Testable MVC

Building Testable ASP.NET MVC Applications

Justin Etheredge

This article discusses:

- ASP.NET MVC
- Building testable solutions
- Dependency injection

This article uses the following technologies:
ASP.NET

Contents

You've probably heard the old adage, "Better, faster, cheaper, pick any two." If you want something good and fast, it isn't going to be cheap, and if you want something fast and cheap, it isn't going to be very good. Cheaper, faster, and better means that we need to write more code more quickly, right? If only it were that simple. Learning to type faster might satisfy two of the requirements, but it isn't going to make the software you develop any better. So how do we make software better? What does "better" mean?

"Better" means producing flexible, maintainable software with a low number of defects; better software is all about long-term maintainability. To achieve this, a key design decision is to ensure that components are loosely coupled. Loosely coupled software has many benefits. One that stands out is that it improves our ability to test solutions. If we write software that can be broken down easily into small parts, it becomes easier to test. It sounds simple when you word it that way, but the amount of software in use today that is difficult to test or maintain shows that it is not as straightforward as we might like to think. Software needs to be coupled to do anything useful, but developers need tools and techniques to decrease coupling so that solutions are easier to test.

Testing Support in ASP.NET MVC

We are seeing something of a renaissance in testing tools and techniques. Everywhere you look you find test frameworks and tools such as mocking containers and dependency injection frameworks. Not all these tools' purpose is for testing, but in the end they allow us to build applications that are much more testable.

From a general perspective, a testable application is an application that is loosely coupled enough to allow its independent parts to be tested in isolation. Writing an application that is testable requires a lot of design work from the start, but to build testable applications, developers must also use tools and frameworks that are designed around paradigms that are easily testable. If you are not working with tools that allow you to build testable applications, no amount of good design will help. Thankfully, ASP.NET MVC was designed from the ground up to be testable, and several key design choices were made to allow developers to create testable applications.

Some would say that the Model View Controller (MVC) pattern is inherently more testable than the Page Controller pattern (the pattern employed by ASP.NET Web Forms) because it encourages separation of the application flow logic and the display logic. In the MVC pattern, the logic that controls the flow of the application resides inside the controller classes, so a developer can easily instantiate and execute this logic as though it were any other .NET class. This allows developers to easily execute the business logic using a simple unit test in much the same way they would test any other POCO (Plain Old CLR Object) class.

Another choice the development team at Microsoft made was to allow many pieces of the framework to be pluggable. Among the more important pieces are the controller factories, which control the instantiation of controller classes. Allowing developers to replace or extend controller factories makes it possible to execute logic that can resolve and inject the dependencies of the controllers. Allowing for the dependencies of a controller to be injected at run time is key to creating more loosely coupled and testable applications.

In traditional ASP.NET, one of the obstacles that developers come across during testing is the plethora of static classes used during each request. The ASP.NET MVC team made the decision to wrap many of the .NET static helper classes (such as HttpContext and HttpRequest) so that they can be replaced during testing with a stub. ASP.NET MVC provides many abstractions to help developers avoid using these classes, but in places where you are required to use them, the wrappers make this code easier to test.

Tooling

Even though ASP.NET MVC provides many of the tools developers need to create testable applications, you cannot rely on it to guide you in the right direction. Instead, you must design your applications deliberately to support testing, and a few additional tools help with this:

- Unit Testing Framework. xUnit.NET by Brad Wilson and Jim Newkirk. xUnit provides a way to run automated unit tests. Many people use MSTest, but many other unit testing frameworks are out there

MSDN Magazine Blog

MSDN Magazine June Issue Preview
Monday morning the June issue of MSDN Magazine will go live on our Web site. Here's what you can expect to find in the magazine. Windows 8 figures pr... More...
Friday, May 31

MSDN Magazine May Issue Preview
Tomorrow afternoon the May issue of MSDN Magazine will go live on our Web site. Here's what you can look forward to in the upcoming issue. Craig Shoe... More...
Tuesday, Apr 30

More MSDN Magazine Blog entries >

Current Issue

Browse All MSDN Magazines

Subscribe to MSDN Flash newsletter

Receive the MSDN Flash e-mail newsletter every other week, with news and information personalized to your interests and areas of focus.

automated unit tests. Many people use MSTest, but many other unit testing frameworks are out there, and I encourage you to take a look at some of them. I chose xUnit.NET because it is simple, easily extended, and has a very clean syntax. I'm using the xUnit test runner for Resharper, but the xUnit GUI test runner is in the Tools folder of the sample application.

