

[Peter Provost's Geek Noise](#)

Rants, rambles, news and notes from another geek

- [RSS](#)
- [Home](#)
- [Archives](#)
- [Talks](#)
- [GitHub](#)
- [CodePlex](#)
- [Twitter](#)

Unit Testing ASP.NET Web API

Saturday, June 16, 2012 @ 02:00 PM



A couple of days ago a colleague pinged me wanting to talk about unit testing an ASP.NET Web API project. In particular he was having a hard time testing the POST controller, but it got me thinking I needed to explore unit testing the new Web API stuff.

Since it is always fun to add unit tests to someone else's codebase, I decided to start by using the tutorial called [Creating a Web API that Supports CRUD Operations](#) and the [provided solution](#) available on [www.asp.net](#).

What should we test?

In a Web API project, one of the things you need to ask yourself is, "What do we need to test?"

Despite my passion for unit testing and TDD, you might be surprised when I answer "as little as possible." You see, when I'm adding tests to legacy code, I believe strongly that you should only add tests to the things that need it. There is very little value-add in spending hours adding tests to things that might not need it.

I tend to follow the [WELC approach](#), focusing on adding tests to either areas of code that I am about to work on, or areas that I know need some test coverage. The goal when adding tests for legacy code like this is to "pin" the behavior down, so you at least can make positive statements about what it does do right now. But I only really care about "pinning" those methods that have interesting code in them or code we are likely to want to change in the future. (Many thanks to my friend [Arlo Belshee](#) for promoting the phrase "pinning test" for this concept. I really like it.)

So I'm not going to bother putting any unit tests on things like `BundleConfig`, `FilterConfig`, or `RouteConfig`. These classes really just provide an in-code way of configuring the various conventions in ASP.NET MVC and Web API. I'm also not going to bother with any of the code in the `Content` or `Views` folders, nor will I unit test any of the JavaScript (but if this were not just a Web API, but a full web app with important JavaScript, I would certainly think more about that last one).

Since this is a Web API project, its main purpose is to provide an easy to use REST JSON API that can be used from apps or web pages. All of the code that *matters* is in the `Controllers` folder, and in particular the `ProductsController` class, which is the main API for the project.

This is the class we will unit test today.

Unit Tests, not Integration Tests

Notice that I said *unit test* in the previous sentence. For me a unit test is **the smallest bit of code that I can test in isolation from other bits of code**. In .NET code, this tends to be classes and methods. Defining unit test in this way makes it easy to find what to test, but sometimes the *how* part can be tough because of the "in isolation from other bits" part.

When we create tests that bring up large parts of our system, or of the environment, we are really creating **integration tests**. Don't get me wrong, I think integration tests are useful and can be important, but I do not want to get into the habit of depending entirely on integration tests when writing code. Creating a testable, cohesive, decoupled design is important to me. It is the only way to achieve the design goal of simplicity (maximizing the amount of work **not done**).

But in this case we will be adding tests to an existing system. To make the point, I will try to avoid changing the system if I can. Because of this we may find ourselves occasionally creating integration tests because we have no choice. But we can (and should) use that feedback to think about the design of what we have and whether it needs some refactoring.

Analyzing the ProductsController class

The `ProductsController` class isn't too complex, so it should be pretty easy to test. Let's take a look at the code we got in the download:

ProductsController.cs

```
1 namespace ProductStore.Controllers
2 {
3     public class ProductsController : ApiController
4     {
5         static readonly IProductRepository repository = new ProductRepository();
6
7         public IEnumerable<Product> GetAllProducts()
8         {
9             return repository.GetAll();
10        }
11
12        public Product GetProduct(int id)
13        {
14            Product item = repository.Get(id);
15            if (item == null)
16            {
17                throw new HttpResponseException(new HttpResponseMessage(HttpStatusCode.NotFound));
18            }
19            return item;
20        }
21
22
23        public IEnumerable<Product> GetProductsByCategory(string category)
24        {
25            return repository.GetAll().Where(
26                p => string.Equals(p.Category, category, StringComparison.OrdinalIgnoreCase));
27        }
28
29
30        public HttpResponseMessage PostProduct(Product item)
31        {
32            item = repository.Add(item);
33            var response = Request.CreateResponse<Product>(HttpStatusCode.Created, item);
34
35            string uri = Url.Link("DefaultApi", new { id = item.Id });
36            response.Headers.Location = new Uri(uri);
37            return response;
38        }
39
40
41        public void PutProduct(int id, Product contact)
42        {
43            contact.Id = id;
44            if (!repository.Update(contact))
45            {
46                throw new HttpResponseException(new HttpResponseMessage(HttpStatusCode.NotFound));
47            }
48        }
49
50
51        public HttpResponseMessage DeleteProduct(int id)
52        {
53            repository.Remove(id);
54            return new HttpResponseMessage(HttpStatusCode.NoContent);
55        }
56    }
57 }
58 }
```

We have three Get methods, and a method for each of Post, Put and Delete.

Straight away I see the first problem: The `IProductRepository` is private and static. Since I said I didn't want to change the product code, this is an issue. As a static, readonly, private field, we really don't have any way to replace it, so in this one case, I will need to change the product to a more testable design. This isn't as bad as it looks, however, since in the tutorial they acknowledge that this is a temporary measure in their code:

Calling `new ProductRepository()` in the controller is not the best design, because it ties the controller to a particular implementation of `IProductRepository`. For a better approach, see [Using the Web API Dependency Resolver](#).

In a future post I will show how to resolve this dependency with something like Ninject, but for now we will just use manual dependency injection by creating a testing constructor. First I will make the repository field non-static. Then I add a second constructor which allows me to pass in a repository. Finally I update the default constructor to initialize the field with an instance of the concrete `ProductRepository` class.

This approach of creating a testing constructor is a good first step, even if you are going to later add a dependency injection framework. It allows us to provide a stub value for the dependency when we need it, but existing clients of the class can continue to use the default constructor.

Now the class looks like this.

ProductsController.cs

```

1 namespace ProductStore.Controllers
2 {
3     public class ProductsController : ApiController
4     {
5         readonly IProductRepository repository;
6
7         public ProductsController()
8         {
9             this.repository = new ProductRepository();
10        }
11
12        public ProductsController( IProductRepository repository )
13        {
14            this.repository = repository;
15        }
16
17        // Everything else stays the same
18    }
19 }

```

Testing the easy stuff

Now that we can use the testing constructor to provide a custom instance of the `IProductRepository`, we can get back to writing our unit tests.

For these tests I will be using the [xUnit.net](http://xunit.net) unit testing framework. I will also be using Visual Studio 2012 Fakes to provide easy-to-use Stubs for interfaces we depend on like `IProductRepository`. After using NuGet to get an xUnit.net reference in the test project, I added a project reference to the `ProductStore` project. Then by right-clicking on the `ProductStore` reference and choosing **Add Fakes Assembly**, I can create the Stubs I will use in my tests.

Testing all of the methods except for `PostProduct` is pretty straightforward.

GetAllProducts

This is a very simple method that just returns whatever the repository gives it. No transformations, no deep copies, it just returns the same `IEnumerable` it gets from the repository. Here's the test:

Tests for GetAllProducts

```

1 [Fact]
2 public void GetAllProductsReturnsEverythingInRepository()
3 {
4     // Arrange
5     var allProducts = new[] {
6         new Product { Id=111, Name="Tomato Soup", Category="Food", Price = 1.4M },
7         new Product { Id=222, Name="Laptop Computer", Category="Electronics", Price=699.99M }
8     };
9     var repo = new StubIProductRepository
10    {
11        GetAll = () => allProducts
12    };
13    var controller = new ProductsController(repo);
14
15    // Act
16    var result = controller.GetAllProducts();
17
18    // Assert
19    Assert.Same(allProducts, result);
20 }

```

GetProduct

I used two tests to pin the existing behavior of the `GetProduct` method. The first confirms that it returns what the repository gives it, and the second confirms that it will throw if the repository returns null.

