abstrations

In the previous section, you tested the controllers' actions as if they were a general-purpose library that doesn't have anything to do with the Web and HTTP. A big round of applause to the ASP.NET MVC team for coming up with such an easy-to-test framework.

But these are Web applications, and unfortunately at some time you might have to interact with the HTTP runtime to get, for example, a value from the browser's cookies or you might need to interact with `Session` object. Even in these scenarios, however, you'd want to go on testing the actions without setting up the Web server environment. That's what the assembly `System.Web.Abstractions` is for. All the methods in the ASP.NET framework that interact with the HTTP runtime are actually interacting with a wrapper that, in the end, calls the real `System.Web` assembly so that you don't need the Web server runtime to test these interactions. When writing tests, you can replace the implementation of the wrapper with a stub/mock object made on purpose for your test.

This namespace, like the routing one, was developed with ASP.NET MVC in mind, but it was later considered of wider interest and was shipped as part of the SP1 of .NET 3.5 at the end of August 2008.

In the next section, you learn how to mock the objects of the HTTP runtime in order to test HTTP-specific interactions, such as cookies.

# Testing with Mocking

In the previous chapter, you learned about mock objects and Rhino Mocks. Now you will use Rhino Mocks to test the interaction of actions with the `http` objects.

Imagine that you want to read the value of a cookie. To do this, you have to access the `HttpRequest` object and read the `Cookies` property. The code for this action is:

```
public ActionResult About()
{
    string lastLogin = Request.Cookies["LastLogin"].Value;
    ViewData["LastLogin"] = lastLogin;

    return View();
}
```

Now, you want to test that the action actually reads the data from the browser cookie and then passes it over to the view.

In ASP.NET MVC, all the references to the HTTP runtime are made through the abstraction layer provided by the `System.Web.Abstractions` assembly. This allows you to inject your own implementation or a mock object instead of using the real implementation.

The `Request` object is stored inside the `HttpContext` property of the `ControllerContext` object, which, in a production environment, is created by the framework when it handles the request. But during unit testing there is no framework that sets everything up for you, so you have to do it yourself.

As you might have guessed, setting up all the dependencies is not easy. You first need to mock the context, then mock the request, set the expected result, and finally create the cookie you expect to find in the user's browser when the application runs in production.

To keep things neat, it's usually better to encapsulate the mock's setup inside a method marked with the `TestInitialize` attribute that will run before each test method:

```
private MockRepository mocks;
private HttpContextBase mockHttpContext;
```

```
[TestInitialize]
public void SetupHttpContext()
{
    mocks = new MockRepository();
    //Mock setup
    mockHttpContext = mocks.DynamicMock<HttpContextBase>();
    HttpRequestBase mockHttpRequest = mocks.DynamicMock<HttpRequestBase>();
    SetupResult.For(mockHttpContext.Request).Return(mockHttpRequest);

    HttpCookieCollection cookies = new HttpCookieCollection();
    cookies.Add(new HttpCookie("LastLogin","Yesterday"));

    Expect.Call(mockHttpRequest.Cookies).Return(cookies);

    mocks.ReplayAll();
}
```

The first line of the method creates an instance of the mock repository, as you already saw in the previous chapter. Next, using the `DynamicMock` method, a mock object of both the context and the request is created. The context mock object is then instructed to return the mocked request object when the `Request` property is called:

```
SetupResult.For(mockHttpContext.Request).Return(mockHttpRequest);
```

And finally the cookie with the test case value is created and the expectation is set on the request object:

```
Expect.Call(mockHttpRequest.Cookies).Return(cookies);
```

Notice the difference between the two instructions: While the first one—the one that uses the `SetupResult.For` syntax—tells the only mocked object that whenever someone calls the `Request` property, it must return the given object. The one that uses `Expect.Call` does a bit more. It still sets the return value, but it also instructs the mock framework to verify that this method is actually called during the execution of the test.

Now that all the mock objects are ready, the `ReplyAll` method ends the record phase and puts the mock objects in replay mode.

This test method is not much different from the ones that run without mocking. The controller is instantiated, the action is executed, and the results are verified.