- Dependency Injection Framework. Ninject 2 by Nate Kohari. Ninject provides a way to wire up classes in your application. I'll describe this approach in more depth later in the article.
- Mocking Framework. Moq by Clarius Consulting. Moq provides a framework for mocking interfaces and classes during testing.

Application Scenario

To show how to design a testable application, I'll use a simple application scenario in which lists of categories and products are displayed on a Web page. The application also includes simple screens to view, edit, and create categories and products. The simple schema, shown in **Figure 1**, consists of a one-to-many mapping between categories and products.

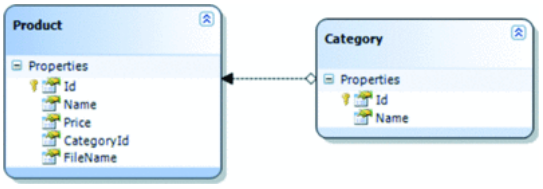

Figure 1 **Schema for the Sample Application**(Click the image for a larger view)

High-Level Application Design

When you create an application, the initial design can go a long way toward helping or hindering the application's long-term health—its maintainability and testability. The approach taken in many architectures is to find the optimal amount of abstraction without creating too much overhead. The MVC pattern already provides some architectural guidance by defining three layers for an application. Some developers might think that these three levels provide enough abstraction to produce large applications. Unfortunately, that often isn't the case, and as you will see, the model can easily be more than a single layer.

You must remember that in many small applications, including the sample for this article, three levels of abstraction is probably sufficient. In these instances, having some views, controllers, and a set of classes to interact with is likely enough. However, several additional layers of abstraction are needed to create a highly testable application. Many architectural patterns at our disposal can help in forming the overall design, including the MVC pattern.

ASP.NET MVC Code-Behind Pages

If you were following ASP.NET MVC before the final release, you may have noticed that the code-behind pages have been removed from the framework. These pages served no functionality in the ASP.NET MVC framework and promoted putting logic into views, where it does not belong. Any logic contained in views is difficult to test in much the same way that code-behind files in ASP.NET Web Forms are difficult to test.

Most of you reading this article are probably familiar with the pattern, but you might not have thought about its implications for application testability. The first part of the MVC pattern is the view, which contains logic for rendering data to a client. Originally, this was a user interface, but a client can be a Web browser, a Web service, client-side JavaScript, and so on. The view should be used only to render data for a consumer, and the amount of logic needed should be abstracted into helper classes as much as possible.

The model represents the back end of an application. This part of the application, which is very loosely defined in the ASP.NET MVC implementation of the pattern, can take on many different forms, and it will very likely depend on the scope of your application. In a small application with little complex business logic, for example, this layer might simply be a set of business objects using the Active Record pattern that you interact with directly in the controller layer.

The controller orchestrates the flow of the application by taking data from the model and passing it to the appropriate view. Since this class is separate from the display logic, by employing a few techniques you should be able to instantiate this class in a unit test and have no dependencies on the ASP.NET runtime. This enables complete testing of the controller without having to run inside a Web server.

