Completing the Picture

In his must-read book, Working Effectively with Legacy Code, Michael Feathers states:

Code without tests is bad code. It doesn't matter how well written it is; it doesn't matter how pretty or object-oriented or well-encapsulated it is. With tests, we can change the behavior of our code quickly and verifiably. Without them, we really don't know if our code is getting better or worse.

Those are strong words. However, we completely agree with him. What he calls the "Cover [with tests] and Modify" approach is demonstratably more efficient than the "Edit and Pray [because there are no automated tests]" approach. And although most of us know this, either through our formal education or through the "school of hard knocks", oftentimes we find ourselves writing what Mr. Feathers calls "bad code". The reasons vary, but typically it's because writing testable code can be difficult, especially when developing on top of certain frameworks.

The good news, in this regard, is that ASP.NET Web API was developed with testability in mind. In this chapter we will demonstrate how to achieve high levels of code coverage relatively easily, and how to safely refactor code.

Finally, based on feedback from the previous edition of the book, we will demonstrate how to consume the task-management service using a simple ASP.NET MVC Web application.

We're in the homestretch now, so let's finish strong!

# Testing the API

Our usual approach to automated testing employs a mix of integration tests and unit tests, with a bias towards unit tests. This is because though integration tests are needed to ensure everything works together, unit tests typically provide much less "friction" when trying to achieve high levels of code coverage (e.g., unit tests don't require access to a database).

In this section we will test the Get Tasks operation. We have chosen Get Tasks because it represents a superset of most of the other operations; i.e., it involves paging, database access, type mapping, hypermedia links, etc. We'll begin by putting unit tests in place, and then we'll go end-to-end with an integration test. A familiarity with NUnit and Moq would be helpful at this point; however, you will still be able to benefit by following along with the test implementation, regardless of your experience with these frameworks.

## Unit Testing

The first thing we need to do is add some dependencies to the WebApi2Book.Web.Api.Tests project. With the solution open, run the following commands in the Package Manager Console to install the testing and mocking frameworks that we introduced in Chapter 3, respectively:

install-package NUnit WebApi2Book.Web.Api.Tests

install-package Moq WebApi2Book.Web.Api.Tests

Next, run the following commands, in this order, in the Package Manager Console to install some ASP.NET Web API framework dependencies:

install-package Microsoft.AspNet.WebApi.WebHost WebApi2Book.Web.Api.Tests

install-package Microsoft.Net.Http WebApi2Book.Web.Api.Tests

Now that the external dependencies have been added, add the following project references to the WebApi2Book.Web.Api.Tests project:

WebApi2Book.Data

WebApi2Book.Web.Api.Models

WebApi2Book.Web.Api

### Testing the Controller

At last, we are ready to write some code. Implement the TasksControllerTest class as follows:

using System;

using System.Net.Http;

using System.Web.Http;

using Moq;

using NUnit.Framework;

using WebApi2Book.Data;

using WebApi2Book.Web.Api.Controllers.V1;

using WebApi2Book.Web.Api.InquiryProcessing;

using WebApi2Book.Web.Api.MaintenanceProcessing;

using WebApi2Book.Web.Api.Models;

namespace WebApi2Book.Web.Api.Tests.Controllers.V1

{

[TestFixture]

public class TasksControllerTest

{

[SetUp]

public void SetUp()

{

\_pagedDataRequestFactoryMock = new Mock<IPagedDataRequestFactory>();

\_allTasksInquiryProcessorMock = new Mock<IAllTasksInquiryProcessor>();

\_taskByIdInquiryProcessorMock = new Mock<ITaskByIdInquiryProcessor>();

\_addTaskMaintenanceProcessorMock = new Mock<IAddTaskMaintenanceProcessor>();

\_updateTaskMaintenanceProcessorMock = new Mock<IUpdateTaskMaintenanceProcessor>();

\_controller = new TasksController(

\_addTaskMaintenanceProcessorMock.Object,

\_taskByIdInquiryProcessorMock.Object,

\_updateTaskMaintenanceProcessorMock.Object,

\_pagedDataRequestFactoryMock.Object,

\_allTasksInquiryProcessorMock.Object);

}

private Mock<IPagedDataRequestFactory> \_pagedDataRequestFactoryMock;

private Mock<IAllTasksInquiryProcessor> \_allTasksInquiryProcessorMock;

private Mock<ITaskByIdInquiryProcessor> \_taskByIdInquiryProcessorMock;

private Mock<IAddTaskMaintenanceProcessor> \_addTaskMaintenanceProcessorMock;

private Mock<IUpdateTaskMaintenanceProcessor> \_updateTaskMaintenanceProcessorMock;

private TasksController \_controller;

public HttpRequestMessage CreateRequestMessage(HttpMethod method = null, string uriString = null)

