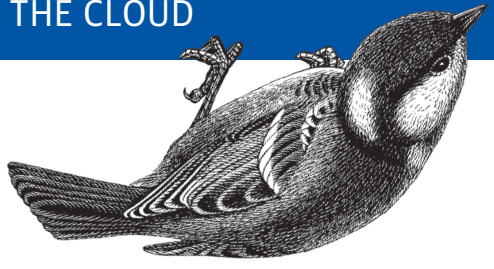
O'REILLY®

Compliments of Pivotal®

FREE CHAPTERS

Building Microservices with ASP.NET Core

DEVELOP, TEST, AND DEPLOY CROSS-PLATFORM SERVICES IN THE CLOUD



Kevin Hoffman

O'REILLY®

Building Microservices with ASP.NET Core

At a time when nearly every vertical, regardless of domain, seems to need software running in the cloud to make money, microservices provide the agility and drastically reduced time to market you require. This hands-on guide shows you how to create, test, compile, and deploy microservices, using the free and open source ASP.NET Core framework. Along the way, you'll pick up good, practical habits for building powerful and robust services.

Building microservices isn't about learning a specific framework or programming language; it's about building applications that thrive in elastically scaling environments that do not have host affinity, and that can start and stop at a moment's notice. This practical book guides you through the process.

- Learn test-driven and API-first development concepts
- Communicate with other services by creating and consuming backing services such as databases and queues
- Build a microservice that depends on an external data source
- Learn about event sourcing, the event-centric approach to persistence
- Use ASP.NET Core to build web applications designed to thrive in the cloud
- Build a service that consumes, or is consumed by, other services
- Create services and applications that accept external configuration
- Explore ways to secure ASP.NET Core microservices and applications

Kevin Hoffman teaches customers how to migrate and modernize their enterprise applications to thrive in the cloud with the latest cloud native patterns, practices, and technology. He's written more than a dozen books on computer programming and has presented at several user groups and conferences.

"ASP.NET Core brings a host of modern practices to developers, and Kevin Hoffman has written a thoughtful, practical guide. Microservices reflect a legitimate way that smart companies build more adaptable software, but there are plenty of new considerations, and Kevin does a masterful job explaining the techniques you need to be successful. Kevin covers more than code; he looks at the lifecycle of building microservices and how to implement the patterns that matter."

—Richard Seroter
Senior Director of Product at Pivotal, Microsoft MVP

US $59.99 CAN $79.99
ISBN: 978-1-491-96173-5

Twitter: @oreillymedia
facebook.com/oreilly

Building .NET microservices?
Steeltoe makes it easy.

Steeltoe is an open-source library that brings Netflix-style microservices patterns to .NET Framework and .NET Core applications.

Built by Pivotal, Steeltoe helps you build **resilient**, **scalable** microservices that run in the cloud, or on-premises.

Read the docs, add to your project via NuGet, contribute open source code, or ask a question in Slack.

Docs: steeltoe.io/docs

NuGet: nuget.org/profiles/steeltoe

GitHub: github.com/steeltoeoss

Slack: slack.steeltoe.io

AVAILABLE TODAY

- ✓ Use remote config store
- ✓ Register services via Netflix Eureka
- ✓ Discover services via Netflix Eureka
- ✓ Connect to MySQL
- ✓ Connect to PostgreSQL
- ✓ Connect to Redis
- ✓ Connect to RabbitMQ
- ✓ Secure apps with OAuth 2.0
- ✓ Integrate with Cloud Foundry

Steeltoe.io

Building Microservices with ASP.NET Core

Develop, Test, and Deploy Cross-Platform Services in the Cloud

This excerpt contains Chapters 3, 8, 9, and 10 of *Building Microservices with ASP.NET Core.* The complete book is available at oreilly.com and through other retailers.

Kevin Hoffman

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Building Microservices with ASP.NET Core
by Kevin Hoffman

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Nan Barber and Brian Foster
Production Editor: Shiny Kalapurakkel
Copyeditor: Kim Cofer
Proofreader: Rachel Head
Indexer: Wendy Catalano
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

September 2017: First Edition

Revision History for the First Edition
2017-08-31: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491961735* for release details.

978-1-491-96173-5

[LSI]

Table of Contents

Preface

The handwriting is on the wall—most people building software and services today are rushing to embrace microservices and their benefits in terms of scale, fault tolerance, and time to market.