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

    controller.ControllerContext = new ControllerContext(
        mockHttpContext,
        new RouteData(),
        controller);
```

```
    // Act
    ActionResult actionResult = controller.About();

    // Assert
    ViewResult result = actionResult as ViewResult;
    Assert.IsNotNull(result, "Not a RenderView Result");
    ViewDataDictionary viewData = result.ViewData;
    Assert.AreEqual("Yesterday", viewData["LastLogin"]);

    mocks.VerifyAll();

}
```

The only differences are the lines highlighted in grey.

The first one is the highlighted section that does the same thing that the framework does when it handles the request in a live application: it injects the `ControllerContext` in the controller. But in this case the context is not the real `HttpContext` but the mocked version created in the test initialization method.

The other highlighted line instructs the mock repository to verify all the expectations. If the action method cheated and didn't retrieve the `LastLogin` value from the cookies, this would have made the test fail. To experiment with this behavior, run the test now, and see everything pass. Then change the action method, replacing the line that reads the cookie with one that sets a static value, like this:

```
public ActionResult About()
{
    string lastLogin = "Yesterday";

    ViewData["LastLogin"] = lastLogin;
    return View();
}
```

Now run the test again. It will fail since the cookie's properties have never been accessed.

The same approach can be used to test actions that access the ASP.NET `Cache` object, the `Session` object, or any of the objects that are inside the current HTTP execution context.

# Testing Routes

Now that you have tested all the components of an ASP.NET MVC Web application, there is one last component that you might want to test:

the routes. In Chapter 7 you saw that there is a route handler that helps you debug routing rules, but it's always better to be able to automate the testing.

The routing engine relies on `HttpContext` and `HttpRequest` to get the URL of the current request, so, in order to test routes, you still need to mock these two objects. This is done in the same way as before:

```
MockRepository mocks = new MockRepository();
//Mock setup
HttpContextBase mockHttpContext = mocks.DynamicMock<HttpContextBase>();
HttpRequestBase mockHttpRequest = mocks.DynamicMock<HttpRequestBase>();
SetupResult.For(mockHttpContext.Request).Return(mockHttpRequest);
```

Once the mock HTTP stack is set up, you need to supply the URL that you want to test:

```
SetupResult.For(mockHttpRequest.AppRelativeCurrentExecutionFilePath)
    .Return("~/archive/2008/10/15");
```

This is done by setting the result for the property of the `HttpRequest` that contains the URL relative to the application root path.

The exercise phase consists of calling the helper method that registers the routes in the `global.asax.cs` file. It then acts as part of the routing engine and calls the framework methods that search the collection with all the registered routes and retrieves the one that matches the URL supplied: the `GetRouteData` method.

```
RouteCollection routes = new RouteCollection();
MvcApplication.RegisterRoutes(routes);

RouteData routeData = routes.GetRouteData(mockHttpContext);
```

And, finally, the test needs to verify that the segments have been evaluated correctly and the route data has been populated with the right values:

```
RouteData routeData = routes.GetRouteData(mockHttpContext);
Assert.IsNotNull(routeData, "Should have found the route");

Assert.AreEqual("Blog", routeData.Values["Controller"]
    , "Expected a different controller");

Assert.AreEqual("Index", routeData.Values["action"]
    , "Expected a different action");

Assert.AreEqual("2008", routeData.Values["year"]
    , "Expected a different year parameter");

Assert.AreEqual("10", routeData.Values["month"]
    , "Expected a different month parameter");

Assert.AreEqual("15", routeData.Values["day"]
    , "Expected a different day parameter");
```

# Putting It All Together

Now that you know everything about testing specific elements of an ASP.NET MVC application, it's time to test a real-world application. You will learn how to do this by making the application used as an example in Chapter 4 testable.

## Analyzing a Legacy Application

The code sample you will focus on is the application that retrieves a list of Pink Floyd albums. It used a repository developed with LINQ to SQL. But this is a *legacy* application, which, among the other things, means "not developed to be testable." The controller directly depends on the repository. The complete code of the controller is shown in Listing 4-9, but the parts relevant to show this approach are included here:

```
public class AlbumController : Controller
{
    public ActionResult Index()
    {
        ViewData["Title"] = "List of Pink Floyd Albums";
        ViewData["Albums"] = AlbumsDataContext.GetAlbums();

        return View();
    }

    public ActionResult Individual(int id)
    {
        var album = AlbumsDataContext.GetAlbum(id);

        ViewData["Title"] = album.Name;
        ViewData["Album"] = album;

        return View();
    }
}
```

The lines where all the problems lay are highlighted in grey. The data access layer is using a "hard" reference that makes it impossible to test the controller in isolation.

The other problem is that the methods used to access the repository are static methods. So, even if the controller didn't use the reference to the implementation of the data access layer, it would have been impossible to abstract out this method, since static methods cannot be part of interfaces.

