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 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 based Single Page 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 automated 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 certainly 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 one 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, if we were to implement the DeleteTask action method we would 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 would 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's public signature (or to the TasksControllerTest SetUp method, which is coupled to the TasksController constructor signature). The TasksControllerDependencyBlock constructor will necessarily change, but that's okay because managing dependencies is its only responsility. And, on the testing side, we don't generally unit 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 abruptly remind us that we still need to update the TasksControllerTest. Let's attend to that right now by updating it so it appears as follows:

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()

{

var 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);

var 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 a controller action method. Go ahead and celebrate this little victory, 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 noteworthy (aside from the fact that it is self-documenting), consider this: with it, 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 ideal from a testing (and code reuse) perspective.

The final class that we're going to unit test in this section is AllTasksInquiryProcessor. The last time we saw this class' implementation was all the way back in Chapter 7, so we'll show it again, below, as a bit of a memory refresher:

using System.Collections.Generic;

using System.Linq;

using WebApi2Book.Common.TypeMapping;

using WebApi2Book.Data;

using WebApi2Book.Web.Api.LinkServices;

using WebApi2Book.Web.Api.Models;

using PagedTaskDataInquiryResponse =

WebApi2Book.Web.Api.Models.PagedDataInquiryResponse<WebApi2Book.Web.Api.Models.Task>;

namespace WebApi2Book.Web.Api.InquiryProcessing

{

public class AllTasksInquiryProcessor : IAllTasksInquiryProcessor

{

public const string QueryStringFormat = "pagenumber={0}&pagesize={1}";

private readonly IAutoMapper \_autoMapper;

private readonly ICommonLinkService \_commonLinkService;

private readonly IAllTasksQueryProcessor \_queryProcessor;

private readonly ITaskLinkService \_taskLinkService;

public AllTasksInquiryProcessor(IAllTasksQueryProcessor queryProcessor, IAutoMapper autoMapper,

ITaskLinkService taskLinkService, ICommonLinkService commonLinkService)

{

\_queryProcessor = queryProcessor;

\_autoMapper = autoMapper;

\_taskLinkService = taskLinkService;

\_commonLinkService = commonLinkService;

}

public PagedTaskDataInquiryResponse GetTasks(PagedDataRequest requestInfo)

{

var queryResult = \_queryProcessor.GetTasks(requestInfo);

var tasks = GetTasks(queryResult.QueriedItems).ToList();

var inquiryResponse = new PagedTaskDataInquiryResponse

{

Items = tasks,

PageCount = queryResult.TotalPageCount,

PageNumber = requestInfo.PageNumber,

PageSize = requestInfo.PageSize

};

AddLinksToInquiryResponse(inquiryResponse);

return inquiryResponse;

}

public virtual void AddLinksToInquiryResponse(PagedTaskDataInquiryResponse inquiryResponse)

{

inquiryResponse.AddLink(\_taskLinkService.GetAllTasksLink());

\_commonLinkService.AddPageLinks(inquiryResponse, GetCurrentPageQueryString(inquiryResponse),

GetPreviousPageQueryString(inquiryResponse),

GetNextPageQueryString(inquiryResponse));

}

public virtual IEnumerable<Task> GetTasks(IEnumerable<Data.Entities.Task> taskEntities)

{

var tasks = taskEntities.Select(x => \_autoMapper.Map<Task>(x)).ToList();

tasks.ForEach(x =>

{

\_taskLinkService.AddSelfLink(x);

\_taskLinkService.AddLinksToChildObjects(x);

});

return tasks;

}

public virtual string GetCurrentPageQueryString(PagedTaskDataInquiryResponse inquiryResponse)

{

return

string.Format(QueryStringFormat,

inquiryResponse.PageNumber,

inquiryResponse.PageSize);

}

public virtual string GetPreviousPageQueryString(PagedTaskDataInquiryResponse inquiryResponse)

{

return

string.Format(QueryStringFormat,

inquiryResponse.PageNumber - 1,

inquiryResponse.PageSize);

}

public virtual string GetNextPageQueryString(PagedTaskDataInquiryResponse inquiryResponse)

{

return string.Format(QueryStringFormat,

inquiryResponse.PageNumber + 1,

inquiryResponse.PageSize);