Unit Testing the `GetProduct` method

```

1 [Fact]
2 public void GetProductReturnsCorrectItemFromRepository()
3 {
4     // Arrange
5     var product = new Product { Id = 222, Name = "Laptop Computer", Category = "Electronics", Price = 699.99M };
6     var repo = new StubIProductRepository { GetInt32 = id => product };
7     var controller = new ProductsController(repo);
8
9     // Act
10    var result = controller.GetProduct(222);
11
12    // Assert
13    Assert.Same(product, result);
14 }
15
16 [Fact]
17 public void GetProductThrowsWhenRepositoryReturnsNull()
18 {
19     var repo = new StubIProductRepository
20     {
21         GetInt32 = id => null
22     };
23     var controller = new ProductsController(repo);
24
25     Assert.Throws<HttpResponseException>(() => controller.GetProduct(1));
26 }

```

GetProductsByCategory

I just used one test to pin the behavior of GetProductsByCategory.

Unit Testing GetProductsByCategory

```

1 [Fact]
2 public void GetProductsByCategoryFiltersByCategory()
3 {
4     var products = new[] {
5         new Product { Id=111, Name="Tomato Soup", Category="Food", Price = 1.4M },
6         new Product { Id=222, Name="Laptop Computer", Category="Electronics", Price=699.99M }
7     };
8     var repo = new StubIProductRepository { GetAll = () => products };
9     var controller = new ProductsController(repo);
10
11     var result = controller.GetProductsByCategory("Electronics").ToArray();
12
13     Assert.Same(products[1], result[0]);
14 }

```

PutProduct

I used three tests to pin the various aspects of the PutProduct method:

Unit Testing PutProduct

```

1 [Fact]
2 public void PutProductUpdatesRepository()
3 {
4     var wasCalled = false;
5     var repo = new StubIProductRepository
6     {
7         UpdateProduct = prod => wasCalled = true
8     };
9     var controller = new ProductsController(repo);
10    var product = new Product { Id = 111 };
11
12    // Act
13    controller.PutProduct(111, product);
14
15    // Assert
16    Assert.True(wasCalled);
17 }
18
19 [Fact]
20 public void PutProductThrowsWhenRepositoryUpdateReturnsFalse()
21 {
22     var repo = new StubIProductRepository
23     {
24         UpdateProduct = prod => false

```

```

25     };
26     var controller = new ProductsController(repo);
27
28
29     Assert.Throws<HttpResponseException>(() => controller.PutProduct(1, new Product()));
30 }
31
32 [Fact]
33 public void PutProductSetsIdBeforeUpdatingRepository()
34 {
35     var updatedId = Int32.MinValue;
36     var repo = new StubIProductRepository
37     {
38         UpdateProduct = prod => { updatedId = prod.Id; return true; }
39     };
40     var controller = new ProductsController(repo);
41
42     controller.PutProduct(123, new Product { Id = 0 });
43
44     Assert.Equal(123, updatedId);
45 }

```

DeleteProduct

Like the PUT handler, we had a few cases to handle to correctly pin the behavior of this method.

Unit Testing DeleteProduct

```

1  [Fact]
2  public void DeleteProductCallsRepositoryRemove()
3  {
4      var removedId = Int32.MinValue;
5      var repo = new StubIProductRepository
6      {
7          RemoveInt32 = id => removedId = id
8      };
9      var controller = new ProductsController(repo);
10
11     controller.DeleteProduct(123);
12
13     Assert.Equal(123, removedId);
14 }
15
16 [Fact]
17 public void DeleteProductReturnsResponseMessageWithNoContentStatusCode()
18 {
19     var repo = new StubIProductRepository();
20     var controller = new ProductsController(repo);
21
22     var result = controller.DeleteProduct(123);
23
24     Assert.IsType<HttpResponseMessage>(result);
25     Assert.Equal(HttpStatusCode.NoContent, result.StatusCode);
26 }

```

Testing the harder stuff: PostProduct

The `PostProduct` method is where things get interesting. Because the HTTP spec says that when you create a resource from a POST you are supposed to return a created HTTP status code and include a location to the new resource, the method we want to test does some funny things to get the `HttpResponseMessage` assembled.