```
namespace LinqSample.Models
{
    public class AlbumsDataContext
    {
        public static List<Album> GetAlbums()
        {
            ...
        }

        public static Album GetAlbum(int id)
        {
            ...
        }
    }
}
```

Last, but not least, the `Album` class was generated by the LINQ designer and is part of the `LinqDataContext`, so it might not be available to the application if the test code doesn't have a reference to the assembly that contains the data access layer.

To verify that the controller is behaving correctly in isolation from the data access layer, you need to refactor the code in order to make it testable.

# Refactoring to Testable

Dependency Injection is the key for this refactoring process, which will be based on the following steps:

1. Define the `Album` model class.
2. Define the `IAlbumContext` interface.
3. Have the concrete `AlbumDataContext` implement it.
4. Change the `AlbumController` to use the DI pattern.

## Step 1 – The Model Object

This step is quite easy: Just copy the same class autogenerated by the LINQ designer, as shown in Listing 9-2.

***Listing 9-2: Album object***

```
namespace TestingDAL.Models
{
    public class Album
```

```
    {
        public int ID { get; set; }
        public string Name { get;  set; }
        public string ReleaseYear { get; set; }
    }
}
```

## Step 2 – The Data Access Layer Interface

To make it possible to swap the implementation of the data access layer without changing and recompiling the application, the methods needed by the application to work must be extracted to an interface that will later be used as the only way controllers and other clients refer to the data access layer (DAL). (See Listing 9-3.)

***Listing 9-3: IAlbumDataContext interface***

```
using System.Collections.Generic;

namespace TestingDAL.Models
{
    public interface IAlbumDataContext
    {
        List<Album> GetAlbums();
        Album GetAlbum(int id);
    }
}
```

## Step 3 – The Concrete DAL Implements the Interface

Since this sample is only about testing, you could have skipped this step, but if you want to completely refactor the application, you also need to have the original data access layer implement the new `IAlbumDataContext` interface. (See Listing 9-4.)

***Listing 9-4: The concrete AlbumDataContext***

```
using System.Collections.Generic;

namespace TestingDAL.Models
{
    public class AlbumDataContext: IAlbumDataContext
    {
        public List<Album> GetAlbums()
        {
```

```
            ...
        }

        public Album GetAlbum(int id)
        {
            ...
        }
    }
}
```

If you compare this with the old version, you will notice that the only difference besides the implementation of the interface is that the methods are now instance methods and not static ones.

## Step 4 – Writing the Controller Using the DI Pattern

The biggest changes need to be applied to the implementation of the controller. As you already know from the previous chapter, the central concept of Dependency Injection is that the methods must not create their own private instance of the depended on component (in this sample of the `AlbumDataContext`), but they must use the instance that has been passed to the controller when it was instantiated. In order to that, there must be a private field inside the controller to hold the instance of the DOC:

```
private readonly IAlbumDataContext _albumDataContext;
```

Furthermore, a constructor of the controller must allow the caller to supply the instance of the data access layer that needs to be used:

```
public AlbumController(IAlbumDataContext albumDataContext)
{
    _albumDataContext = albumDataContext;
}
```

In a live production environment, the default controller factory of the ASP.NET Framework instantiates the controller by using its parameterless constructor, so you need to add another constructor that creates an instance of the LINQ to SQL–based data access layer:

```
public AlbumController()
{
    _albumDataContext= new AlbumDataContext();
}
```

This constructor creates a compile-type dependency to a specific concrete implementation of the data context class, so it makes the implementation of the DI pattern a bit dirty. In Chapter 16 you will see that this can be avoided replacing the default controller factory with one factory

that uses an Inversion of Control Container to instantiate the controllers supplying the right dependencies.

And finally, the methods that use the data context to retrieve the albums must be changed to use the private field and not the static method as before:

```
_albumDataContext.GetAlbums();
```

The new version of the `AlbumController`, after all the changes, is shown in Listing 9-5.

***Listing 9-5: AlbumContoller***

```
using System.Web.Mvc;
using TestingDAL.Models;

namespace TestingDAL.Controllers
{
    [HandleError]
    public class AlbumController : Controller
    {
        private IAlbumDataContext _albumDataContext;


        public AlbumController()
        {
            _albumDataContext= new AlbumDataContext();
        }

        public AlbumController(IAlbumDataContext albumDataContext)
        {
             albumDataContext = albumDataContext;
        }


         public ActionResult Index()
         {
             ViewData["Title"] = "List of Pink Floyd Albums";
           ViewData["Albums"] = _albumDataContext.GetAlbums();


             return View();
         }

         public ActionResult Individual(int id)
         {
           var album = _albumDataContext.GetAlbum(id);