}

}

}

Now let's implement the test for this class as follows:

using System;

using System.Collections.Generic;

using System.Linq;

using Moq;

using NUnit.Framework;

using WebApi2Book.Common.TypeMapping;

using WebApi2Book.Data;

using WebApi2Book.Web.Api.InquiryProcessing;

using WebApi2Book.Web.Api.LinkServices;

using WebApi2Book.Web.Api.Models;

using PagedTaskDataInquiryResponse =

WebApi2Book.Web.Api.Models.PagedDataInquiryResponse<WebApi2Book.Web.Api.Models.Task>;

using Task = WebApi2Book.Data.Entities.Task;

namespace WebApi2Book.Web.Api.Tests.InquiryProcessing

{

[TestFixture]

public class AllTasksInquiryProcessorTest

{

[SetUp]

public void SetUp()

{

\_autoMapperMock = new Mock<IAutoMapper>();

\_commonLinkServiceMock = new Mock<ICommonLinkService>();

\_allTasksQueryProcessorMock = new Mock<IAllTasksQueryProcessor>();

\_taskLinkServiceMock = new Mock<ITaskLinkService>();

\_inquiryProcessor = new AllTasksInquiryProcessorTestDouble(\_allTasksQueryProcessorMock.Object,

\_autoMapperMock.Object, \_taskLinkServiceMock.Object, \_commonLinkServiceMock.Object);

}

private const string QueryStringFormat = "pagenumber={0}&pagesize={1}";

private Mock<IAutoMapper> \_autoMapperMock;

private Mock<ICommonLinkService> \_commonLinkServiceMock;

private Mock<IAllTasksQueryProcessor> \_allTasksQueryProcessorMock;

private Mock<ITaskLinkService> \_taskLinkServiceMock;

private AllTasksInquiryProcessorTestDouble \_inquiryProcessor;

private const int PageNumber = 1;

private const int PageSize = 20;

private class AllTasksInquiryProcessorTestDouble : AllTasksInquiryProcessor

{

public AllTasksInquiryProcessorTestDouble(IAllTasksQueryProcessor queryProcessor, IAutoMapper autoMapper,

ITaskLinkService taskLinkService, ICommonLinkService commonLinkService)

: base(queryProcessor, autoMapper, taskLinkService, commonLinkService)

{

}

public Func<IEnumerable<Task>, IEnumerable<Models.Task>> GetTasksTestDouble { get; set; }

public Action<PagedTaskDataInquiryResponse> AddLinksToInquiryResponseTestDouble { get; set; }

public Func<PagedTaskDataInquiryResponse, string> GetCurrentPageQueryStringTestDouble { get; set; }

public Func<PagedTaskDataInquiryResponse, string> GetNextPageQueryStringTestDouble { get; set; }

public Func<PagedTaskDataInquiryResponse, string> GetPreviousPageQueryStringTestDouble { get; set; }

public override IEnumerable<Models.Task> GetTasks(IEnumerable<Task> taskEntities)

{

return GetTasksTestDouble == null ? base.GetTasks(taskEntities) : GetTasksTestDouble(taskEntities);

}

public override void AddLinksToInquiryResponse(PagedTaskDataInquiryResponse inquiryResponse)

{

if (AddLinksToInquiryResponseTestDouble == null)

{

base.AddLinksToInquiryResponse(inquiryResponse);

}

else

{

AddLinksToInquiryResponseTestDouble(inquiryResponse);

}

}

public override string GetCurrentPageQueryString(PagedTaskDataInquiryResponse inquiryResponse)

{

return GetCurrentPageQueryStringTestDouble == null

? base.GetCurrentPageQueryString(inquiryResponse)

: GetCurrentPageQueryStringTestDouble(inquiryResponse);

}

public override string GetNextPageQueryString(PagedTaskDataInquiryResponse inquiryResponse)

{

return GetNextPageQueryStringTestDouble == null

? base.GetNextPageQueryString(inquiryResponse)

: GetNextPageQueryStringTestDouble(inquiryResponse);

}

public override string GetPreviousPageQueryString(PagedTaskDataInquiryResponse inquiryResponse)

{

return GetPreviousPageQueryStringTestDouble == null

? base.GetPreviousPageQueryString(inquiryResponse)

: GetPreviousPageQueryStringTestDouble(inquiryResponse);

}

}

[Test]

public void AddLinksToInquiryResponse\_adds\_AllTasks\_link()

{

var link = new Link();

var inquiryResponse = new PagedTaskDataInquiryResponse();

\_taskLinkServiceMock.Setup(x => x.GetAllTasksLink()).Returns(link);

\_inquiryProcessor.AddLinksToInquiryResponse(inquiryResponse);

Assert.AreSame(link, inquiryResponse.Links.Single());

}

[Test]

public void AddLinksToInquiryResponse\_adds\_page\_links()