My first attempt at a test looked like this:

Failed attempt at unit testing PostProduct

```

1  [Fact]
2  public void PostProductReturnsCreatedStatusCode()
3  {
4      // Arrange
5      var repo = new StubIProductRepository
6      {
7          AddProduct = item => item
8      };
9      var controller = new ProductsController(repo);
10
11     // Act
12     var result = controller.PostProduct(new Product { Id = 1 });

```

```

13
14 // Assert
15 Assert.Equal(HttpStatusCode.Created, result.StatusCode);
16 }

```

Unfortunately, that didn't work. You end up getting a `NullReferenceException` thrown by `Request.CreateResponse` because it expects a fair amount of web config stuff to have been assembled. This is a bummer, but it is what it is.

I reached out to [Brad Wilson](#) for help, and we figured out how to test this without going all the way to creating a web server/client pair, but there is clearly a lot of extra non-test code still running. We had to assemble a whole bunch of interesting configuration and routing classes to make the `Request.CreateResponse` method happy, but it did work.

The first test we wrote looked like this:

Successfully unit testing PostProduct

```

1 [Fact]
2 public void PostProductReturnsCreatedStatusCode()
3 {
4     // Arrange
5     var repo = new StubIProductRepository
6     {
7         AddProduct = item => item
8     };
9
10    var config = new HttpConfiguration();
11    var request = new HttpRequestMessage(HttpMethod.Post, "http://localhost/api/products");
12    var route = config.Routes.MapHttpRoute("DefaultApi", "api/{controller}/{id}");
13    var routeData = new HttpRouteData(route, new HttpRouteValueDictionary { { "controller", "products" } });
14    var controller = new ProductsController(repo);
15    controller.ControllerContext = new HttpControllerContext(config, routeData, request);
16    controller.Request = request;
17    controller.Request.Properties[HttpPropertyKeys.HttpConfigurationKey] = config;
18
19    // Act
20    var result = controller.PostProduct(new Product { Id = 1 });
21
22    // Assert
23    Assert.Equal(HttpStatusCode.Created, result.StatusCode);
24 }

```

In a future post, I may take a look at how we might use Visual Studio 2010 Fakes to create a Shim to remove all that config stuff, but this will have to do for now.

Since I knew I needed to make a few more tests to adequately pin the behavior of `PostProduct`, I refactored out the ugly config code into a private method in the test class called `SetupControllerForTests`. I find that when I have issues like this, I really like to keep the weird setup code close to the tests. I generally prefer this over creating abstract test classes because I like it to be very obvious what is happening. I also like my tests to be easily read and understood without having to jump around in the class hierarchy.

Helper method for configuring the controller

```

1 private static void SetupControllerForTests(ApiController controller)
2 {
3     var config = new HttpConfiguration();
4     var request = new HttpRequestMessage(HttpMethod.Post, "http://localhost/api/products");
5     var route = config.Routes.MapHttpRoute("DefaultApi", "api/{controller}/{id}");
6     var routeData = new HttpRouteData(route, new HttpRouteValueDictionary { { "controller", "products" } });
7
8     controller.ControllerContext = new HttpControllerContext(config, routeData, request);
9     controller.Request = request;
10    controller.Request.Properties[HttpPropertyKeys.HttpConfigurationKey] = config;
11 }

```

Now that I have the helper method, I can refactor the status code test, and add two more to check the location and to confirm that it actually calls the `AddProduct` method on the repository.