{

method = method ?? HttpMethod.Get;

var uri = string.IsNullOrWhiteSpace(uriString)

? new Uri("http://localhost:12345/api/whatever")

: new Uri(uriString);

var requestMessage = new HttpRequestMessage(method, uri);

requestMessage.SetConfiguration(new HttpConfiguration());

return requestMessage;

}

[Test]

public void GetTasks\_returns\_correct\_response()

{

var requestMessage = CreateRequestMessage();

var request = new PagedDataRequest(1, 25);

var response = new PagedDataInquiryResponse<Task>();

\_pagedDataRequestFactoryMock.Setup(x => x.Create(requestMessage.RequestUri)).Returns(request);

\_allTasksInquiryProcessorMock.Setup(x => x.GetTasks(request)).Returns(response);

var actualResponse = \_controller.GetTasks(requestMessage);

Assert.AreSame(response, actualResponse);

}

}

}

Paradoxically, it seems that this required a lot of code, and yet the test itself required very little code. Let's first talk about the test itself. It required very little code because the GetTasks controller method is quite simple. We extolled the virtues of "thin" controller methods and a loosely-coupled architecture in the previous chapter, and now we are experiencing some of the benefits; namely, it makes unit testing easy to do. In the GetTasks\_returns\_correct\_response test method (indicated as a test by the NUnit Test attribute), all we are doing is setting up the mocked dependencies, invoking the target method, and comparing the result with the expected value. Go ahead and build and run the test; the test should pass.

Now let's address the problem of so much apparent "noise" code in this test class. In particular, we're mocking a bunch of dependencies to satisfy the TasksController constructor, and yet we're only using one of them. Well, for one thing, if we were to test all of the the controller methods we would be using all of the dependencies; so, in that sense, they aren't gratuitous. However, from a design perspective this arrangement is suboptimal; we are breaking the Open Closed and Single Responsibility principles. For example, when we implement the DeleteTask action method we will need to modify the TaskController constructor to accept a parameter of type IDeleteTaskQueryProcessor (note, this method is implemented in the example code on the book's github site). We will also need to modify the unit test class to pass an IDeleteTaskQueryProcessor instance to the updated constructor. How do we improve this design?

The good news here is that we've already made the first step towards improving, or refactoring, the code: we've covered the implementation with a test that passes. Now that the implementation is covered, it can be safely modified.

For the next step, let's fold all of these dependencies into a single dependency. Implement as follows:

ITasksControllerDependencyBlock Interface

using WebApi2Book.Web.Api.InquiryProcessing;

using WebApi2Book.Web.Api.MaintenanceProcessing;

namespace WebApi2Book.Web.Api.Controllers.V1

{

public interface ITasksControllerDependencyBlock

{

IAddTaskMaintenanceProcessor AddTaskMaintenanceProcessor { get; }

ITaskByIdInquiryProcessor TaskByIdInquiryProcessor { get; }

IUpdateTaskMaintenanceProcessor UpdateTaskMaintenanceProcessor { get; }

IPagedDataRequestFactory PagedDataRequestFactory { get; }

IAllTasksInquiryProcessor AllTasksInquiryProcessor { get; }

}

}

TasksControllerDependencyBlock Class

using WebApi2Book.Web.Api.InquiryProcessing;

using WebApi2Book.Web.Api.MaintenanceProcessing;

namespace WebApi2Book.Web.Api.Controllers.V1

{

public class TasksControllerDependencyBlock : ITasksControllerDependencyBlock

{

public IAddTaskMaintenanceProcessor AddTaskMaintenanceProcessor { get; private set; }

public ITaskByIdInquiryProcessor TaskByIdInquiryProcessor { get; private set; }

public IUpdateTaskMaintenanceProcessor UpdateTaskMaintenanceProcessor { get; private set; }

public IPagedDataRequestFactory PagedDataRequestFactory { get; private set; }

public IAllTasksInquiryProcessor AllTasksInquiryProcessor { get; private set; }

public TasksControllerDependencyBlock(IAddTaskMaintenanceProcessor addTaskMaintenanceProcessor,

ITaskByIdInquiryProcessor taskByIdInquiryProcessor,

IUpdateTaskMaintenanceProcessor updateTaskMaintenanceProcessor,

IPagedDataRequestFactory pagedDataRequestFactory,

IAllTasksInquiryProcessor allTasksInquiryProcessor)

{

AddTaskMaintenanceProcessor = addTaskMaintenanceProcessor;