{

var inquiryResponse = new PagedTaskDataInquiryResponse();

const string currentPageQueryString = "current";

const string previousPageQueryString = "previous";

const string nextPageQueryString = "next";

\_inquiryProcessor.GetCurrentPageQueryStringTestDouble = response => currentPageQueryString;

\_inquiryProcessor.GetPreviousPageQueryStringTestDouble = response => previousPageQueryString;

\_inquiryProcessor.GetNextPageQueryStringTestDouble = response => nextPageQueryString;

\_inquiryProcessor.AddLinksToInquiryResponse(inquiryResponse);

\_commonLinkServiceMock.Verify(

x =>

x.AddPageLinks(inquiryResponse, currentPageQueryString, previousPageQueryString, nextPageQueryString));

}

[Test]

public void GetCurrentPageQueryString\_returns\_correct\_value()

{

var expectedResult = string.Format(QueryStringFormat, PageNumber, PageSize);

var inquiryResponse = new PagedTaskDataInquiryResponse {PageNumber = PageNumber, PageSize = PageSize};

var actualResult = \_inquiryProcessor.GetCurrentPageQueryString(inquiryResponse);

Assert.AreEqual(expectedResult, actualResult);

}

[Test]

public void GetNextPageQueryString\_returns\_correct\_value()

{

var expectedResult = string.Format(QueryStringFormat, PageNumber + 1, PageSize);

var inquiryResponse = new PagedTaskDataInquiryResponse {PageNumber = PageNumber, PageSize = PageSize};

var actualResult = \_inquiryProcessor.GetNextPageQueryString(inquiryResponse);

Assert.AreEqual(expectedResult, actualResult);

}

[Test]

public void GetPreviousPageQueryString\_returns\_correct\_value()

{

var expectedResult = string.Format(QueryStringFormat, PageNumber - 1, PageSize);

var inquiryResponse = new PagedTaskDataInquiryResponse {PageNumber = PageNumber, PageSize = PageSize};

var actualResult = \_inquiryProcessor.GetPreviousPageQueryString(inquiryResponse);

Assert.AreEqual(expectedResult, actualResult);

}

[Test]

public void GetTasks\_adds\_child\_links()

{

var taskEntity1 = new Task {TaskId = 300};

var taskEntity2 = new Task {TaskId = 600};

var task1 = new Models.Task {TaskId = taskEntity1.TaskId};

var task2 = new Models.Task {TaskId = taskEntity2.TaskId};

var taskEntities = new List<Task> {taskEntity1, taskEntity2};

var tasks = new List<Models.Task> {task1, task2};

for (var i = 0; i < taskEntities.Count; ++i)

{

var index = i;

\_autoMapperMock.Setup(x => x.Map<Models.Task>(taskEntities[index])).Returns(tasks[index]);

\_taskLinkServiceMock.Setup(x => x.AddLinksToChildObjects(tasks[index])).Verifiable();

}

\_inquiryProcessor.GetTasks(taskEntities);

\_taskLinkServiceMock.VerifyAll();

}

[Test]

public void GetTasks\_adds\_links()

{

var requestInfo = new PagedDataRequest(PageNumber, PageSize);

var taskEntity = new Task {TaskId = 300};

var queriedItems = new[] {taskEntity};

var queryResult = new QueryResult<Task>(queriedItems, queriedItems.Count(), PageSize);

var task = new Models.Task {TaskId = taskEntity.TaskId};

var tasks = new[] {task};

\_allTasksQueryProcessorMock.Setup(x => x.GetTasks(requestInfo)).Returns(queryResult);

\_inquiryProcessor.GetTasksTestDouble = items => items == queriedItems ? tasks : null;

var linksWereAdded = false;

\_inquiryProcessor.AddLinksToInquiryResponseTestDouble =

response => linksWereAdded = tasks.SequenceEqual(response.Items)

&&

response.PageCount ==

queryResult.TotalPageCount

&& response.PageNumber == PageNumber

&& response.PageSize == PageSize;

\_inquiryProcessor.GetTasks(requestInfo);

Assert.IsTrue(linksWereAdded);

}

[Test]

public void GetTasks\_adds\_self\_link\_to\_tasks()

{

var taskEntity1 = new Task {TaskId = 300};

var taskEntity2 = new Task {TaskId = 600};

var task1 = new Models.Task {TaskId = taskEntity1.TaskId};

var task2 = new Models.Task {TaskId = taskEntity2.TaskId};

var taskEntities = new List<Task> {taskEntity1, taskEntity2};

var tasks = new List<Models.Task> {task1, task2};

for (var i = 0; i < taskEntities.Count; ++i)