             ViewData["Title"] = album.Name;
             ViewData["Album"] = album;
```

```
            return View();
        }

    }
}
```

This implementation takes only a few more lines of code but is much more testable then before. It is also more maintainable and extensible, as you will see in the next sections.

# Testing the Application

Now that the application has been refactored, it's possible to test the controllers in isolation, without the overhead of hitting the database for every test. The key concept of these tests is that the data context will be automatically created at runtime by the mocking framework.

In the previous chapter, you learned that mock objects have two modes: the record mode and the replay mode. In all the previous samples, you made the switch calling the method `mocks.ReplayAll()` when you finished recording all the expectations and calling the method `mocks.VerifyAll()` when you finished exercising the SUT and verifying all the expectations. But RhinoMocks also has another syntax to mark the beginning and the end of the two modes. This syntax uses the `using` statement to group all the instructions that belong to each mode.

The record mode is defined like this:

```
using(mocks.Record())
{
    //All the expectations are set here
}
```

And the replay mode (sometimes also referred to as playback mode as consequence of the name of the method) is:

```
using (mocks.Playback())
{
  //The exercise phase and the asserts are inside this block
}
```

In my opinion this is an easier syntax as it clearly separates the two modes and will not lead to unexpected results if you forget to call one of the two methods.

In the next Try It Out section, you will finally write the test for the `AlbumController`.

type="activity"

## Try It Out: Testing a Controller

This sample builds on the refactored code for the application that lists music albums. So, before you begin with the steps, create a new ASP.NET MVC project, click Yes when asked if you want to create a test project, and copy all the code of the previous listings in the newly created project (`Album` and `IAlbumDataContext` in the `Model` folder and `AlbumController` in the `Controllers` folder). Then follow these steps:

1. Create a new test file in the test project, and call it **`AlbumControllerTest.cs`**.
2. Erase all the content of the class, leaving only the class declaration.
3. Inside the class add the two following fields:

```
private MockRepository mocks;
private IAlbumDataContext mockContext;
```

4. Add a new method, marked with the `TestInitialize` attribute:

```
[TestInitialize]
public void SetupMocks()
{
    mocks = new MockRepository();
    mockContext = mocks.DynamicMock<IAlbumDataContext>();
}
```

5. The first test will prove the correctness of the `Index` method of the controller:

```
[TestMethod]
public void Index_Returns_10_Albums()
{
}
```

6. Inside this method add the code for the setup phase of the test:

```
// Arrange
List<Album> albumList = new List<Album>();
for (int i = 0; i < 10; i++)
{
    albumList.Add(new Album());
}
using(mocks.Record())
{
    Expect.Call(mockContext.GetAlbums()).Return(albumList);
}

AlbumController controller = new AlbumController(mockContext);
```

7. Then enter the Playback mode:

```
using (mocks.Playback())
{
}
```

8. And inside the playback mode block, exercise the `Index` method and verify that it returns the expected results:

```
// Act
ActionResult result = controller.Index();

// Assert
ViewResult viewResult = result as ViewResult;
Assert.IsNotNull(viewResult, "Not a view result");

IList<Album> list = viewResult.ViewData["Albums"] as IList<Album>;
Assert.AreEqual(10, list.Count);
Assert.AreEqual(String.Empty,viewResult.ViewName);
```

9. Then add the test method that verifies the `Individual` action:

```
[TestMethod]
public void Can_Get_Specific_Album()
{
    // Arrange
    Album specificAlbum = new Album()
                              {
                                  ID = 34,
                                  Name = "The Division Bell",
                                  ReleaseYear = "1994"
                              };
    using (mocks.Record())
    {
        Expect.Call(mockContext.GetAlbum(0)).IgnoreArguments()
            .Return(specificAlbum);
    }

    AlbumController controller = new AlbumController(mockContext);

    using (mocks.Playback())
    {
        // Act
        ActionResult result = controller.Individual(0);

        // Assert
        ViewResult viewResult = result as ViewResult;
        Assert.IsNotNull(viewResult, "Not a view result");

        Assert.AreEqual("The Division Bell", viewResult.ViewData["Title"]);

        Album album = viewResult.ViewData["Album"] as Album;
        Assert.IsNotNull(album, "No Album provided");
        Assert.AreEqual("1994", album.ReleaseYear);
        Assert.AreEqual("The Division Bell", album.Name);
        Assert.AreEqual(34, album.ID);

        Assert.AreEqual(String.Empty, viewResult.ViewName);
    }
}
```

10. The complete test class will look like Listing 9-6.

***Listing 9-6: AlbumControllerTest***

```
using System;
using System.Collections.Generic;
using System.Web.Mvc;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Rhino.Mocks;
using TestingDAL.Controllers;
using TestingDAL.Models;

namespace TestingDAL.Tests.Controllers
{
    /// <summary>
    /// Summary description for AlbumControllerTest
    /// </summary>
    [TestClass]
    public class AlbumControllerTest
    {

        private MockRepository mocks;
        private IAlbumDataContext mockContext;