This isn't just because it's a shiny new fad. The momentum behind microservices and the concepts driving them is far more important, and those looking for the pendulum to swing back away from the notion of smaller, independently deployed modules will be left behind.

Today, we need to be able to build resilient, elastically scalable applications, and we need to do it rapidly to satisfy the needs of our customers and to keep ahead of our competition.

What You'll Build

Unlike other more reference-style books that are all about showing you each and every API, library, and syntax pattern available to you in a given language, this book is written and meant to be consumed as a guide to building services, with ASP.NET Core simply being the framework in which all the code samples are built.

This book will not teach you every single nuance of low-level C# code; there are far thicker books written by other people if that's what you're looking for. My goal is that by the end of the book, creating, testing, compiling, and deploying microservices in ASP.NET Core will be *muscle memory* for you. You'll develop good, practical habits that will help you rapidly build stable, secure, reliable services.

The mentality I'd like you to have is that after reading this book, you'll have learned a lot about how to build services that are going to be deployed in elastically scalable, high-performance cloud environments. ASP.NET Core in C# is *just one of many* languages and frameworks you can use to build services, but the language does not make the service—*you* do. The care, discipline, and diligence you put into building your

services is far more a predictor of their success in production than any one language or tool ever could be.

The paintbrushes and canvas do not make the painting, the painter does. You are a painter of services, and ASP.NET Core is just one brush among many.

In this book, you'll start with the basic building blocks of any service, and then learn how to turn them into more powerful and robust services. You'll connect to databases and other backing services, and use lightweight distributed caches, secure services, and web apps, all while keeping an eye on the ability to continuously deliver immutable release artifacts in the form of Docker images.

Why You're Building Services

Different teams work on different release cadences with different requirements, motivations, and measures of success. Gone are the days of building monoliths that require a custom, handcrafted, artisanal server in order to run properly. Hopefully, gone as well are the days of gathering a hundred people in conference rooms and on dial-in lines to hope and pray for the successful release of a product at 12:01 on a Sunday morning.

Microservices, if done *properly*, can give us the agility and drastically reduced time to market that our companies need in order to survive and thrive in this new world where nearly every vertical, regardless of its domain, seems to need software running in the cloud to make money.

As you progress through the book you'll see the rationalizations for each decision made. From the individual lines of code to the high-level architectural "napkin drawings," I'll discuss the pros and cons of each choice.

What You'll Need to Build Services

First and foremost, you'll need the .NET Core command-line utilities and the appropriate software development kit (SDK) installed. In the first chapter I'll walk you through what you'll need to get that set up.

Next, you're going to need *Docker*. Docker and the container technology that supports it are ubiquitous these days. Regardless of whether you're deploying to Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP), or your own infrastructure, Docker provides the portable and immutable release artifacts that you crave (and I'll get more into the details of why this is the case throughout the book).

The development and build pipeline for the services in this book is the creation of Docker images running on Linux infrastructure in the cloud. As such, the path of least friction for readers of this book is likely a Mac or a Linux machine. You'll be able

to work with Windows, but some things may be higher-friction or require extra workarounds. The new Linux subsystem for Windows 10 helps with this, but still isn't ideal.

Docker on Windows and the Mac will use virtual machines to host a Linux kernel (required for Docker's container tech), and as such you may find your machine struggling a bit if you don't have enough RAM.

If you're using Linux (I used Ubuntu to verify the code), then you don't need any virtual machines as Docker can run directly on top of a Linux kernel.

Online Resources

- Microsoft's website
- This book's GitHub repo

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/microservices-aspnetcore*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: *Building Microservices with ASP.NET Core* by Kevin Hoffman (O'Reilly). Copyright 2017 Kevin Hoffman, 978-1-491-96173-5.

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Safari

Safari (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit *http://oreilly.com/safari*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://oreil.ly/2esotzv*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

This book would not have been possible without the superhuman patience and tolerance of my family. Their support is the only thing that helped take this book from a concept to a published work. I honestly don't know how they put up with my stress and quirks and awful schedule of travel, maintaining my day job, and devoting an absurd amount of hours to this book.

For every chapter and sample in a book like this, there are countless hours of coding, testing, research, consulting with experts, and the mandatory smashing of the head on the desk. I need to thank the open source community at large for their involvement and engagement with .NET Core, especially the advocates and developers at Microsoft.