Abstracting the Model

When looking at these layers, you might see a few places where additional abstraction would make it easier to test an application. This is common with more complex applications, one area being between the controller and the model. If you have the layering I described earlier, your controller actions might

look something like the following:

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(int id, FormCollection formCollection)
{
    Category category = Category.FindById(id);
    category.Name = formCollection["name"];
    category.Save();
    return RedirectToAction("List");
}
```

The challenges here are at least twofold. First, you cannot easily test this code without having a database connection. Because we are using the Active Record pattern, our data access code is coupled to the domain entity, so we cannot easily swap it out. Also, if you have business logic that is not entirely contained in the domain entity, you could start leaking business or application logic into the controller class. This might lead to duplication of code or even worse—bugs stemming from inconsistencies in implementation. This can hinder testability because you now need to make sure you are testing the same behavior in multiple places within the application.

The Repository Pattern

The next step in improving the testability of this design is to leverage the Repository pattern, which is detailed by Martin Fowler in his book Patterns of Enterprise Application Architecture. The Repository pattern is often misinterpreted as a layer that sits between the application and a database, but in fact it is a layer that sits between your application and any sort of persistent storage. This pattern confines all your data access to a few key places, allowing you to remove it to facilitate testing application logic in isolation.

Implementations of the Repository pattern provide a view of your persistent storage that appears to be an in-memory repository. The interface to such a repository could be similar to this:

```
public interface ICategoryRepository
{
    IEnumerable<Category> FindAll();
    void Insert(Category category);
    Category FindById(int id);
    void Update(Category category);
}
```

The important thing to note here is that you are moving the data persistence out of the domain entity and into a separate class, which you can replace more easily during testing. Applying the pattern to the Edit Action shown earlier would look like **Figure 2**.

Injecting the Repository

The code in **Figure 2** looks clean, but how would you replace the repository class during testing? That's easy. Instead of instantiating the class inside the method, you pass it in the class. In this case, you can pass it in as a constructor parameter. The problem, however, is that you don't control the instantiation of the controller classes. To solve this problem, you can use the dependency injection framework that I referenced earlier. Numerous frameworks are available on the market. I chose to use Ninject 2. (If you aren't familiar with dependency injection, see the article in the September 2005 issue of *MSDN Magazine* by Griffin Caprio, "Dependency Injection." As I said, the first step in using dependency injection is to pass the CategoryRepository into the controller via the controller's constructor:

Testing in Isolation

Some people might think that testing in isolation is not important and choose to only write integration tests. A testing framework such as MSTest or NUnit can still execute integration tests, but they test multiple layers of the application, including making calls across the network, accessing the disk, persisting data to a database, and so on. These sorts of tests are very important and should be present in all applications, but as an application grows, these tests become very slow and can be brittle on different developer machines. Separating unit tests and integration tests allows you to run the unit tests quickly and reliably on developer machines while running your integration tests regularly on your build server.

When you begin to separate classes from one another and test in isolation, you'll find that testing becomes much less brittle and much easier. When you mock a dependency, you control exactly what it returns, and you can more easily test the edge cases of the calling class. Also, because you aren't relying on multiple levels of dependencies, you don't need to account for as much when you are preparing for your test scenarios. Just keep in mind that testing classes in isolation is not a replacement for writing integration tests to make sure that all components work well together.

```
public CategoryController(ICategoryRepository categoryRepository)
{
    this.categoryRespository = categoryRepository;
}
```

Figure 2 Leveraging the Repository

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(int id, FormCollection formCollection)
{
    var categoryRepository = new CategoryRepository();
    Category category = categoryRepository.FindById(id);
    category.Name = formCollection["name"];
    categoryRepository.Update(category);

    return RedirectToAction("List");
}
```

Next, you need to tell Ninject that whenever it sees an ICategoryService interface, it needs to inject an instance of the CategoryService. Every dependency injection framework has a different method of configuration, and Ninject uses a code-based configuration instead of the more usual XML. Because this binding is very straightforward, the code to implement it is just one line:

```
Bind<ICategoryRepository>().To<CategoryRepository>().InTransientScope();
```

This statement tells the dependency injection framework to bind ICategoryRepository to CategoryRepository in a transient scope, which means that each time an ICategoryRepository is requested, you get a new instance of CategoryRepository. This is in contrast to something like a singleton scope, where you continue to get back the same instance of the repository.

