# Test Helper

The **TestHelper** library allows simpler unit testing on MVC.Net controllers than can be accomplished through standard mocking and subclassing. It is designed to reduce the amount of work required to test controllers, and therefore should be used by your test projects but not in your web server project.

* [General Concepts](http://mvccontrib.codeplex.com/Wiki/View.aspx?title=TestHelper&referringTitle=Documentation#GeneralConcepts)
* [Examples](http://mvccontrib.codeplex.com/Wiki/View.aspx?title=TestHelper&referringTitle=Documentation#Examples)

## General Concepts

One of the major advantages of the new MVC.Net framework is the ease with which it can be tested. While this is generally true, there are a number of areas where testing the framework becomes difficult to do. The**TestHelper** assists by providing a Controller Factory that creates controllers with internal data members correctly initialized. These include:

* HttpContext
* HttpRequest
* HttpResponse
* HttpSession
* Form
* TempData
* QueryString
* ApplicationPath
* PathInfo

It also provides a way to determine what parameters were passed to RenderView and RedirectToAction.

## Examples

More examples can be found in the Samples directory of the MVC Contrib project. In that directory is a project for MvcContrib.TestHelper.Sample which contains a very simple MVC.Net project and Test. There are a number of tests in the test project which detail how to use the Test Helper. A couple of them are included below.

### Testing data from RenderView method call

[Test]

public void ListControllerSelectsListView()

{

ActionResult result = controller.List();

result.AssertRenderedView().ForView("List");

}

### Testing data from RedirectToAction method call

[Test]

public void AddFormStarShouldRedirectToList()

{

StarsController controller = new StarsController();

ActionResult result = controller.AddFormStar();

result.AssertActionRedirect().ToAction("List");

}

### Testing data from Form and Session variables

[Test]

public void AddSessionStarShouldSaveFormToSession()

{

TestControllerBuilder builder = new TestControllerBuilder();

StarsController controller = new StarsController();

builder.InitializeController(controller);

//note that this is assigned before the controller action. This simulates the server filling out the form data from the request

builder.Form["NewStarName"] = "alpha c";

//this assumes that AddSessionStar takes the form data and adds it to the session

controller.AddSessionStar();

Assert.AreEqual("alpha c", controller.HttpContext.Session["NewStarName"]);

}

# Fluent Route Testing in ASP.NET MVC

Have you written code like this recently?

[Test]

public void tedious\_route\_test()

{

    Global.RegisterRoutes(RouteTable.Routes);

    var httpContext = MockRepository.GenerateStub<HttpContextBase>();

    httpContext.Stub(x =>x.Request).Return(MockRepository.GenerateStub<HttpRequestBase>());

    httpContext.Request.Stub(x => x.PathInfo).Return(**""**);

    httpContext.Request.Stub(x =>x.AppRelativeCurrentExecutionFilePath).Return(**"~/foo/bar"**);

     var routeData = RouteTable.Routes.GetRouteData(httpContext);

     Assert.That(routeData.Values[**"controller"**], Is.EqualTo(**"foo"**));

   Assert.That(routeData.Values[**"action"**], Is.EqualTo(**"bar"**));

}

This mess of nastiness is what is required to test your routes in ASP.NET MVC.  There are a number of things wrong with this.  First and foremost, we're doing magical-mocking here.  That is - we somehow know exactly the two properties we need to mock for this to work.  It turns out this isn't such a magical process as it is looking at reflector and a lot of trial and error.  The next thing wrong that I see is that our route values (and keys) are case-insensitive, so we'd need to include some more flexibility in our Asserts here.

Who wants to deal with that mess?  I sure don't.  Last night, my pal & co-author [**Jimmy Bogard**](http://www.lostechies.com/blogs/jimmy_bogard) banged together a fluent API for testing routes without needing all of this crap.

Here is a test to verify our root route is working properly:

**"~/"**.Route().ShouldMapTo<FooController>();

And a more complex case where we have an action parameter:

**"~/foo/bar/widget"**.Route().ShouldMapTo<FooController>(x => x.Bar(**"widget"**));

These are a lot nicer, as we are dealing with no unnecessary string here, the controller are now strongly-typed (we get intellisense and refactoring support), and we can easily test routes with a single line.

This does have a couple caveats though:  it assumes

* you're using NUnit
* you're using RhinoMocks
* you add your routes to the RouteTable.Routes collection

So how did we do this?  I have to admit, I tried to get this to work on the bus ride to work, but without an internet connection I was dead in the water.  Jimmy helped me grasp the way that Expression<T> works.  His wise words were: "Here's where you just set a breakpoint and fire up the debugger and look."  He's right - the API there is quite opaque.  But once I got it to  work it made a lot of sense.  Expression<T> is awesome!