And as always, I must thank the other members of the A-Team (Dan, Chris, and Tom) for continuing to be a source of inspiration that keeps programming fun and interesting.

CHAPTER 1

Building a Microservice with ASP.NET Core

Up to this point in the book we have only been scratching at the surface of the capabilities of .NET Core. In this chapter we're going to expand on the simple "hello world" middleware we've built and create our first microservice.

We'll spend a little time defining what a microservice is (and is not), and discuss concepts like *API First* and *Test-Driven Development*. Then we'll build a sample service that manages teams and team membership.

Microservices Defined

Today, as I have been quoted to say, *we can't swing a dead cat without hitting a microservice.*[1]

The word is everywhere, and unfortunately, it is as overloaded and potentially misleading as the acronym SOA was years ago. Every time we see the word, we're left with questions like, "What is a service, really?" and "Just how micro is micro?" and "Why don't we just call them 'services'?"

These are all great questions that we should be asking. In many cases, the answer is "It depends." However, in my years of building modular and highly scalable applications, I've come up with a definition of *microservice*:

> A microservice is a standalone unit of deployment that supports a specific business goal. It interacts with backing services, and allows interaction through semantically versioned, well-defined APIs. Its defining characteristic is a strict adherence to the Single Responsibility Principle (SRP).

1 Origins of the "can't swing a dead cat" phrase are as morbid as they are plentiful. I have been unable to discover a single credible source for the original quote.

This might seem like a somewhat controversial definition. You'll notice it doesn't mention REST or JSON or XML anywhere. You can have a microservice that interacts with consumers via queues, distributed messaging, or traditional RESTful APIs. The shape and nature of the service's API is *not* the thing that qualifies it as a service or as "micro."

It is a *service* because it, as the name implies, *provides a service*. It is *micro* because it *does one and only one thing*. It's not micro because it consumes a small amount of RAM, or because it consumes a small amount of disk, or because it was handcrafted by artisanal, free-range, grass-fed developers.

The definition also makes a point to mention *semantic versioning*. You cannot continually grow and maintain an organically changing microservice ecosystem without strict adherence to semantic versioning and API compatibility rules. You're welcome to disagree, but consider this: are you building a service that will be deployed to production once, in a vacuum, or building an app that will have dozens of services deployed to production frequently with independent release cycles? If you answered the latter, then you should spend some time considering your API versioning and backward compatibility strategies.

When building a microservice from scratch, ask yourself about the frequency of changes you expect to make to this service and how much of the service might be unrelated to the change (and thus potentially a candidate for being in a separate service).

This brings to mind Sam Newman's golden rule of microservices change:

> Can you make a change to a service and deploy it by itself without changing anything else?
>
> —Sam Newman, *Building Microservices* (O'Reilly)

There's no magic to microservices. In fact, most of us simply consider the current trend toward microservices as just the way Service-Oriented Architecture (SOA) should have been done originally.

The small footprint, easy deployment, and stateless nature of true microservices make them ideal for operating in an elastically scaling cloud environment, which is the focus of this book.

Introducing the Team Service

As fantastic as the typical "hello world" sample might be, it has no practical value whatsoever. More importantly, since we're building our sample with testing in mind, we need real functionality to test. As such, we're going to build a real, semi-useful service that attempts to solve a real problem.

Whether it's sales teams, development teams, support, or any other kind of team, companies with geographically distributed team members often have a difficult time keeping track of those members: their locations, contact information, project assignments, and so forth.

The team service aims to help solve this problem. The service will allow clients to query team lists as well as team members and their details. It should also be possible to add or remove teams and team members.

When designing this service, I tried to think of the many different team visualizations that should be supported by this service, including a map with pins for each team member as well as traditional lists and tables.

In the interest of keeping this sample realistic, individuals should be able to belong to more than one team at a time. If removing a person from a team orphans that person (they're without a team), then that person will be removed. This might not be optimal, but we have to start somewhere and starting with an imperfect solution is far better than waiting for a perfect one.

API First Development

Before we write a single line of code we're going to go through the exercise of defining our service's API. In this section, we'll talk about why *API First* makes sense as a development strategy for teams working on microservices, and then we'll talk about the API for our sample team management service.

Why API First?