With the controller refactored so that the repository is passed in through the constructor, you can now test the behavior of the controller independently of the database. One way to accomplish this is by writing a dummy class (commonly referred to as a stub), which implements the ICategoryRepository interface. You can then implement the methods you need to return dummy data and perform dummy operations, although this seems like a lot of work. In addition, what happens when you need to return more than one set of dummy data for a single method in different tests? You would end up with multiple implementations of the stub, or flags in the stub, and that might be a significant amount of extra effort.

Here is where a good mocking framework can come in. A mocking framework allows you to create mock objects. Mock objects allow developers to emulate, through interfaces or virtual methods, the behavior of another class during testing and then verify that expected events occurred. The behavior of mock objects (also simply called mocks) is what allows you to replace dependencies in your classes and then test those classes without having the real dependencies in place.

This might be a bit confusing, but you need the mocking framework to let you do some stubbing. Yes, that's right. Because a mock is an object that allows you to assert that a certain action has been performed on it, and a stub provides dummy data or actions, it should be clear that in most real-world scenarios you want a mixture of the two. Given this fact, most mocking frameworks have functionality that allows you to stub out methods or properties without asserting any behavior. (If you are still a bit confused about mocks versus stubs, Martin Fowler has a good article on the topic, "Mocks Aren't Stubs."

The code in **Figure 3** shows a test that mocks the ICategoryRepository interface, stubs out the FindAll method so that it returns a dummy list of categories, and then passes the mocked instance to the controller class. Then the List method on the controller class is called, and the result can be asserted. In this case, the test is asserting that the model is an IEnumerable of Category and that there is one category in the list.

Figure 3 List Action Test with Mocks

```
[Fact]
public void ListActionReturnsListOfCategories()
{
    // Arrange

    // create the mock
    var mockRepository = new Mock<ICategoryRepository>();

    // create a list of categories to return
    var categories = new[] { new Category {Id = 1, Name = "test"} };

    // tell the mock that when FindAll is called,
    // return the list of categories
    mockRepository.Setup(cr => cr.FindAll()).Returns(categories);

    // pass the mocked instance, not the mock itself, to the category
    // controller using the Object property
    var controller = new CategoryController(mockRepository.Object);

    // Act
    var result = (ViewResult) controller.List();

    // Assert
    var listCategories = Assert.IsAssignableFrom<IEnumerable<Category>>
(result.ViewData.Model);
    Assert.Equal(1, listCategories.Count());
}
```

At this point, the application is looking good. We have a repository that can be replaced during testing and a method for mocking this repository so that we can test the controller behavior in isolation. For some applications, this might be all that's required with respect to the controllers, but many medium to large applications still have another layer of abstraction in place.

Implementing the Service Layer

A service layer sits on top of an application's domain model and provides a set of operations that can be performed against it. This gives you a place to centralize logic that belongs in your application but might not necessarily belong inside the domain model—logic that would have otherwise likely leaked into the controller's methods. For example, what if you needed to do a security check around the code in **Figure 2**? You don't want to perform the operation in the action method (although you might under certain circumstances) because reusing this action in another place would require you to bring the security along with it. Or, what if you wanted to put a transaction around an operation? You certainly don't want this logic inside the controller class.

Fine-Grained Services

In a more complex application, you might take a more fine-grained service approach and have a set of task or event classes that represent the actions that can be performed in the system. This can have a twofold effect. First, it can reduce the size and complexity of your service classes. These classes can grow quite large if they are not contained. Second, it gives you more control over dependencies because now you only need to worry about the dependencies of individual actions rather than the dependencies of your entire service.