Here's the method with most of the goods:

public static RouteData ShouldMapTo<TController>(this RouteData routeData,Expression<Func<**TController, ActionResult>> action)**

**where TController : Controller**

**{**

**Assert.That(routeData, Is.Not.Null, "The URL did not match any route");**

**//check controller**

    routeData.ShouldMapTo<TController>();

**//check action**

    var methodCall = (MethodCallExpression) action.Body;

    string actualAction = routeData.Values.GetValue(**"action"**).ToString();

    string expectedAction = methodCall.Method.Name;

    actualAction.AssertSameStringAs(expectedAction);

**//check parameters**

    for (int i = 0; i < methodCall.Arguments.Count; i++)

    {

        string name = methodCall.Method.GetParameters()[i].Name;

        object value = ((ConstantExpression) methodCall.Arguments[i]).Value;

        Assert.That(routeData.Values.GetValue(name),Is.EqualTo(value.ToString()));

    }

     return routeData;

}

This might be a tad brittle, but in my preliminary testing it worked wonders.  This code is now in **[MvcContrib](http://mvccontrib.org" \t "_blank)**'s TestHelper project.

# TestControllerBuilder in MvcContrib

Yesterday, I cracked open the [MvcContrib](http://mvccontrib.org" \t "_blank) source code to see if it had....ummm...ok, I swear I was looking for something legitimate in there. But as soon as I updated the latest and opened the project, my eyes zeroed in on the TestControllerHelper class. So with a quick mutter of "oooh, shiny" to myself, I proceeded to what I knew would be the dearth of my productivity for the rest of my day.

Sitting here this morning, I can report to you that I am now one step closer to being able to adequately (but still not quite usefully) test controller actions that include a RedirectToAction method. Here's a sample test:

[ Test ]

public void Test\_redirect( )

{

TestControllerBuilder builder = new TestControllerBuilder( );

var controller = builder.CreateController<LocationController>( );

controller.TestRedirect( );

Assert.That( builder.RedirectToActionData.ControllerName, Is.EqualTo( "Job" ) );

}

The CreateController method on my TestControllerBuilder uses Castle's [DynamicProxy](http://www.castleproject.org/dynamicproxy/index.html" \t "_blank)to create the controller. It also creates a ControllerContext using dynamic mocks from [Rhino Mocks](http://www.ayende.com/projects/rhino-mocks.aspx) similar to the way Haackselman have [already](http://haacked.com/archive/2007/11/05/rhino-mocks-extension-methods-mvc-crazy-delicious.aspx) [showed](http://www.hanselman.com/blog/ASPNETMVCSessionAtMix08TDDAndMvcMockHelpers.aspx) you and wires it up to the controller.

An interceptor is also attached to the controller to intercept calls to RenderView andRedirectToAction. (Actually, it intercepts all calls but only these two have special handling.) Calls to either method will populate an appropriate object on the builder class. In the example above, I'm using the RedirectToActionData object.

So far, this is all sweet and dandy like raccoon candy. I have indeed verified that it works as advertised. But at present, it seems to work only for controllers that have a parameterless constructor. Which is going to work for exactly none of my controllers.

I have a version working that allows you to pass in constructor arguments to CreateControllerand will submit it as a patch for MvcContrib, assuming it hasn't already been addressed.

Oh yeah, I remember! I wanted to see if MvcContrib had the Flash feature I saw added to (but not yet used in) CodeCampServer.

# [Using MvcContrib's TestControllerBuilder to Test Your ASP.NET MVC Controllers](http://www.shanebauer.com/blog/post/2008/04/Using-MvcContribs-TestControllerBuilder-to-Test-Your-ASPNET-MVC-Controllers.aspx)

April 14, 2008 06:40 by [sbauer](http://www.shanebauer.com/blog/author/sbauer.aspx)

One benefit of ASP.NET MVC (or any MVC framework) is the ability to test your code without having to start up a web server. Sure, you could have always unit tested your model layer, but what if you wanted to ensure that your web application code was interacting correctly with your domain?

#### A Quick Scenario

Lets say, for example, you have an Article controller. The controller has an action called "view". This action accepts one parameter, an integer, representing the article ID. The action code would then take the ID, and query the database for the article record. What happens if it doesn't find a record? Obviously, you could do any number of things. You could redirect them to a not found action or just display a different view. To keep things simple in this example, we'll just redirect the user to the main article page.

**Controllers are Test-Friendly, kind of**

As I said above, a benefit of the MVC framework is the ability to test your code without having to bother with ugly hacks to manipulate a web server and/or the ASP.NET pipeline. The ASP.NET MVC is an improvement in this area without a doubt. However, it's not perfect. There are areas in which you cannot test easily. RedirectData, for example, is one example of something that's difficult to test. There are ways to work around this limitation, but they are exactly that: a work-around. Fortunately, the MVC team has recognized a few areas that could be improved and are actively addressing our concerns for the next preview.