TaskByIdInquiryProcessor = taskByIdInquiryProcessor;

UpdateTaskMaintenanceProcessor = updateTaskMaintenanceProcessor;

PagedDataRequestFactory = pagedDataRequestFactory;

AllTasksInquiryProcessor = allTasksInquiryProcessor;

}

}

}

Dependency Configuration (add to NinjectConfigurator.AddBindings)

container.Bind<ITasksControllerDependencyBlock>().To<TasksControllerDependencyBlock>().InRequestScope();

Now we'll update the TasksController implementation to make use of this new ITasksControllerDependencyBlock:

using System.Net.Http;

using System.Web.Http;

using WebApi2Book.Common;

using WebApi2Book.Web.Api.MaintenanceProcessing;

using WebApi2Book.Web.Api.Models;

using WebApi2Book.Web.Common;

using WebApi2Book.Web.Common.Routing;

using WebApi2Book.Web.Common.Validation;

namespace WebApi2Book.Web.Api.Controllers.V1

{

[ApiVersion1RoutePrefix("tasks")]

[UnitOfWorkActionFilter]

[Authorize(Roles = Constants.RoleNames.JuniorWorker)]

public class TasksController : ApiController

{

private readonly ITasksControllerDependencyBlock \_tasksControllerDependencyBlock;

public TasksController(ITasksControllerDependencyBlock tasksControllerDependencyBlock)

{

\_tasksControllerDependencyBlock = tasksControllerDependencyBlock;

}

[Route("", Name = "GetTasksRoute")]

public PagedDataInquiryResponse<Task> GetTasks(HttpRequestMessage requestMessage)

{

var request = \_tasksControllerDependencyBlock.PagedDataRequestFactory.Create(requestMessage.RequestUri);

var tasks = \_tasksControllerDependencyBlock.AllTasksInquiryProcessor.GetTasks(request);

return tasks;

}

etc.…

This is much more "clean". Note how the ITasksControllerDependencyBlock is now the only dependency required by TasksController. Its single responsibility is to encapsulate the other dependencies, so new methods can be added to the TasksController without any modification to its constructor (or, similarly, to the TasksControllerTest SetUp method which is coupled to the TasksController constructor). The TasksControllerDependencyBlock constructor will necessarily change, but that's okay because managing dependencies is its responsility. And, on the testing side, we don't generally test these "dependency blocks", so adding new dependencies to "block" types requires no rework in that regard.

Before we become exhausted from patting ourselves on the back, however, an attempt to build the solution will remind us that we still need to update the TasksControllerTest. Let's tend to that right now by updating it so it appears as follows:

using System.Net.Http;

using NUnit.Framework;

using WebApi2Book.Data;

using WebApi2Book.Web.Api.Controllers.V1;

using WebApi2Book.Web.Api.Models;

namespace WebApi2Book.Web.Api.Tests.Controllers.V1

{

[TestFixture]

public class TasksControllerTest

{

[SetUp]

public void SetUp()

{

\_mockBlock = new TasksControllerDependencyBlockMock();

\_controller = new TasksController(\_mockBlock.Object);

}

private TasksControllerDependencyBlockMock \_mockBlock;

private TasksController \_controller;

[Test]

public void GetTasks\_returns\_correct\_response()

{

HttpRequestMessage requestMessage = HttpRequestMessageFactory.CreateRequestMessage();

var request = new PagedDataRequest(1, 25);

var response = new PagedDataInquiryResponse<Task>();

\_mockBlock.PagedDataRequestFactoryMock.Setup(x => x.Create(requestMessage.RequestUri)).Returns(request);

\_mockBlock.AllTasksInquiryProcessorMock.Setup(x => x.GetTasks(request)).Returns(response);

PagedDataInquiryResponse<Task> actualResponse = \_controller.GetTasks(requestMessage);

Assert.AreSame(response, actualResponse);

}

}

}

Note how we've replaced all of the various mocked TasksController dependencies with a single dependency (i.e., TasksControllerDependencyBlockMock), significantly cutting down on the "noise" in the test class. We took things a step further and factored the CreateRequestMessage method out into a separate test utility class (we trust you to implement that without further explanation), because we anticipate that other test classes will require the functionality provided by the method, and we don't want to in any way encourage a "copy and paste" coding style.