{

var index = i;

\_autoMapperMock.Setup(x => x.Map<Models.Task>(taskEntities[index])).Returns(tasks[index]);

\_taskLinkServiceMock.Setup(x => x.AddSelfLink(tasks[index])).Verifiable();

}

\_inquiryProcessor.GetTasks(taskEntities);

\_taskLinkServiceMock.VerifyAll();

}

[Test]

public void GetTasks\_maps\_entities\_to\_web\_models()

{

var taskEntity1 = new Task {TaskId = 300};

var taskEntity2 = new Task {TaskId = 600};

var task1 = new Models.Task {TaskId = taskEntity1.TaskId};

var task2 = new Models.Task {TaskId = taskEntity2.TaskId};

var taskEntities = new List<Task> {taskEntity1, taskEntity2};

var tasks = new List<Models.Task> {task1, task2};

for (var i = 0; i < taskEntities.Count; ++i)

{

var index = i;

\_autoMapperMock.Setup(x => x.Map<Models.Task>(taskEntities[index])).Returns(tasks[index]);

}

var actualResult = \_inquiryProcessor.GetTasks(taskEntities);

Assert.IsTrue(tasks.SequenceEqual(actualResult));

}

[Test]

public void GetTasks\_returns\_correct\_result()

{

var requestInfo = new PagedDataRequest(PageNumber, PageSize);

var taskEntity = new Task {TaskId = 300};

var queriedItems = new[] {taskEntity};

var queryResult = new QueryResult<Task>(queriedItems, queriedItems.Count(), PageSize);

var task = new Models.Task {TaskId = taskEntity.TaskId};

var tasks = new[] {task};

\_allTasksQueryProcessorMock.Setup(x => x.GetTasks(requestInfo)).Returns(queryResult);

\_inquiryProcessor.GetTasksTestDouble = items => items == queriedItems ? tasks : null;

var actualResult = \_inquiryProcessor.GetTasks(requestInfo);

Assert.IsTrue(tasks.SequenceEqual(actualResult.Items), "Incorrect Items in result");

Assert.AreEqual(queryResult.TotalPageCount, actualResult.PageCount, "Incorrect PageCount in result");

Assert.AreEqual(PageNumber, actualResult.PageNumber, "Incorrect PageNumber in result");

Assert.AreEqual(PageSize, actualResult.PageSize, "Incorrect PageSize in result");

}

}

}

There are several notable items in this test class, but first let's mention some things about the AllTasksInquiryProcessor class itself. For one, all of the helper (i.e., non-interface) methods on the AllTasksInquiryProcessor class are virtual. This allows us to stub those out when unit testing.

Secondly, notice that although AllTasksInquiryProcessor requires several dependencies, we do not aggregate the dependencies in a "block" as we did with the TasksController. This is because, unlike the TasksController, we don't expect the AllTasksInquiryProcessor to take on additional responsibilities. This decision should be revisited if the underlying assumption were someday proven to be incorrect.

As for the AllTasksInquiryProcessorTest class, here are some of the more noteworthy elements:

* The class contains a private class, AllTasksInquiryProcessorTestDouble, which derives from AllTasksInquiryProcessor. The AllTasksInquiryProcessorTest test methods use an instance of this private class, stubbing out functionality to allow individual test methods to isolate a single method under test.
* The class employs a mix of state verification and behavior verification, as appropriate, in order to provide high levels of code coverage. For example, AddLinksToInquiryResponse\_adds\_AllTasks\_link employs state verification; the response is examined. On the other hand, AddLinksToInquiryResponse\_adds\_page\_links employs behavior verification; the ICommonLinkService mock is interrogated to verify that the AddPageLinks method was invoked.

Martin Fowler provides a good explanation of these approaches to testing in the following article: http://martinfowler.com/articles/mocksArentStubs.html#ClassicalAndMockistTesting.

* The class defines a QueryStringFormat field, rather than relying on the AllTasksInquiryProcessor's public QueryStringFormat field. This is so that tests will break if the AllTasksInquiryProcessor's field were to ever change. Yes, this is actually desirable! If the functionality changes, then the corresponding test(s) should break.

In a production situation you would also need to test all of the other dependencies, including the ones we didn't cover here (i.e., PrimitiveTypeParser, TaskLinkService, the various extension methods, etc.). However, for the sake of keeping our readers engaged (you are still with us, right?), 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, necessary for you to apply unit test coverage to an ASP.NET Web API-based application.