Final unit tests for PostProduct

```

1 [Fact]
2 public void PostProductReturnsCreatedStatusCode()
3 {
4     var repo = new StubIProductRepository
5     {
6         AddProduct = item => item
7     };
8     var controller = new ProductsController(repo);

```

```

9     SetupControllerForTests(controller);
10
11     var result = controller.PostProduct(new Product { Id = 1 });
12
13     Assert.Equal(HttpStatusCode.Created, result.StatusCode);
14 }
15
16
17 [Fact]
18 public void PostProductReturnsTheCorrectLocationInResponseMessage()
19 {
20     var repo = new StubIProductRepository
21     {
22         AddProduct = item => item
23     };
24     var controller = new ProductsController(repo);
25     SetupControllerForTests(controller);
26
27     var result = controller.PostProduct(new Product { Id = 111 });
28
29     Assert.Equal("http://localhost/api/products/111", result.Headers.Location.ToString());
30 }
31
32 [Fact]
33 public void PostProductCallsAddOnRepositoryWithProvidedProduct()
34 {
35     var providedProduct = default(Product);
36     var repo = new StubIProductRepository
37     {
38         AddProduct = item => providedProduct = item
39     };
40     var controller = new ProductsController(repo);
41     SetupControllerForTests(controller);
42
43     var product = new Product { Id = 111 };
44     var result = controller.PostProduct(product);
45
46     Assert.Same(product, providedProduct);
47 }

```

Conclusion

And now we're done. We have successfully “pinned” the behavior of the entire `ProductsController` class so if we later need to refactor it, we have a way of knowing what the current behavior is. As I discussed in [my previous post about VS 2012 Shims](#), we can't refactor without being able to confirm the current behavior, otherwise we will end up in a Catch-22. Creating “pinning” or “characterization” tests like those created here are the first step to being able to safely and confidently refactor or add new behaviors.

Hopefully this post showed you a few new things. First, we got to see another example of using Stubs in unit tests. Also, we learned a bit about how to deal with the nastiness around the `HttpRequestMessage.CreateResponse` extension method.

Personally I wish that POST handler was as easy to test as the rest of the controller was. One of my favorite things about MVC was always that the separation of concerns let me test things in isolation. When a controller doesn't have tight dependencies on the web stack, it is a well-behaved controller. Unfortunately, when you want to create a controller that followed the HTTP spec for POST, you will find this a bit hard today. But at least we found a way around it.

Let me know what you think!

Posted by Peter Provost Saturday, June 16, 2012 @ 02:00 PM Posted in [Code](#), [Visual Studio](#)

Like 2 Share Tweet  5 Share

[« Anatomy of a File-based Unit Test Plugin Adding Ninject to Web API »](#)

Comments

Comments for this thread are now closed.



7 Comments Peter Provost's Geek Noise

 Login ▾

 Recommend  Share

Sort by Best ▾

Filipe Pinheiro • 4 years ago

I Peter thank you for your post.

I used it to test my Controllers but I bumped with an error after updating my nuget package.
The call to `Url.Link` on POST always returned null.

I manage to correct the problem adding the `routeData` to the request properties

```
controller.Request.Properties[HttpPropertyKeys.HttpRouteDataKey] = routeData;
```

Was this a change on the Web Api implementation ? Is this wrong in your opinion ?

I noticed that when `Link` method called uses the `GetVirtualPath` on the specified route name.

`GetVirtualPath` to get the request `RouteData` calls `HttpRequestMessageExtensions.GetProperty<IHttpRouteData>(request, HttpPropertyKeys.HttpRouteDataKey)` that gets the `RouteData` from the request not the configuration.

Thank you.

13 ^ | v • Share ›

Dave Iffland • 5 years ago

Holy cow this was timely. I heard you mention this in your session at teched, but hadn't gotten to it yet. In my case I was even getting the `NullReferenceException` in `Get` methods because I believe the "current" way of throwing an `HttpResponseException` is this (at least, this is what Dan Roth showed in his session):

```
throw new HttpResponseException(Request.CreateResponse(HttpStatusCode.BadRequest))
for example.
```

That uses the `Request` object with the same null reference. This workaround does feel a bit integration-testy, but it certainly is less lifting than the `SelfHost` process.

1 ^ | v • Share ›



Andy • 4 years ago

Great post - I just hit this area with a new project so the timing was spot on.
Just wondered if you have come across an issue when using `Url.Link` in the controller? I keep getting the error:
`System.ArgumentNullException: Value cannot be null.`

Parameter name: `uriString`

I am now using MVC 4 release.