You've probably noticed that we have one more class to add before we can build and re-run the test, and that's the new TasksControllerDependencyBlockMock class. Let's add it as follows:

using Moq;

using WebApi2Book.Web.Api.Controllers.V1;

using WebApi2Book.Web.Api.InquiryProcessing;

using WebApi2Book.Web.Api.MaintenanceProcessing;

namespace WebApi2Book.Web.Api.Tests.Controllers.V1

{

public class TasksControllerDependencyBlockMock : Mock<ITasksControllerDependencyBlock>

{

private Mock<IAddTaskMaintenanceProcessor> \_addTaskMaintenanceProcessorMock;

private Mock<ITaskByIdInquiryProcessor> \_taskByIdInquiryProcessorMock;

private Mock<IUpdateTaskMaintenanceProcessor> \_updateTaskMaintenanceProcessorMock;

private Mock<IPagedDataRequestFactory> \_pagedDataRequestFactoryMock;

private Mock<IAllTasksInquiryProcessor> \_allTasksInquiryProcessorMock;

public TasksControllerDependencyBlockMock()

{

Setup(x => x.AddTaskMaintenanceProcessor).Returns(AddTaskMaintenanceProcessorMock.Object);

Setup(x => x.TaskByIdInquiryProcessor).Returns(TaskByIdInquiryProcessorMock.Object);

Setup(x => x.UpdateTaskMaintenanceProcessor).Returns(UpdateTaskMaintenanceProcessorMock.Object);

Setup(x => x.PagedDataRequestFactory).Returns(PagedDataRequestFactoryMock.Object);

Setup(x => x.AllTasksInquiryProcessor).Returns(AllTasksInquiryProcessorMock.Object);

}

public Mock<IAddTaskMaintenanceProcessor> AddTaskMaintenanceProcessorMock

{

get { return \_addTaskMaintenanceProcessorMock ??

(\_addTaskMaintenanceProcessorMock = new Mock<IAddTaskMaintenanceProcessor>()); }

set { \_addTaskMaintenanceProcessorMock = value; }

}

public Mock<ITaskByIdInquiryProcessor> TaskByIdInquiryProcessorMock

{

get { return \_taskByIdInquiryProcessorMock ??

(\_taskByIdInquiryProcessorMock = new Mock<ITaskByIdInquiryProcessor>()); }

set { \_taskByIdInquiryProcessorMock = value; }

}

public Mock<IUpdateTaskMaintenanceProcessor> UpdateTaskMaintenanceProcessorMock

{

get { return \_updateTaskMaintenanceProcessorMock ??

(\_updateTaskMaintenanceProcessorMock = new Mock<IUpdateTaskMaintenanceProcessor>()); }

set { \_updateTaskMaintenanceProcessorMock = value; }

}

public Mock<IPagedDataRequestFactory> PagedDataRequestFactoryMock

{

get { return \_pagedDataRequestFactoryMock ??

(\_pagedDataRequestFactoryMock = new Mock<IPagedDataRequestFactory>()); }

set { \_pagedDataRequestFactoryMock = value; }

}

public Mock<IAllTasksInquiryProcessor> AllTasksInquiryProcessorMock

{

get { return \_allTasksInquiryProcessorMock ??

(\_allTasksInquiryProcessorMock = new Mock<IAllTasksInquiryProcessor>()); }

set { \_allTasksInquiryProcessorMock = value; }

}

}

}

This simple class is simply a mock of mocks, encapsulating all of the mocked dependencies required by the TasksController. And now, with that in place, we are ready to build and re-run the GetTasks\_returns\_correct\_response test. It should pass. Congratulations, you have tested and safely refactored some code. Go ahead and pat yourself on the back, but make it quick because now we've got to test the dependencies that were used in the GetTasks implementation.

### Testing the Dependencies

The first thing we need to do is add some more dependencies to the WebApi2Book.Web.Api.Tests project. With the solution open, run the following commands in the Package Manager Console to install the logging framework that we introduced in Chapter 3:

install-package log4net WebApi2Book.Web.Api.Tests

Next, reference System.Web and the WebApi2Book.Common project from the WebApi2Book.Web.Api.Tests project. With that complete, we are now ready to test the first dependency that we used in the GetTasks method; namely, PagedDataRequestFactory. Therefore, let's add a new test, PagedDataRequestFactoryTest, as follows:

using System;

using System.Net;

using System.Net.Http;

using System.Web;

using log4net;

using Moq;

using NUnit.Framework;

using WebApi2Book.Common.Logging;

using WebApi2Book.Web.Api.InquiryProcessing;

namespace WebApi2Book.Web.Api.Tests.InquiryProcessing

{

[TestFixture]

public class PagedDataRequestFactoryTest

{

[SetUp]

public void SetUp()

{

\_logMock = new Mock<ILog>();

\_logManagerMock = new Mock<ILogManager>();