**Introducing the TestControllerBuilder**

The [MvcContrib](http://www.mvccontrib.org" \t "_blank) project is an open source project that boasts quite a few additional features you can add to your project. For example, MvcContrib offers a variety of controller factories that integrate will all of the major IoC containers. In addition, MvcContrib also has a collection of view engines you can use if you're not satisfied with the WebForms view engine that's enabled by default.

The TestControllerBuilder is another contribution that's also available in this project. TestControllerBuilder uses a dynamic proxy and an interceptor to relieve some of the pain behind controller testing. Below is a sample test using the TestControllerBuilder, Rhino Mocks and MbUnit. In order for this to work, I need to make sure my mock instance of IArticleService returns a null instance. I'm using Rhino Mocks "SetupResult" for this.

[view plaincopy to clipboardprint?](http://www.shanebauer.com/blog/post/2008/04/Using-MvcContribs-TestControllerBuilder-to-Test-Your-ASPNET-MVC-Controllers.aspx)

1. [Test]
2. **public** **void** Unknown\_Article\_Redirects\_To\_Index()
3. {
4. TestControllerBuilder builder = **new** TestControllerBuilder();
5. ArticleController controller = builder.CreateController<ArticleController>(**new** **object**[] { articleService });
6. SetupResult.For(articleService.GetArticle(1)).IgnoreArguments().Return(**null**);
7. controller.View(1);
8. Assert.AreEqual(builder.RedirectToActionData.ActionName, "index");
9. }

Now, here's the action implementation.

[view plaincopy to clipboardprint?](http://www.shanebauer.com/blog/post/2008/04/Using-MvcContribs-TestControllerBuilder-to-Test-Your-ASPNET-MVC-Controllers.aspx)

1. **public** **void** View(**int** id)
2. {
3. Article article = articleService.GetArticle(id);
4. **if** (article == **null**)
5. RedirectToAction("index");
6. ArticleControllerViewData viewData = **new** ArticleControllerViewData {Article = article};
7. RenderView("view", viewData);
8. }

**Changes are Coming**

As I mentioned above, changes are coming. I'm not sure how long this code will be appropriate (hopefully not long). Either way, I thought it was worth mentioning.

# [TDD/BDD with ASP.NET MVC](http://russelleast.wordpress.com/2009/01/20/tddbdd-with-aspnet-mvc/)

[with 3 comments](http://russelleast.wordpress.com/2009/01/20/tddbdd-with-aspnet-mvc/#comments)

I have recently been going hammer and tongs with ASP.NET MVC. One of the selling points for ASP.NET MVC is its testability, so while I have been learning this technology, I have found a couple of annoyances.

When you create a new ASP.NET MVC project, the wizard in VS2008 asks you if you want to create a test project. If you do create the test project, you get a couple of pre made tests that really don’t portray good TDD and because you only get a couple of tests, you have low code coverage and have untested code. I have had to back peddle  tests for the existing code.

For doing any development with ASP.NET MVC, the [MvcContrib](http://www.codeplex.com/MVCContrib" \t "_blank) project is an absolute must. The MvcContrib project contains general features that really improve your development, Dependency Injection, type safety, a Grid and much more. I found that the MvcContrib examples, are so straight forward that in 60 seconds of eyeballing the code I could understand it fully.

Keeping to the subject, MvcContrib provides a library that really help when it comes to testing. I have been developing with MonoRail fairly solidly for the last 6 months or so and their are similarities between the Castle test classes and the Library from MvcContrib.

To make things more real world:

* My project is using Unity for its IoC container and I am injecting dependancies into my controllers (again the MvcContrib provides libraries that make this easy to implement).
* For writing my tests/specs, I am using NBehave, NUnit and RhinoMocks 3.5.

I started rewriting the tests for the HomeController because is was the easiest. The tests check that the controller displays the right view.

using System.Web.Mvc;

using RussellEast.Mvc.Controllers;

using MvcContrib.TestHelper;

using NBehave.Narrator.Framework;

using NBehave.Spec.NUnit;

using NUnit.Framework;

using Specification = NUnit.Framework.TestAttribute;

namespace RussellEast.Mvc.Tests.Controllers.Home

{

[TestFixture]

public class When\_displaying\_home\_pages : SpecBase

{

private HomeController controller;

private Story story;

private TestControllerBuilder builder;

[Story]

public override void MainSetup()

{

base.MainSetup();

story = new Story("Home page");

story.AsA("user")

.IWant("to go to the home page")

.SoThat("i can start using the application");

PrepareController();

}

private void PrepareController()

{

builder = new TestControllerBuilder();

controller = builder.CreateController();