Instead, you can provide a layer of classes that define an application's services and use those classes to perform different actions. In a simple application, such as the sample in this article, the service layer might very closely mimic the methods on the repository, as you can see in the following code, but if the application had more business logic, these methods would likely represent business actions rather than basic CRUD methods:

```
public interface ICategoryService
{
```

```
    IEnumerable<Category> FindAll();
    Category FindById(int id);
    void Save(Category category);
}
```

For the first two methods in the service, calls are delegated to the repository, and then the results are returned, but you might notice that the services have a Save method instead the Update and Delete methods on the repository. Since the application is able to determine whether an object has been saved, the decision to call Update or Insert on the repository can be left up to the service.

Inserting the service layer into the mix greatly reduces the coupling between the controller classes and the repository classes while keeping your controllers lighter because you can push a lot of logic into the services. The design of the controller classes must be changed to reflect the use of services instead of repositories, so instead of injecting repositories into these classes, you inject the services. In turn, you inject the repositories into the services. This might sound complicated, but with dependency injection, you won't even notice—it all gets wired up for you.

The entire dependency injection configuration (including the Product repository and service) looks like **Figure 4**.

Figure 4 Dependency Injection Configuration

```
public class DefaultModule: NinjectModule
{
    public override void Load()
    {
        Bind<ICategoryService>().To<CategoryService>().InTransientScope();
        Bind<ICategoryRepository>().To<CategoryRepository>().InTransientScope();

        Bind<IProductService>().To<ProductService>().InTransientScope();
        Bind<IProductRepository>().To<ProductRepository>().InTransientScope();
    }
}
```

Now that we are passing repositories and services through constructors, you need to construct the controllers using the dependency injection framework. The dependency injection framework handles the construction and injection of the dependencies, but you need to ask for the controller class from the framework. This is much easier than you might anticipate because the ASP.NET MVC team made the controller factories pluggable. You simply need to implement a controller factory as in **Figure 5**, and you can then easily replace the default instantiation of the controller classes.

Figure 5 Ninject Controller Factory

```
public class NinjectControllerFactory : DefaultControllerFactory
{
    private readonly IKernel kernel;

    public NinjectControllerFactory(IKernel kernel)
    {
        this.kernel = kernel;
    }

    protected override IController GetControllerInstance(Type controllerType)
    {
        return (IController)kernel.Get(controllerType);
    }
}
```

Since the controller factory inherits from the default implementation, you just need to override the GetControllerInstance method and then request the type from the Ninject kernel, which is the class that controls object instantiation in Ninject. When the kernel receives a controller type, Ninject self-binds (tries to construct) the type because it is a concrete type. If the kernel receives a CategoryController type, the constructor will have a parameter of type ICategoryService. Ninject looks to see if it has a binding for this type, and when it finds the type, it performs the same action, looking for a constructor. The Ninject kernel instantiates a CategoryRepository and passes it to the constructor for ControllerService. Then it passes the ControllerService object to the constructor for the CategoryController. All this happens inside the dependency injection container simply by asking for the type.

The controller factory must be registered for it to work. Registration requires the addition of only a single line to the global.asax file:

```
public void Application_Start()
{
    // get the Ninject kernel for resolving dependencies
    kernel = CreateKernel();

    // set controller factory for instantiating controller and injecting dependencies
    ControllerBuilder.Current.SetControllerFactory(new NinjectController
Factory(kernel));
}
```

Now, when the user requests a URL and ASP.NET MVC tries to create a controller, it actually ends up being constructed by the dependency injection container. When the controller is tested, you can mock the service to test the controller in isolation, as shown in **Figure 6**.