\_logManagerMock.Setup(x => x.GetLog(It.IsAny<Type>())).Returns(\_logMock.Object);

\_requestFactory = new PagedDataRequestFactory(\_logManagerMock.Object);

}

private const int DefaultPageNumber = 1;

private const int MaxPageSize = 50;

private const int DefaultPageSize = 25;

private Mock<ILog> \_logMock;

private Mock<ILogManager> \_logManagerMock;

private PagedDataRequestFactory \_requestFactory;

[Test]

public void Create\_throws\_HttpException\_when\_given\_invalid\_query\_string()

{

var requestMessage = HttpRequestMessageFactory.CreateRequestMessage(HttpMethod.Get,

"http://www.foo.com/bar?pageNumber=2&pageSize=10&pageNumber=50");

try

{

\_requestFactory.Create(requestMessage.RequestUri);

Assert.Fail();

}

catch (HttpException e)

{

Assert.AreEqual((int) HttpStatusCode.BadRequest, e.GetHttpCode());

}

}

[Test]

public void Create\_uses\_corrected\_supplied\_pageNumber()

{

const int pageNumber = 0;

var requestMessage = HttpRequestMessageFactory.CreateRequestMessage(HttpMethod.Get,

string.Format("http://www.foo.com/bar?pageNumber={0}", pageNumber));

var inquiryRequestData = \_requestFactory.Create(requestMessage.RequestUri);

Assert.AreEqual(DefaultPageNumber, inquiryRequestData.PageNumber);

}

[Test]

public void Create\_uses\_corrected\_supplied\_pageSize()

{

const int pageSize = 2000;

var requestMessage = HttpRequestMessageFactory.CreateRequestMessage(HttpMethod.Get,

string.Format("http://www.foo.com/bar?pageSize={0}", pageSize));

var inquiryRequestData = \_requestFactory.Create(requestMessage.RequestUri);

Assert.AreEqual(MaxPageSize, inquiryRequestData.PageSize);

}

[Test]

public void Create\_uses\_default\_pageNumber()

{

var requestMessage = HttpRequestMessageFactory.CreateRequestMessage();

var inquiryRequestData = \_requestFactory.Create(requestMessage.RequestUri);

Assert.AreEqual(DefaultPageNumber, inquiryRequestData.PageNumber);

}

[Test]

public void Create\_uses\_default\_pageSize()

{

var requestMessage = HttpRequestMessageFactory.CreateRequestMessage();

var inquiryRequestData = \_requestFactory.Create(requestMessage.RequestUri);

Assert.AreEqual(DefaultPageSize, inquiryRequestData.PageSize);

}

[Test]

public void Create\_uses\_supplied\_pageNumber()

{

const int pageNumber = 1;

var requestMessage = HttpRequestMessageFactory.CreateRequestMessage(HttpMethod.Get,

string.Format(

"http://www.foo.com/bar?pageNumber={0}", pageNumber));

var inquiryRequestData = \_requestFactory.Create(requestMessage.RequestUri);

Assert.AreEqual(pageNumber, inquiryRequestData.PageNumber);

}

[Test]

public void Create\_uses\_supplied\_pageSize()

{

const int pageSize = 20;

var requestMessage = HttpRequestMessageFactory.CreateRequestMessage(HttpMethod.Get,

string.Format(

"http://www.foo.com/bar?pageSize={0}", pageSize));

var inquiryRequestData = \_requestFactory.Create(requestMessage.RequestUri);

Assert.AreEqual(pageSize, inquiryRequestData.PageSize);

}

}

}

If this test class does not appear to be very interesting, consider this: we have achieved virtually one hundred percent code coverage of the PagedDataRequestFactory class. The CreateRequestMessage utility that we introduced in the previous section helped make this relatively easy to accomplish. And, with the TasksController and PagedDataRequestFactory tests now out of the way, we are finished dealing with classes that have any dependency on the ASP.NET Web API; it's all just plain old C# from here on out, which is really nice from a testing (and code reuse) perspective.

The last class that we're going to unit test in this section is AllTasksInquiryProcessor.

Todo - the test

In a production situation you would also need to test all of the other dependencies, including the ones we didn't cover here (e.g., PrimitiveTypeParser, TaskLinkService, the various extension methods). However, for the sake of keeping our readers engaged (are you still with us?), we will now move on to the integration tests. At this point we trust that we've provided the basic foundation, and some helpful techniques, needed to provide unit test coverage to an ASP.NET Web API-based application.

## Integration Testing

Test the query processor and the site.

# Going Live!

TODO - Jamie's section on the UI, CORS, CSRF…

# Summary