        [TestInitialize]
        public void SetupMocks()
        {
            mocks = new MockRepository();
            mockContext = mocks.DynamicMock<IAlbumDataContext>();
        }


        [TestMethod]
        public void Index_Returns_10_Albums()
        {
            // Arrange
            List<Album> albumList = new List<Album>();
            for (int i = 0; i < 10; i++)
            {
                albumList.Add(new Album());
            }
            using(mocks.Record())
            {
                Expect.Call(mockContext.GetAlbums()).Return(albumList);
            }

            AlbumController controller = new
AlbumController(mockContext);

            using (mocks.Playback())
            {
                // Act
                ActionResult result = controller.Index();
```

```
            // Assert
            ViewResult viewResult = result as ViewResult;
            Assert.IsNotNull(viewResult, "Not a view result");

            IList<Album> list = viewResult.ViewData["Albums"]
                as IList<Album>;
            Assert.AreEqual(10, list.Count);
            Assert.AreEqual(String.Empty,viewResult.ViewName);
        }
    }


    [TestMethod]
    public void Can_Get_Specific_Album()
    {
        // Arrange
        Album specificAlbum = new Album()
                                {
                                    ID = 34,
                                    Name = "The Division Bell",
                                    ReleaseYear = "1994"
                                };
        using (mocks.Record())
        {
            Expect.Call(mockContext.GetAlbum(0)).IgnoreArguments()
                .Return(specificAlbum);
        }

        AlbumController controller = new
AlbumController(mockContext);

        using (mocks.Playback())
        {
            // Act
            ActionResult result = controller.Individual(0);

            // Assert
            ViewResult viewResult = result as ViewResult;
            Assert.IsNotNull(viewResult, "Not a view result");

            Assert.AreEqual("The Division Bell",
                viewResult.ViewData["Title"]);

            Album album = viewResult.ViewData["Album"] as Album;
            Assert.IsNotNull(album, "No Album provided");
            Assert.AreEqual("1994", album.ReleaseYear);
            Assert.AreEqual("The Division Bell", album.Name);
            Assert.AreEqual(34, album.ID);

            Assert.AreEqual(String.Empty, viewResult.ViewName);
```

```
                }
            }
        }
}
```

11. Now run the tests and see them pass, as shown in Figure 9-2.

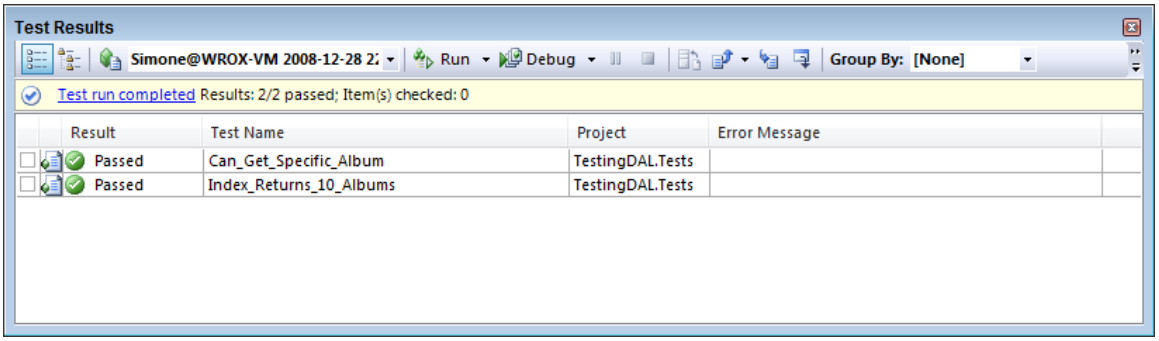


**Figure 9-2**

## HOW IT WORKS

This test uses the mocking library RhinoMocks, as did the other tests in the chapter, so let's focus only on the new elements that are coming out in this specific sample.

At Step 4, in the test initialization method, the mock object of the data context is created, asking the mocking framework to use the `IAlbumDataContext` interface:

```
mockContext = mocks.DynamicMock<IAlbumDataContext>();
```

The test you are writing wants to prove that when the database returns 10 albums, the controller populates the `ViewData` collection with a list containing 10 `Album` objects, and the view rendered is the default one (the one whose name is automatically retrieved by the name of the action). So first a list of 10 empty objects is created and then the mock object is instructed to return it as the return value for the call to the `GetAlbums` method.