## Integration Testing

The first thing we need to do is add some dependencies to the WebApi2Book.Web.Api.IntegrationTests 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.IntegrationTests

install-package Moq WebApi2Book.Web.Api.IntegrationTests

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.IntegrationTests

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

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

WebApi2Book.Common

WebApi2Book.Web.Api

WebApi2Book.Web.Api.Models

Let's start off by adding a utility class to help the tests consume the task-management service (implement as follows):

using System;

using System.Net;

using System.Text;

using WebApi2Book.Common;

namespace WebApi2Book.Web.Api.IntegrationTests

{

public class WebClientHelper

{

public WebClient CreateWebClient(string username = "bhogg",

string contentType = Constants.MediaTypeNames.TextJson)

{

var webClient = new WebClient();

var creds = username + ":" + "ignored";

var bcreds = Encoding.ASCII.GetBytes(creds);

var base64Creds = Convert.ToBase64String(bcreds);

webClient.Headers.Add("Authorization", "Basic " + base64Creds);

webClient.Headers.Add("Content-Type", contentType);

return webClient;

}

}

}

By default, the helper method uses the highly-privilaged "bhogg" credentials for happy-path scenarios. The username, and content type, may be specified by on a per-test basis, as needed.

Now let's add the integration test (implement as follows):

using Newtonsoft.Json.Linq;

using NUnit.Framework;

using WebApi2Book.Web.Api.Models;

namespace WebApi2Book.Web.Api.IntegrationTests

{

[TestFixture]

public class TasksControllerTest

{

[SetUp]

public void Setup()

{

\_webClientHelper = new WebClientHelper();

}

private const string UriRoot = "http://localhost:61589/api/v1/";

private WebClientHelper \_webClientHelper;

[Test]

public void GetTasks()

{

var client = \_webClientHelper.CreateWebClient();

try

{

const string address = UriRoot + "tasks";

var responseString = client.DownloadString(address);

var jsonResponse = JObject.Parse(responseString);

Assert.IsNotNull(jsonResponse.ToObject<PagedDataInquiryResponse<Task>>());

}

finally

{

client.Dispose();

}

}

}

}

This simple test creates a WebClient object using our new WebClientHelper. We went ahead and accepted the defaults for this happy-path scenario. The test simply verifies that the response is structurally valid. Other possible scenarios include passing in a bogus or restricted username to test security, or specifying a different content type to test content negotiation.

Now let's execute the test to make sure everything works properly. With the WebApi2Book.Web.Api application running, go ahead and run the test. It should pass. Excellent work, you've successfully implemented end-to-end integration testing!

Of course, as we alluded to a moment ago, other integration tests could, and should, be written for a production situation. And not just integration tests of the service; the integration test portfolio should include tests of the query processors that interact with the database (e.g., AllTasksQueryProcessor). However, at this point we trust that we've provided the basic foundation, and some helpful techniques, necessary for you to apply integration test coverage to an ASP.NET Web API-based application.

Now, by popular demand, let's wrap things up by consuming our task-management service with a simple Single Page Application…

# Going Live!

Well, now that we're at the end of the book, let's examine a JavaScript-based consumer of our Task Management service. You may have noticed a project in the WebApi2Book solution called WebApi2BookSPA. This project is a very simple Single Page Application (SPA) built with the following technologies:

* ASP.NET MVC Framework
* Twitter Bootstrap
* KnockoutJS
* jQuery
* jQuery.Cookie
* Microsoft's JSON Web Token NuGet package

The application leverages ASP.NET MVC to serve up the login and TasksViewer pages. We use a standard MVC controller (and corresponding action) to receive the login POST request (from the form submission), which simply generates a JSON Web Token to be returned to the browser as a response cookie.

As you'll see shortly, the application doesn't actually validate the credentials. We've already covered both Basic and JWT-based authentication back in Chapter 6. At this point, we merely want to demonstrate a jQuery AJAX call into our Task Management service - protected with a valid JWT. So rather than worry about authentication, or relying on a real token issuer, our SPA will generate its own token. And to simplify even more, the login page (as shown in Figure 9-1) will only accept a username and role - which will be used to create the signed JWT.

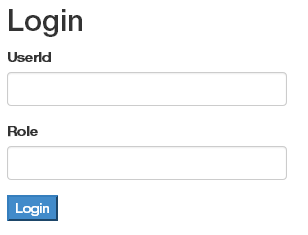


Figure 9-1. Single Page Application's login page

The application's login page allows the user to enter the desired username and role to use when making the REST call to the Task Management service.