Regards,

Andy

^ | v • Share ›



Andy ➔ Andy • 4 years ago

So my call to `Url.Link` is returning null :(
`Url.Link("DefaultApi", new { Id = item.Id });`

^ | v • Share ›



Andy ➔ Andy • 4 years ago

Fixed it. I needed to add the `RouteData` to the `Request.Properties`.
`controller.Request.Properties.Add(
HttpPropertyKeys.HttpRouteDataKey, routeData);`

10 ^ | v • Share ›

john nystrom • 4 years ago

THANK YOU FOR THIS! Extremely helpful and saved a TON of time.

^ | v • Share ›



Peter Gfader • 5 years ago

Great walk through.

Wouldn't it be great if those pinning tests existed in the 1st place?

Why did they not write them before?

^ | v • Share ›

 [Subscribe](#)  [Add Disqus to your site](#) [Add Disqus](#) [Add](#)  [Privacy](#)

About Me



Peter Provost is a life-long-learner, hacker, maker, agilista, musician and heavy metal fan. He lives in Colorado with his wife and two children and works as a Program Manager Lead on Microsoft's Visual Studio ALM tools.

[Contact Info](#)

Sponsors

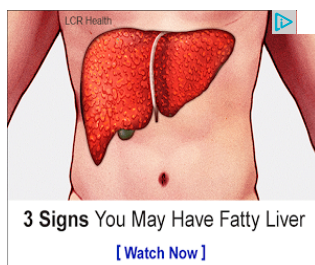
Recent Posts

- [Playing around with JavaScript modules](#)
- [What Really Matters in Developer Testing](#)
- [HTML5 Frameworks](#)
- [Interviewed on Radio TFS](#)
- [Visual Studio 2012 Fakes - Part 3 - Observing Stub Behavior](#)

Tags

[entlib](#) [team rooms](#) [warcraft](#) [vsts](#) [stress](#) [git](#) [agile](#) [planning](#) [tdd](#) [octopress](#) [jekyll](#) [pair programming](#) [hippo](#) [hyper-v](#) [subversion](#) [life skills](#) [remote desktop](#) [security](#) [smartphone](#) [ood](#) [xmi](#) [kids](#) [games](#) [Web API](#) [partial trust](#) [javascript](#) [TDD](#) [t4](#) [cryptography](#) [http](#) [cab](#) [wallpaper](#) [mocking](#) [markdown](#) [vim](#) [Unit Testing](#) [blog](#) [kata](#) [pdc](#) [scams](#) [personal](#) [objectbuilder](#) [science](#) [fakes](#) [.net](#) [sql](#) [server](#) [ie7](#) [webcasts](#) [leadership](#) [powershell](#) [pomodoro](#) [video](#) [ninject](#) [unit testing](#) [windows](#) [server](#) [best of irc](#) [blogengine.net](#) [TV](#) [mmo](#) [teched](#) [dsl](#) [toolkit](#) [database](#) [codelens](#) [uac](#) [wcf](#) [family](#) [keyboard](#) [shortcuts](#) [vista](#) [mef](#) [tortoisesvn](#) [chatzilla](#) [cheat sheet](#) [tfs](#) [lua](#) [vs11](#) [music](#) [patterns & practices](#) [gat](#) [outlook](#) [rules](#) [email](#) [wpf](#) [architecture](#) [vs2010](#) [uml](#) [agile](#) [development](#) [vssdk](#) [tools](#) [patterns & practices](#) [Visual Studio](#) [fps](#) [disbursed](#) [development](#) [blogging](#) [codeplex](#) [ASP.NET](#) [inbox](#) [zero](#) [onenote](#) [training](#) [gaming](#) [web api](#) [software](#) [factories](#) [visual studio](#)

Sponsors





Disclaimer

All content on this blog are provided "AS IS" with no warranties, and confer no rights. Content on this site represents my opinions and do not reflect the opinion of my employer or sponsor(s).

License

This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License](https://creativecommons.org/licenses/by-nc-sa/3.0/).



Copyright © 2013 - Peter Provost - Powered by [Octopress](https://octopress.org/)