```
List<Album> albumList = new List<Album>();
for (int i = 0; i < 10; i++)
{
    albumList.Add(new Album());
}
using(mocks.Record())
{
    Expect.Call(mockContext.GetAlbums()).Return(albumList);
}
```

Notice that the expectation is set inside the using code block that delimits the record mode of the mock repository. The most important line of the test method is the one that creates an instance of the controller, using the constructor that accepts the data context to use as a parameter:

```
AlbumController controller = new AlbumController(mockContext);
```

After this, the mock repository goes in playback mode, exercises the action method, and verifies the expectations. There's not much to be said about this part, since it's the same as in the other examples in this chapter. The only line worth noticing is the one that verifies the name of the view rendered:

```
Assert.AreEqual(String.Empty, viewResult.ViewName);
```

The `ViewName` is empty when the controller doesn't request any specific view to be rendered and relies on the `ActionName` at runtime to get the name of the view.

The other test method wants to prove that when the `Individual` action is called, the album titled "*The Division Bell,*" released during *1994* is rendered on the screen. The expectation for this action is set in a slightly different way than the other test: Instead of specifying a specific ID as the parameter for the action, the `IgnoreArguments` method is used. This instructs the mock framework not to care about the value of the parameter, as long as the method is called.

```
Expect.Call(mockContext.GetAlbum(0)).IgnoreArguments()
    .Return(specificAlbum);
```

The rest of the test method is the same as the other one, so there is no point in repeating the same things again.

Place HIW end rule here

This approach to testing can be applied to almost all the actions that depend on a data access layer or any external component. The next step in the journey through testing is Test Driven Development (TDD), which you will learn about in the next section.

# Developing with a TDD Approach

Thanks to the design for testability in the ASP.NET MVC framework, another methodology that can be used is Test Driven Development, or, as it's usually called, TDD.

This methodology adopts an iterative approach that consists of a few phases:

1. First, the developer adds a new test that covers a new desired feature or improvement that is yet to be implemented.
2. Then the test suite is run. This ensures that the test is well designed. If the test passes, it means that the test was not written correctly, since the new feature hasn't been implemented yet.
3. The production code to implement this feature or improvement is written to make the test pass.
4. The code is then refactored to remove duplications and clean it up.
5. Tests are re-run to verify that the refactoring didn't break anything.

This process is also referred to as the TDD Mantra: "*Red, Green, Refactor*," from the colors of the icons that show the result of a test run.

This approach has some benefits over the classic "testing after coding" approach:

- It ensures the developer understands the requirements before writing the code.
- The resulting code is inherently designed with testability in mind. It has a clear separation of concerns, low coupling (which means that modules of the application depends on interfaces and not on their implementation), and adheres to the single responsibility principle.
- Only the code that makes the test pass is implemented, so there is no needless complexity.
- It always provides 100% code coverage.
- The tests, if written using terms coming from the business/domain, also act as additional documentation of how the code works.

A lot has been written on TDD, and if you are interested in this topic, I really encourage you to read a book about Test Driven Development. In the remaining pages of this chapter, you will get a feel for how to write an application using TDD.

# Requirements

The first step is writing the requirements. They will be formalized in a series of tests, and then the code will be written to pass the tests and implement the application.

The requirements that you are going to implement are:

- If it's Christmas, the view rendered will be the StoreClosed one.
- When it's not Christmas, the Index view will be rendered.

# Testing for Christmas Day

The first test verifies that on the day of Christmas the store is closed:

```
[TestMethod]
public void HomePage_Renders_StoreClosed_View_When_Its_Xmas()
{
    MockRepository mocks = new MockRepository();
    IDateProvider provider =
        mocks.DynamicMock<IDateProvider>();
    using (mocks.Record())
    {
        DateTime xmas = new DateTime(2008,12,25);
        SetupResult.For(provider.GetDate()).Return(xmas);
    }
    HomeController controller = new HomeController(provider);

    using (mocks.Playback())
    {

        ViewResult result = controller.Index() as ViewResult;

        Assert.IsNotNull(result, "A view has not been rendered");
        Assert.AreEqual("StoreClosed", result.ViewName);
    }
}
```

## The Test Fails

You don't want to wait for Christmas to test your application. It's better to use an external provider that returns the current date: `IDateProvider`. The real production implementation will return the

current date (`DateTime.Now()`), but the mock one will always return the day of Christmas.