}

[Specification]

public void Index()

{

ActionResult result = null;

story.WithScenario("home index page")

.Given("the controller has been prepared", () => controller.ShouldNotBeNull())

.When("the user navigates to the index view", () => result = controller.Index())

.Then("verify that the result is correct", () => result.AssertViewRendered().ForView("Index"));

}

[Specification]

public void About()

{

ActionResult result = null;

story.WithScenario("about page")

.Given("the controller has been prepared", () => controller.ShouldNotBeNull())

.When("the user navigates to the about view", () => result = controller.About())

.Then("verify that the result is correct", () => result.AssertViewRendered().ForView("About"));

}

}

}

The controller code is:

public class HomeController : Controller

{

[Authorize, DefaultAction]

public ActionResult Index()

{

ViewData["Title"] = "Home Page";

ViewData["Message"] = "Sample text being displayed";

return View("Index");

}

public ActionResult About()

{

ViewData["Title"] = "About Page";

return View("About");

}

}

Note: to order for the above tests to pass, you need to supply the name of the view to render.

The tests above cover the behaviours for the HomeController which are very basic, so to raise the bar. The ASP.NET MVC project template comes with an AccountController that works with a membership provider and Forms Authentication. The controller has a constructor that accepts two parameters. Because the “System under test” is the Account Controller i am going to stub the dependencies and write tests for the “Login” and  “Logout” actions.

using System.Web.Mvc;

using System.Web.Security;

using RussellEast.Mvc.Controllers;

using MvcContrib.TestHelper;

using NBehave.Narrator.Framework;

using NBehave.Spec.NUnit;

using NUnit.Framework;

using Rhino.Mocks;

using Specification = NUnit.Framework.TestAttribute;

namespace EWS.Steeple.Tests.Controllers.Authentication

{

[TestFixture]

public class When\_authenticating : SpecBase

{

private AccountController controller;

private IFormsAuthentication formsAuthentication;

private MembershipProvider membershipProvider;

private Story story;

[Story]

public override void MainSetup()

{

base.MainSetup();

story = new Story("Authenticating");

story.AsA("known user to the system")

.IWant("to be authenticated")

.SoThat("i can use the system");

PrepareController();

}

private void PrepareController()

{

TestControllerBuilder builder = new TestControllerBuilder();

formsAuthentication = MockRepository.GenerateStub();

membershipProvider = MockRepository.GenerateStub();

controller = builder.CreateController(new object[] {formsAuthentication, membershipProvider});

}

[Specification]

public void Should\_log\_in\_with\_user\_name\_and\_password\_while\_not\_caching\_details()

{

string username = string.Empty;

string password = string.Empty;

bool rememberMe = false;

ActionResult result = null;

story.WithScenario("log in with validate details")

.Given("a controller instance", ()=> controller.ShouldNotBeNull())

.And("a user name", () => username = "Username")

.And("a password", () => password = "password")

.When("i attempt to log in", () =>

{

membershipProvider.Expect(x => x.ValidateUser(username, password)).Return(true);

formsAuthentication.Expect(x => x.SetAuthCookie(username, rememberMe));

result = controller.Login(username, password, rememberMe, string.Empty);

})

.Then("redirect user to the home page", () =>

{

result.AssertActionRedirect().ToAction(x => x.Index());

formsAuthentication.VerifyAllExpectations();

membershipProvider.VerifyAllExpectations();

});

}

[Specification]

public void Should\_sign\_out\_and\_return\_to\_login\_when\_logging\_out()

{

ActionResult result = null;

story.WithScenario("log out and return to login view")

.Given("a controller instance", () => controller.ShouldNotBeNull())

.When("i log out", () =>

{

formsAuthentication.Expect(x => x.SignOut());

result = controller.Logout();

})

.Then("i expect to be sent to the login view", () =>

{

result.AssertActionRedirect().ToAction("Login");

formsAuthentication.AssertWasCalled(x => x.SignOut());

});

}

}

}

Some points to be aware of.

* I have used two different ways to test the ActionResult was a redirect. The first and my preferred way is using the generic controller type and the lambda expression. I have been forced to use the string argument approach due to the fact that the “Login” action is overloaded. It complained when trying to use a lambda approach.;-(.
* MvcContrib test library provides a “TestControllerBuilder” that not only creates that instance of the controller, but mocks out the HttpContext, Sessions and form data. The docs for MvcContrib explains all. I have used this in my “PrepareController” method.
* The default code in the AccountController has not been changed to suit these tests.

Credits:

Russell East http://russelleast.wordpress.com/

Shane Bauer http://www.shanebauer.com/blog/

Ben Scheirman http://flux88.com/blog/

Kyle Baley http://baley.codebetter.com/

Bottom of Form