If your team is building a "hello world" application that sits in isolation and has no interaction with any other system, then the API First concept isn't going to buy you much.

But in the real world, *especially* when we're deploying all of our services onto a platform that abstracts away our infrastructure (like Kubernetes, AWS, GCP, Cloud Foundry, etc.), even the simplest of services is going to consume other services and will be consumed by services or applications.

Imagine we're building a service used by the services owned and maintained by two other teams. In turn, our service relies upon two more services. Each of the upstream and downstream services is also part of a dependency chain that may or may not be linear. This complexity wasn't a problem back in the day when we would schedule our releases six months out and release *everything* at the same time.

This is not how modern software is built. We're striving for an environment where each of our teams can add features, fix bugs, make enhancements, and deploy to production live without impacting any other services. Ideally we also want to be able to

perform this deployment with zero downtime, without even affecting any live consumers of our service.

If the organization is relying on shared code and other sources of tight, internal coupling between these services, then we run the risk of breaking all kinds of things every time we deploy, and we return to the dark days where we faced a production release with the same sense of dread and fear as a zombie apocalypse.

On the other hand, if every team agrees to conform to published, *well-documented* and semantically versioned[2] APIs as a firm contract, then it frees up each team to work on its own release cadence. Following the rules of semantic versioning will allow teams to enhance their APIs without breaking ones already in use by existing consumers.

You may find that adherence to practices like API First is far more important as a foundation to the success of a microservice ecosystem than the technology or code used to construct it.

If you're looking for guidance on the mechanics of documenting and sharing APIs, you might want to check out API Blueprint and websites like Apiary. There are innumerable other standards, such as the OpenAPI Specification (formerly known as Swagger), but I tend to favor the simplicity offered by documenting APIs with Markdown. Your mileage may vary, and the more rigid format of the OpenAPI Spec may be more suitable for your needs.

The Team Service API

In general, there is nothing requiring the API for a microservice to be RESTful. The API can be a contract defining message queues and message payload formats, or it can be another form of messaging that might include a technology like Google's Protocol Buffers.[3] The point is that RESTful APIs are just one of many ways in which to expose an API from a service.

That said, we're going to be using RESTful APIs for most (but not all) of the services in this book. Our team service API will expose a root resource called `teams`. Beneath that we will have resources that allow consumers to query and manipulate the teams themselves as well as to add and remove members of teams.

For the purposes of simplicity in this chapter, there is no security involved, so any consumer can use any resource. Table 1-1 represents our public API (we'll show the JSON payload formats later).

2 For more information on semver, check out *http://semver.org/*.

3 Protocol Buffers, or "protobufs" for short, are a platform-neutral, high-performance serialization format documented at *https://developers.google.com/protocol-buffers/*.

Table 1-1. Team service API

Resource	Method	Description
`/teams`	GET	Gets a list of all teams
`/teams/{id}`	GET	Gets details for a single team
`/teams/{id}/members`	GET	Gets members of a team
`/teams`	POST	Creates a new team
`/teams/{id}/members`	POST	Adds a member to a team
`/teams/{id}`	PUT	Updates team properties
`/teams/{id}/members/{memberId}`	PUT	Updates member properties
`/teams/{id}/members/{memberId}`	DELETE	Removes a member from the team
`/teams/{id}`	DELETE	Deletes an entire team

Before settling on a final API design, we could use a website like Apiary to take our API Blueprint documentation and turn it into a functioning stub that we can play with until we're satisfied that the API feels right. This exercise might seem like a waste of time, but we would rather discover ugly smells in an API using an automated tool first rather than discovering them after we've already written a test suite to certify that our (ugly) API works.

For example, we might use a mocking tool like Apiary to eventually discover that there's no way to get to a member's information without first knowing the ID of a team to which she belongs. This might irritate us, or we might be fine with it. The important piece is that this discovery might not have happened until too late if we didn't at least simulate exercising the API for common client use cases.

Test-First Controller Development

In this section of the chapter we're going to build a controller to support our newly defined team API. While the focus of this book is not on TDD and I may choose not to show the code for tests in some chapters, I did want to go through the exercise of building a controller test-first so you can experience this in ASP.NET Core.