We're not going to walk through building the single page application step-by-step. There are plenty of really good web development resources out there to help you create SPAs in ASP.NET. But, if you're curious, you can find the complete code for the examples shown below in the book's Github repository. Throughout the rest of this chapter, we'll be pointing out the important parts from that code. So let's get started!

## Logging In

Per the standard ASP.NET MVC pattern, we need to start with a model which will be used to submit our login form's data:

namespace WebApi2BookSPA.Models

{

public class LoginModel

{

public string UserId { get; set; }

public string Role { get; set; }

}

}

And we need a view:

@model WebApi2BookSPA.Models.LoginModel

<h2>Login</h2>

@using (@Html.BeginForm("Login", "Login", FormMethod.Post, new {role = "form"}))

{

<div class="form-group">

@Html.LabelFor(m => m.UserId)

@Html.TextBoxFor(m => m.UserId, new { @class = "form-control" })

</div>

<div class="form-group">

@Html.LabelFor(m => m.Role)

@Html.TextBoxFor(m => m.Role, new {@class = "form-control"})

</div>

<button type="submit" class="btn-primary">Login</button>

}

In case you haven't ever used Twitter Bootstrap, the various CSS classes specified in the view are included in Bootstrap, and make it extremely easy to create good-looking web pages. Figure 9-1 is the resulting page.

And here's the controller with all that is needed to show out login page:

using System.Web.Mvc;

namespace WebApi2BookSPA.Controllers

{

public class LoginController : Controller

{

public ActionResult Index()

{

return View("LoginView");

}

}

}

Nothing terribly exciting yet, of course. So now let's examine the controller action that accepts the login form's POST request, thereby creating and returning a JWT. Note that the key, issuer, and audience values for the created JWT must match that of the Task Management service's JWT authentication handler. Normally, if using a real token issuer, the symmertic signing key would be agreed upon and shared between parties, as would the issuer value. In our case, since we built the Task Management service and configured the JwtAuthForWebApi package via web.config, we're simply going to copy the key, audience, and issuer into the LoginController.

The resulting code is actually very simple, and similar to the CreateJwt console app we created back in Chatper 6. Here's the LoginController now with the required JWT-creation code:

using System;

using System.IdentityModel.Tokens;

using System.Security.Claims;

using System.Web;

using System.Web.Mvc;

using WebApi2BookSPA.Models;

namespace WebApi2BookSPA.Controllers

{

public class LoginController : Controller

{

public const string SymmetricKey = "cXdlcnR5dWlvcGFzZGZnaGprbHp4Y3Zibm0xMjM0NTY=";

public const string Issuer = "corp";

public const string Audience = "http://www.example.com";

public ActionResult Index()

{

return View("LoginView");

}

[HttpPost]

public ActionResult Login(LoginModel model)

{

SetAuthCookie(model.UserId, model.Role);

return RedirectToAction("Index", "Tasks");

}

private void SetAuthCookie(string userId, string role)

{

var token = CreateJwt(userId, role);

var cookie = new HttpCookie("UserToken", token) {HttpOnly = false};

Response.SetCookie(cookie);

}

private string CreateJwt(string userId, string role)

{

var key = Convert.FromBase64String(SymmetricKey);

var credentials = new SigningCredentials(

new InMemorySymmetricSecurityKey(key),

"http://www.w3.org/2001/04/xmldsig-more#hmac-sha256",

"http://www.w3.org/2001/04/xmlenc#sha256");

var expiration = DateTime.UtcNow.AddMinutes(20).ToLongTimeString();

var tokenDescriptor = new SecurityTokenDescriptor

{

Subject = new ClaimsIdentity(new[]

{

new Claim(ClaimTypes.Name, userId),

new Claim(ClaimTypes.Role, role),

new Claim("exp", expiration)

}),

TokenIssuerName = Issuer,

AppliesToAddress = Audience,

SigningCredentials = credentials

};

var tokenHandler = new JwtSecurityTokenHandler();

var token = tokenHandler.CreateToken(tokenDescriptor);

var tokenString = tokenHandler.WriteToken(token);

return tokenString;

}

}

}

Note the three constants that are copies of the corresponding values used in Chapter 6. All three must match; if not, the JWT validation will fail when we make the AJAX request to the service. The audience is typically the intended target - i.e., the resource server to which the user is needing access. And the issuer identifies (via any string value) the issuer of the token. This helps establish the trust relationship between the resource server (in this case, our Task Management service) and the issuer of the JWT - i.e., the service doesn't want to accept tokens from just any issuer.