Figure 6 Testing List Action with Mocked Service

```
[Fact]
public void ListActionReturnsListOfCategories()
{
    // Arrange
```

```
    var mockService = new Mock<ICategoryService>();
    var categories = new[] { new Category { Id = 1, Name = "test" } };
    mockService.Setup(cr => cr.FindAll()).Returns(categories);
    var controller = new CategoryController(mockService.Object);

    // Act
    var result = (ViewResult) controller.List();

    // Assert
    var listCategories = Assert.IsAssignableFrom<IEnumerable<Category>>
(result.ViewData.Model);
    Assert.Equal(1, listCategories.Count());
}
```

Isolating View Logic

You might think at this point that you have enough layers to allow for complete testability. However, we have not yet explored the complexity of views. Imagine a scenario in which you have a product class with a price attribute. (The sample application shows this.) Now let's say that you want to display this price to the user formatted as currency. Some developers might say you should put this requirement in your view, as shown here:

```
<%= Html.Encode(String.Format("{0:c}", this.Model.Price)) %>
```

I personally don't like this approach because I can't easily conduct unit tests on this logic when the field is being formatted in the view. (I don't mind using a UI testing framework, but I want to get as much of my testing as possible done in my unit tests.) One solution is to add a property on the product class to format the price for display, but I don't like doing that because it means that the concerns of my view are starting to leak into the domain layer. Something as simple as a price format might not seem like a big deal, but problems always seem to increase as your application grows in size, and small items start to cause issues. For example, what happens if you need to display a price differently on two pages? Will you start adding overloads for each different version to your domain object?

One good approach is to use the presentation model pattern. Presentation models can be mapped to and from domain objects and can hold view-specific logic and values that can easily be tested. In the sample application, a presentation model has been created for each form. Since most of the forms are quite similar, they inherit from shared base classes. In a more complex application, however, they would likely contain very different code and logic depending on the form.

If a presentation model were employed, the previous statement would instead look like this:

```
<%= Html.Encode(this.Model.DisplayPrice) %>
```

You could easily test this property in a unit test like that shown in **Figure 7**. Note that the DisplayPrice was placed on the abstract base class, so a stub was created that inherits from this base class to facilitate testing. Remember, a stub is just an object that returns fake data used for testing. Since you cannot instantiate an abstract class, you use the stub to test the base functionality.

Figure 7 Unit Test for a Method

```
[Fact]
public void PriceIsFormattedAsCurrency()
{
    // Arrange

    var product = new Product {Price = 9.22m};
    var presentationProduct = new
        PresentationProductStub(product);

    // Act
    string cost = presentationProduct.DisplayPrice;

    // Assert
    Assert.Equal("$9.22", cost);
}
```

The PresentationProductStub class descends from the PresentationProductBase class, which maps to and from the Product class. This causes the List action for the ProductController class to look like the following:

```
public ActionResult List()
{
    var products = productService.FindAll();
    return View(new ProductList().MapList(products));
}
```

The products are being pulled normally from the service, but before they are sent to the view, they are passed to a method on the ProductList class that converts the list of Product classes into a list of ProductList classes. Using a model for your views has several distinct advantages. First, as you already saw, you can add view-specific behavior to your class that can easily be unit tested. This is useful for formatting or transforming data for display. Second, you can use the default model binders built into ASP.NET MVC and place your presentation model on the parameter list of a POST method, as shown here:

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Create(ProductCreate productCreate)
```

```
{
    productService.Save(productCreate.GetProduct());
    return RedirectToAction("List");
}
```

If you are not aware of how the ASP.NET MVC default model binder works, it simply maps fields in the HTML by name onto properties of the class that are in the parameter list. If I put the Product class in the parameter list in the preceding code, a user could maliciously POST a piece of data named "Id" and overwrite the key on the model. This is obviously not a good thing. A user could similarly match up the names of any field on my class and overwrite the values. Using models on the views allows you to use the default model binders (if you choose) while still being able to control which properties are mapped to your domain object.

The alternative is to define mapping classes that you can use to map entities coming in through action method parameters to entities that you want to save to the database. This allows you to control which properties to copy before saving to the database. Another option is to create a reflection-based mapping tool that you could use to map the fields that you want, similar to the AutoMapper tool created by Jimmy Bogard.

Testing Routes

Now, with many of the high-level architectural tips for testing out of the way, let's focus on more fine-grained testing approaches. The first step in implementing an ASP.NET MVC application is to write tests to verify the behavior of the routes within the application. It is important to ensure that the base route

"~/" will forward to the appropriate controller and action, and that other controllers and actions forward correctly as well. It is extremely important to have these tests in place from the beginning so that later, as more routes are added to your application, you are guaranteed that routes that were already defined are not broken.

Start by defining the test for the default route (see **Figure 8**). In this application, the default controller is defined to be the "category" controller, and the default action in this controller is set to "list". Our test should check the base route and assert that the correct values are contained in the route data.

Figure 8 Testing Route Data