To start with, we can copy over a couple of the scaffolding classes we created in the previous chapter to create an empty project. I'm trying to avoid using wizards and IDEs as a starting point to avoid locking people into any one platform that would negate the advantages of Core's cross-platform nature. It is also incredibly valuable to know what the wizards are doing and why. Think of this like the math teacher withholding the "easy way" until you've understood why the "hard way" works.

In classic Test-Driven Development (TDD), we start with a failing test. We then make the test pass by writing *just enough* code to make the light go green. Then we write another failing test, and make that one pass. We repeat the entire process until the list of passing tests includes all of our API design that we've done in the preceding table

and we have a test case that asserts the positives and negatives for each of the things the API must support.

We need to write tests that certify that if we send garbage data, we get an HTTP 400 (bad request) back. We need to write tests that certify that all of our controller methods behave as expected in the presence of missing, corrupt, or otherwise invalid data.

One of the key tenets of TDD that a lot of people don't pick up on is that a compilation failure *is a failing test*. If we write a test asserting that our controller returns some piece of data and the controller doesn't yet exist, that's still a failing test. We make that test pass by creating the controller class, and adding a method that returns just enough data to make the test pass. From there, we can continue iterating through expanding the test to go through the fail–pass–repeat cycle.

This cycle relies on very small iterations, but adhering to it and building habits around it can dramatically increase your confidence in your code. Confidence in your code is a key factor in making rapid and automated releases successful.

If you want to learn more about TDD in general, then I highly recommend reading *Test Driven Development* by Kent Beck (Addison-Wesley Professional). The book is old but the concepts outlined within it still hold true today. Further, if you're curious as to the naming conventions used for the tests in this book, they are the same guidelines as those used by the Microsoft engineering team that built ASP.NET Core.

Each of our unit test methods will have three components:

Arrange
: Perform any setup necessary to prepare the test.

Act
: Execute the code under test.

Assert
: Verify the test conditions in order to determine pass/fail.

The "arrange, act, assert" pattern is a pretty common one for organizing the code in unit tests but, like all patterns, is a recommendation and doesn't apply universally.

Our first test is going to be very simple, though as you'll see, it's often the one that takes the most time because we're starting with nothing. This test will be called `QueryTeamListReturnsCorrectTeams`. The first thing this method does is verify that we get *any* result back from the controller. We'll want to verify more than that eventually, but we have to start somewhere, and that's with a failing test.

First, we need a test project. This is going to be a separate module that contains our tests. Per Microsoft convention, if we have an assembly called `Foo`, then the test assembly is called `Foo.Tests`.

In our case, we are building applications for a fictitious company called the *Statler and Waldorf Corporation*. As such, our team service will be in a project called *StatlerWaldorfCorp.TeamService* and the tests will be in *StatlerWaldorfCorp.TeamService.Tests*. If you're curious about the inspiration for this company, it is a combination of the appreciation of cranky old hecklers and the Muppets of the same name.

To set this up, we'll create a single root directory that will contain both the main project and the test project. The main project will be in *src/StatlerWaldorfCorp.TeamService* and the test project will be in *test/StatlerWaldorfCorp.TeamService.Tests*. To get started, we're just going to reuse the *Program.cs* and *Startup.cs* boilerplate from the last chapter so that we just have something to compile, so we can add a reference to it from our test module.

To give you an idea of the solution that we're building toward, Example 1-1 is an illustration of the directory structure and the files that we'll be building.

Example 1-1. Eventual project structure for the team service

```
├── src
│   └── StatlerWaldorfCorp.TeamService
│       ├── Models
│       │   ├── Member.cs
│       │   └── Team.cs
│       ├── Program.cs
│       ├── Startup.cs
│       ├── StatlerWaldorfCorp.TeamService.csproj
│       └── TeamsController.cs
└── test
    └── StatlerWaldorfCorp.TeamService.Tests
        ├── StatlerWaldorfCorp.TeamService.Tests.csproj
        └── TeamsControllerTest.cs
```

If you're using the full version of Visual Studio, then creating this project structure is fairly easy to do, as is creating and manipulating the relevant *.csproj* files. A point on which I will continue to harp is that for automation and simplicity, all of this needs to be something you can do with simple text editors and command-line tools.

As such, Example 1-2 contains the XML for the *StatlerWaldorf.TeamService.Tests.csproj* project file. Pay special attention to how the test project references the project under test and how we *do not* have to redeclare dependencies we inherit from the main project.