Looking at the contoller code, the first thing that happens when the LogIn action is invoked is we create the JWT using the supplied username and role. Even though the act of creating a JWT would typically be handled by a token issuer of some kind, the resulting JWT is the exact same. As discussed previously, it is just a signed collection of claims.

Once the JWT is created - using the predetermined key, audience, and issuer - we create an HttpCookie in which to store the JWT for the client. In this way, the token will be readily available for use by the Tasks Viewer page's jQuery AJAX call to the Task Management service.

Then finally, we redirect the browser to the Tasks view.This next block of code shows the TasksView view:

<h2>Tasks Viewer</h2>

<p>&nbsp;</p>

<p>

<button class="btn btn-primary"

data-bind="click: refreshTasks">Refresh task list</button>

</p>

<p>

<pre data-bind="text: statusMessage"

style="width: 1000px; height: 400px; overflow-x: scroll"></pre>

</p>

<script>

$(function () {

var viewModel = new indexViewModel();

ko.applyBindings(viewModel);

});

</script>

In this view we are leverging Bootstrap to create a reasonable layout that includes a button and a big <pre> text area. We're not going to show the tasks in a grid or tree view at this point. The goal here is just to show the API-level interation with the Task Management service.

The last block in that view is some JavaScript to load our view model, and then use KnockoutJS to bind to the view. Looking at the code above, there are two bindings into our view model: the button click to fetch the tasks, and the returned text being showed in the <pre> element.

Now let's examine the indexViewModel code:

var taskManagementUrl = "http://localhost:52975/api/V1";

var indexViewModel = function() {

var self = this;

self.statusMessage = ko.observable("(Click Refresh button to load tasks)");

self.refreshTasks = function() {

var token = $.cookie('UserToken');

$.ajax({

type: 'GET',

url: taskManagementUrl + '/tasks',

headers: {

Authorization : "Bearer " + token

},

contentType: 'application/json;charset=utf8',

success: self.onRefreshSuccess,

error: self.onRefreshError

});

};

self.onRefreshSuccess = function(data, status) {

self.statusMessage(JSON.stringify(data, null, 4));

};

self.onRefreshError = function(error) {

self.statusMessage(error.responseText);

};

};

The indexViewModel class includes a single property for displaying the returned JSON string, and a function to respond to the referesh button click. The main point here is that we're pulling the JWT from the browser's cookie collection (using the jQuery.cookie library), and then adding it to the AJAX request as a Bearer token in the Authorization header. It could hardly be simpler!

Before we move on, just a quick note… with earlier versions of jQuery, you may need to use the beforeSend event - instead of the newer headers property - to set the Authorization header.

## Support for CORS

If you were to run this web site now, logging in and clicking the Refresh button to fetch some tasks from the Task Management service, the call would fail. As many web developers have experienced, the browser will prevent AJAX calls being made to origins - i.e., servers - that are not the same as from where the current page was loaded. There are several security vulnerabilities associated with making cross-origin requests like this, and so most (if not all) modern browsers prevent such calls from being made.

Instead, a standard known as Cross Origin Resource Sharing (CORS), which is meant to allow for the types of cross-origin AJAX calls we're trying to making here, specifies that before making the desired HTTP request, the browser must first submit an OPTIONS request that essentially askes permission from the server to make the cross-origin request. Cracking open the developer tools in Chrome or Firefox while hitting the Refresh button referenced above, you will see two new header values added to the OPTIONS request related to CORS:

Access-Control-Request-Headers: accept, authorization, content-type

Access-Control-Request-Method: GET

These two header values let the server know that the browser intends to make a cross-origin request that includes, among other things, an authorization header. It's the server's responsibility, under the CORS standard, to respond with the origin(s), headers, and methods the server is willing to accept. The browser is then correspondingly responsible to respect these response headers, and not send the actual GET or POST request if the response headers don't indicate the server would allow it.

So, let's enable CORS support on our Task Management service. With the WebApi2Book solution open, add a couple new NuGet packages with the following commands:

Install-Package Microsoft.AspNet.WebApi.Cors -Pre WebApi2Book.Web.Api

Install-Package Microsoft.AspNet.WebApi.WebHost -Pre WebApi2Book.Web.Common

Install-Package Microsoft.AspNet.WebApi.WebHost -Pre WebApi2Book.Web.Api

At the time of this writing, CORS support in ASP.NET Web API required the use of an alpha release of both the Microsoft.AspNet.WebApi.WebHost and Microsoft.AspNet.WebApi.Cors packages. Once those packages have been installed, add the following line of code to the WebApiConfig class's Register method:

config.EnableCors();

Then, on the TasksController class, add the following attribute:

[EnableCors("http://localhost:52976", "\*", "\*")]

The EnableCors attribute (along with the config.EnableCors() call) is all that is needed to add CORS support to your ASP.NET Web API service. Once configured as such, your service will appropriately respond to the browser's OPTIONS request discussed above.