```
[Fact]
public void DefaultUrlRoutesToCategoryControllerAndListAction()
{
    // Arrange
    var context = ContextHelper.GetMockHttpContext("~/");
    var routes = new RouteCollection();
    MvcApplication.RegisterRoutes(routes);

    // Act
    RouteData routeData = routes.GetRouteData(context.Object);

    // Assert
    RouteTestingHelper.AssertRouteData(routeData, "Category", "List", "");
}
```

This test is using a helper function to assert the values of the default route, shown here:

```
public static void AssertRouteData(RouteData routeData,
    string controller, string action, string id)
{
    Assert.NotNull(routeData);
    Assert.Equal(controller, routeData.Values["controller"]);
    Assert.Equal(action, routeData.Values["action"]);
    Assert.Equal(id, routeData.Values["id"]);
}
```

Using this same general test pattern, you can pass any route that follows the Controller/Action/Id pattern (such as "~/Product/Edit/12") to the HttpContext and assert the values (see **Figure 9**).

Figure 9 Testing Route Data with Helper

```
[Fact]
public void ProductEditUrlRoutesToProductControllerAndListAction()
{
    // Arrange
    var context = ContextHelper.GetMockHttpContext("~/Product/Edit/12");
    var routes = new RouteCollection();
    MvcApplication.RegisterRoutes(routes);

    // Act
    RouteData routeData = routes.GetRouteData(context.Object);

    // Assert
    RouteTestingHelper.AssertRouteData(routeData, "Product", "Edit", "12");
}
```

With this method, the tests for routes are a little verbose (for clarity of learning), and they could certainly be shortened. Ben Scheirman has done some interesting work in this area, and he has a good post on his blog about making very terse route assertions ("Fluent Route Testing in ASP.NET MVC").

Passing Dependencies of Actions as Parameters

The second tip is to pass all dependencies to controller actions as parameters to the action method. I illustrate this with the file download example in the sample application. As I mentioned earlier, an HttpRequest in the ASP.NET runtime is a static class that is incredibly hard to replace or mock during unit testing. In ASP.NET MVC, wrapper classes are provided to allow these classes to be mocked, but the

process of mocking them or stubbing them out can still be complex.

To mock an HttpRequestBase object, you need to mock the HttpContextBase object that it sits on and then fill in several properties for each. For the file-saving problem, you need to mock the HttpServerUtilityBase that sits on HttpContextBase. Instead of trying to create multiple mock objects to mimic all the different pieces of the HttpContextBase, it would be nice to just make the HttpServerUtilityBase a parameter of the action method and then pass in that single mocked class during testing:

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(int id, ViewProduct viewProduct,
    HttpServerUtilityBase server, HttpPostedFileBase imageFile)
{
```

Notice that the third parameter in the preceding code is the type that we need to use, and the type that we need to mock. With the method signature like this, we only need to mock the HttpServerUtilityBase class. If we had accessed this class through this.HttpContext.Server, it would have been necessary to mock the HttpContextBase class as well. The problem is that just adding the HttpServerUtilityBase class to the method parameters won't make it work; you have to provide a way to tell ASP.NET MVC how to instantiate this class. This is where the framework's model binders come in. Note that HttpPostedFileBase already has a custom model binder assigned to it by default.