```
namespace TDDSample.Models
{
    public interface IDateProvider
    {
        DateTime GetDate();
    }
}
```

If you now run the test, the application will not compile, because, obviously, the `HomeController` has not been implemented yet. So, since you first want the test to fail, you "fake" the `HomeController` and implement a nonworking implementation of it:

```
namespace TDDSample.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            throw new NotImplementedException();
        }
}
```

## The Test Passes

Now that the test fails, you can write the controller action to make the test pass, which will be just:

```
namespace TDDSample.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View("StoreClosed");
        }
}
```

The test verifies that the `StoreClosed` view is rendered when it's Christmas, but nothing has been discussed yet about what to do when it's not. Therefore, to keep the code as simple as possible, it always returns `StoreClosed`.

# Testing for Any Other Day

Now the second requirement leads to the following test:

```
[TestMethod]
```

```
public void HomePage_Renders_Index_View_When_Its_Not_Xmas()
{
    MockRepository mocks = new MockRepository();
    IDateProvider provider =
        mocks.DynamicMock<IDateProvider>();
    using (mocks.Record())
    {
        DateTime xmas = new DateTime(2008, 3, 20);
        SetupResult.For(provider.GetDate()).Return(xmas);
    }
    HomeController controller = new HomeController(provider);

    using (mocks.Playback())
    {
        // Execute
        ViewResult result = controller.Index() as ViewResult;

        Assert.IsNotNull(result, "A view has not been rendered");
        Assert.AreEqual("Index", result.ViewName);
    }
}
```

You repeat all the processes, run the test, and watch it fail. (The controller will still return `StoreClosed` view.) (See Figure 9-3.)

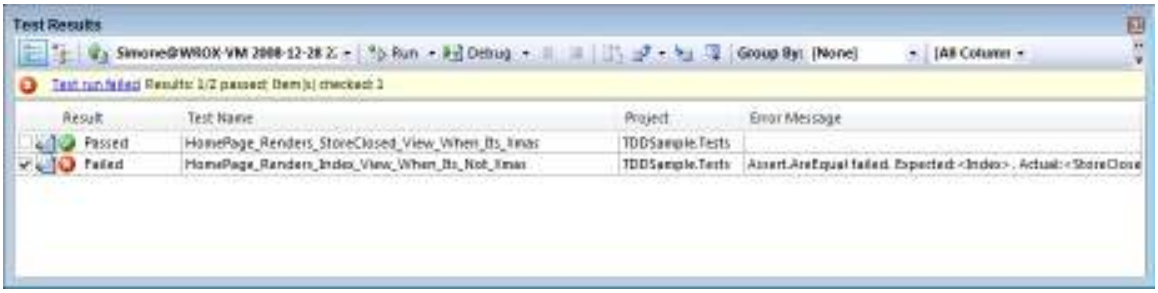


**Figure 9-3**

Now it's time to really implement the check for the day of Christmas:

```
namespace TDDSample.Controllers
{
    public class HomeController : Controller
    {
        private IDateProvider _provider;
        public HomeController(IDateProvider provider)
        {
            _provider = provider;
        }

        public ActionResult Index()
```

```
        {
            DateTime current = _provider.GetDate();
            if(current.Day==25 && current.Month==12)
                return View("StoreClosed");
            return View("Index");
        }

    }
}
```

Now you run all the tests and they all pass.

# Testing for New Year's Day

Let's do it one more time. You want to test that on New Year's Day there is yet another view. You write the test:

```
[TestMethod]
public void HomePage_Renders_HappyNewYear_View_When_Its_NewYear()
{
    MockRepository mocks = new MockRepository();
    IDateProvider provider =
        mocks.DynamicMock<IDateProvider>();
    using (mocks.Record())
    {
        DateTime day = new DateTime(2008, 1, 1);

        SetupResult.For(provider.GetDate()).Return(day);
    }
    HomeController controller = new HomeController(provider);

    using (mocks.Playback())
    {
        // Execute
        ViewResult result = controller.Index() as ViewResult;

        Assert.IsNotNull(result, "A view has not been rendered");

        Assert.AreEqual("HappyNewYear", result.ViewName);

    }
}
```

Again, the test fails and you implement the new code that also takes New Year's Day into account:

```
public ActionResult Index()
{
    DateTime current = _provider.GetDate();
    if(current.Day==25 && current.Month==12)
        return View("StoreClosed");

    if (current.Day == 1 && current.Month == 1)
        return View("HappyNewYear");

    return View("Index");
}
```

# It's Refactoring Time

Now all three tests pass, but the logic in the controller is not very sound, so you decide to remove the `if` statements in order to reduce the complexity of the method (there are currently three exit paths) and increase the readability of the method. It's refactoring time. (See Listing 9-7.)

***Listing 9-7: Home Controller after the refactoring***