Now recompile the solution, start up both the WebApi2Book.Web.Api and WebApi2BookSPA projects, and browse to the <http://localhost:52976/login>. This time, make sure you enter JuniorWorker for the role (as shown in Figure 9-2). If you recall from previous chapters, this role is required when fetching tasks from the Task Management service.

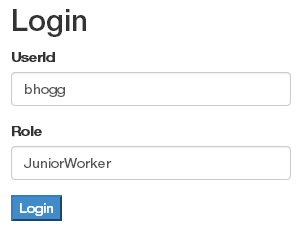


Figure 9-2. Specifying the JuniorWorker role.

Once logged in, click the Refresh button again. This time, you should see the following following headers included in the server's response for the OPTIONS request:

Access-Control-Allow-Headers: authorization,content-type

Access-Control-Allow-Origin: http://localhost:52976

This lets the browser know that it can go ahead and make the desired GET request from the <http://localhost:52976> origin, and can include the Authorization header. The next request seen in the developer tools Network tab is the GET request, and it will include the JWT in the Authorization header. Finally, you should see the currently configured tasks displayed in the pre element box.

Try this now: change the EnableCors attribute to specify a different origin - e.g. <http://www.example.com>. Recompile and rerun the two projects, and try the Refresh button again. This time, the call will fail, and will see a message similar to the following in the Developer Tools Console tab:

XMLHttpRequest cannot load <http://localhost:52975/api/V1/tasks>. No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin '[http://localhost:52976](http://localhost/)' is therefore not allowed access.

This is exactly what we'd expect, in that the server isn't going to allow a request from any origin that isn't http://www.example.com. Interestingly, per the CORS standard, it is the browser that is throwing this exception - not the server. Again, the browser is responsible for respecting the CORS-related attributes, and so it is throwing the exception to prevent your JavaScript code from proceeding with the AJAX GET request - that will include the JWT.

In short, CORS support is very easy to add to your ASP.NET Web API service, and will allow it to be called from more than just your own server. That is the point of Cross Origin Resource Sharing - i.e., to share resources between systems.

# Summary

In this chapter we covered a representative portion of our task-management service with automated unit and integration tests. By doing so, we trust that we've provided the basic foundation, and some helpful techniques, necessary for you to apply automated unit and integration test coverage to your own ASP.NET Web API-based applications. The ASP.NET Web API team has developed, and is continuing to develop, the framework with testing in mind, and we heartily recommend that you take advantage of their efforts.

We also completed the picture, so to speak, by demonstrating how to consume our task-management service using a simple Single Page Application. Calling an ASP.NET Web API service is very easy with today's JavaScript libraries, as we showed with the use of KnockoutJS. We also showed how easy it is to enable your service to be called by other-than-your server, via the CORS standard and the recent addition of CORS support to the ASP.NET Web API.

This brings us to the end of our present journey. We've demonstrated how to leverage the features and capabilities of the ASP.NET Web API to build a RESTful web service from start to finish. We first explained the REST architectural style, and then built on that knowledge, the ASP.NET Web API, and some helpful patterns, tricks, and techniques, to go from a blank slate to a fully functional, secure, and tested RESTful service.

ASP.NET Web API will continue to evolve, so the end of this book is not the end of the overall story. It will be interesting to see what exciting new capabilities will become available if the Open Web Interface for .NET (OWIN) and project Katana (which is built on OWIN) become mainstream in the Web API development community. As an example, at that point, authentication filters, not message handlers, will be the preferred approach to authentication. You will no longer need to write your own authentiation mechanisms for standard authentication schemes, because all of the standard authentiation filters will be built into Katana. You can read about OWIN and Katana here: <http://www.asp.net/aspnet/overview/owin-and-katana/an-overview-of-project-katana>. And, of course, be sure to periodically visit Microsoft's official ASP.NET Web API site to stay abreast of the latest features and releases.

With that, we bid you a fond farewell, at least for now. It's time for you to get started on your next ASP.NET Web API project! May our paths meet again someday!