Model binders are classes that implement the IModelBinder interface and provide a way for ASP.NET MVC to instantiate types on action methods. In this case, we need a model binder for the HttpServerUtilityBase. We can create a class that looks like the following:

```
public class HttpServerModelBinder: IModelBinder
{
    public object BindModel(ControllerContext controllerContext,
        ModelBindingContext bindingContext)
    {
        return controllerContext.HttpContext.Server;
    }
}
```

The model binder has access to both the controller's context and a binding context. This model binder simply accesses the controller's context and returns the HttpServerUtilityBase class on the HttpContext class. All that is left is to tell the ASP.NET MVC runtime to use this model binder when it finds a parameter of type HttpServerUtilityBase. Note that this code is placed in the Global.asax file as so:

```
public void Application_Start()
{
    // assign model binder for passing in the
    // HttpServerUtilityBase to controller actions
    ModelBinders.Binders.Add(typeof(HttpServerUtilityBase), new
HttpServerModelBinder());
}
```

Now, when an HttpServerUtilityBase is passed as a parameter to an action method, the ASP.NET MVC runtime invokes the BindModel method on the HttpServerModelBinder to get an instance of that class. It is very simple to create model binders, and they make testing controller actions much easier.

Testing with Action Results

One of the early complaints about ASP.NET MVC concerned the difficulty of testing operations such as which view was being rendered from an action method. You had to mock the controller context, a view engine, and so on. It was really painful. In preview 3, the ASP.NET MVC team added the concept of action results, which are classes that descend from the ActionResult class and represent a task that the action is going to perform. Now, in ASP.NET MVC, every action method returns the type ActionResult, and a developer can choose from a number of built-in action-result types or create his or her own. This helps with testing because you can invoke an action method and inspect the result to see what occurred. Let's take the example of a ViewResult that we can create by invoking the View method on the base Controller class, as shown here:

```
public ActionResult List()
{
    IEnumerable<Category> categories = categoryService.FindAll();
    return View(ViewCategory.MapList(categories));
}
```

In this action method, a list of categories is passed as the model to the view, but no view name is provided. This means that this action will render the default view named List. If you want to make sure that this action always returns an empty view name, you can write a test like **Figure 10**.

Figure 10 Testing Action Results

```
[Fact]
public void ListActionReturnsViewResultWithDefaultName()
{
    // Arrange
    var mockService = new Mock<ICategoryService>();
    var controller = new CategoryController(mockService.Object);

    // Act
    var result = controller.List();

    // Assert
    var viewResult = Assert.IsType<ViewResult>(result);
```

```
        Assert.Empty(viewResult.ViewName);
    }
```

This test mocks the CategoryService to create the controller class. Then the test invokes the List action to get the ActionResult object. The test finally asserts that the result is of type ViewResult and the name of the view is empty. It would be very easy to test the view name against some known string value, or in cases where the return type is something a bit more complex (JSON format, for example), to check the Data property on the result to ensure that it contains the expected information as returned by a call to the action method.

Wrapping Up

The ASP.NET MVC team has invested a great deal of effort into creating a flexible architecture that allows for easy testing. Developers can now more easily test their systems because of features like pluggable controller factories, action result types, and wrappers around the ASP.NET context types. All this provides ASP.NET MVC developers with a good starting point, but the onus is still on developers to design and construct applications that are loosely coupled and testable. I hope this article helps as you progress down this path. If you are interested, I've listed these in the Recommended Reading sidebar.

Recommended Reading

- *Agile Principles, Patterns, and Practices in C#* by Robert C. Martin and Micah Martin (Prentice Hall, 2006)
- *Patterns of Enterprise Application Architecture* by Martin Fowler (Addison-Wesley Professional, 2002)
- *Microsoft .NET: Architecting Applications for the Enterprise* by Dino Esposito and Andrea Saltarello (Microsoft Press, 2008)

Justin Etheredge is a Microsoft C# MVP, author at CodeThinked.com, and founder of the Richmond Software Craftsmanship Group. He is a senior consultant at Dominion Digital in Richmond, Virginia, where he provides guidance in designing and building systems of all shapes and sizes.