```
using System;
using System.Collections.Specialized;
using System.Globalization;
using System.Web.Mvc;
using TDDSample.Models;

namespace TDDSample.Controllers
{
    public class HomeController : Controller
    {
        private IDateProvider _provider;
        private StringDictionary _holidayViews;

        public HomeController(IDateProvider provider)
        {
            _provider = provider;
            _holidayViews = new StringDictionary();
            _holidayViews.Add("25-12", "StoreClosed");
            _holidayViews.Add("01-01", "HappyNewYear");
        }

        public ActionResult Index()
        {
            return View(GetViewByDate(_provider.GetDate()));
        }

        private string GetViewByDate(DateTime date)
        {
            return _holidayViews [date.ToString("dd-MM",
               CultureInfo.InvariantCulture)] ?? "Index";
        }
    }
}
```

After the refactoring, all the three tests keep on passing, as shown in Figure 9-4.

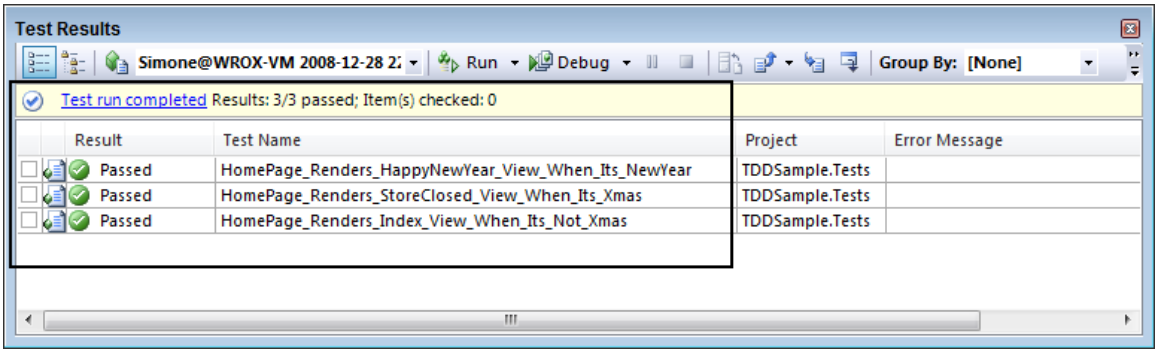


**Figure 9-4**

The `Index` action changed from six lines of code to just one and from three exit paths to just one. Furthermore, now the code is more readable, and it is easier to add a new view for a new holiday. In a real application, you would probably have extracted the dictionary of views to an external repository that the application reads the list of holidays from. This way the controller would only act as the controller and not behave as the model.

# Summary

Even if you decide not to adopt a Test Driven Development approach, testing your applications is something you should do to ensure they are easily maintainable, are well designed, and meet high quality standards. Since testing WebForm applications was not very easy, Microsoft fully embraced these new development methodologies and designed the ASP.NET MVC framework to be test-friendly and allow the developer to easily test an application.

In this chapter, you learned:

- How to create a test project for an ASP.NET MVC application
- How to test actions
- How to verify `ActionResults`

- How to verify that an action renders the correct view or redirects to the correct action
- What the `System.Web.Abstractions` assembly is
- How to mock the `HttpContext` in order to test interaction with the `Http` runtime
- How to test routes
- How to spot when an application is not testable
- How to refactor code so that it is more testable
- How to mock a data access layer used inside a controller's actions
- Another way of switching between Record and Playback mode in mock objects
- How to design an application using the TDD methodology

In the next chapters, you will learn about more advanced features of the ASP.NET MVC framework, starting from building a view that makes use of components. But before moving on to the next topic, take the time to test your knowledge of what you just learned in this chapter.

# Exercises

For these exercises, you have the following controller action:

```
public ActionResult Index(string pageTitle, string language)
{
    ViewData["Message"] = pageTitle +" "+language.toUpperInvariant();
    if (!language.Equals("it") && !language.Equals("en"))
    {
        TempData["Language"] = language;
        return RedirectToAction("NotSupported");
    }

    return View("Index");
}
```

And the `global.asax.cs` file contains the following route registration:

```
routes.MapRoute(
    "Language",
    "{language}/{pageTitle}",
    new { language = "en",
        controller = "Home",
        action = "Index",
        pageTitle = "Home" }
);
```

1. Test that the URL `/it` sends the URL to the `Index` action in the Home controller with the title.set to "Home Page".
2. Test that the Index view is rendered when the languages `it` or `en` are supplied.
3. Add a test that verifies that the test rendered on the page is in the format "*<pagetitle> <LANG>*" and that if the parameters are `BookList` and `en`, the text rendered is `BookList EN`.
4. Verify that when someone requests a page in French the flow is redirected to the `NotSupported` action.
5. Expand the last test to verify also that the `TempData` contains the original language supplied.