Example 1-2. StatlerWaldorfCorp.TeamService.Tests.csproj

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
```

```
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp1.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <ProjectReference
      Include
="../../src/StatlerWaldorfCorp.TeamService/StatlerWaldorfCorp.TeamService.csproj"/>
    <PackageReference Include="Microsoft.NET.Test.Sdk"
       Version="15.0.0-preview-20170210-02" />
    <PackageReference Include="xunit"
       Version="2.2.0" />
    <PackageReference Include="xunit.runner.visualstudio"
       Version="2.2.0" />
  </ItemGroup>
</Project>
```

Before we create a controller test and a controller, let's just create a class for the `Team` model, as in Example 1-3.

Example 1-3. src/StatlerWaldorfCorp.TeamService/Models/Team.cs

```
using System;
using System.Collections.Generic;

namespace StatlerWaldorfCorp.TeamService.Models
{
    public class Team {

        public string Name { get; set; }
        public Guid ID { get; set; }
        public ICollection<Member> Members { get; set; }

        public Team()
        {
            this.Members = new List<Member>();
        }

        public Team(string name) : this()
        {
            this.Name = name;
        }

        public Team(string name, Guid id)  : this(name)
        {
            this.ID = id;
        }

        public override string ToString() {
            return this.Name;
        }
```

```
    }
}
```

Since each team is going to need a collection of `Member` objects in order to compile, let's create the `Member` class now as well, as in Example 1-4.

Example 1-4. src/StatlerWaldorfCorp.TeamService/Models/Member.cs

```
using System;

namespace StatlerWaldorfCorp.TeamService.Models
{
    public class Member {
        public Guid ID { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }

        public Member() {
        }

        public Member(Guid id) : this() {
            this.ID = id;
        }

        public Member(string firstName,
          string lastName, Guid id) : this(id) {
            this.FirstName = firstName;
            this.LastName = lastName;
        }

        public override string ToString() {
            return this.LastName;
        }
    }
}
```

In a complete, 100% pure TDD world, we would have created the failing test first and then gone and created all of the things we need to allow it to compile. Since these are just simple model objects, I don't mind skipping a few steps.

Now let's create our first failing test, shown in Example 1-5.

Example 1-5. test/StatlerWaldorfCorp.TeamService.Tests/TeamsControllerTest.cs

```
using Xunit;
using StatlerWaldorfCorp.TeamService.Models;
using System.Collections.Generic;

namespace StatlerWaldorfCorp.TeamService
{
```

```
    public class TeamsControllerTest
    {
        TeamsController controller = new TeamsController();

        [Fact]
        public void QueryTeamListReturnsCorrectTeams()
        {
            List<Team> teams = new List<Team>(
               controller.GetAllTeams());
        }
    }
}
```

To see this test fail, open a terminal and `cd` to the *test/StatlerWaldorf.TeamService.Tests* directory. Then run the following commands:

```
$ dotnet restore
...
$ dotnet test
...
```

The `dotnet test` command invokes the test runner and executes all discovered tests. We use `dotnet restore` to make sure that the test runner has all the dependencies and transitive dependencies necessary to build and run. As expected, the `test` command will fail if either the test code or the project being tested fails to compile.

This test doesn't compile because we're missing the controller we want to test. To make this pass, we're going to need to add a `TeamsController` to our main project that looks like Example 1-6.

Example 1-6. src/StatlerWaldorfCorp.TeamService/Controllers/TeamsController.cs

```
using System;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using System.Linq;
using StatlerWaldorfCorp.TeamService.Models;

namespace StatlerWaldorfCorp.TeamService
{
  public class TeamsController
  {
    public TeamsController() {
    }

    [HttpGet]
    public IEnumerable<Team> GetAllTeams()
    {
      return Enumerable.Empty<Team>();
```

```
    }
  }
}
```

With this first test passing (it just asserts that we can call the method), we want to add a new assertion that we know is going to fail. In this case, we want to check that we get the right number of teams in response. Since we don't (yet) have a mock, we'll come up with an arbitrary number:

```
    List<Team> teams = new List<Team>(controller.GetAllTeams());
    Assert.Equal(teams.Count, 2);
```

Now let's make this test pass by hardcoding some random nonsense in the controller. A lot of people like to skip this step because they're in a hurry, they're over-caffeinated, or they don't fully appreciate the iterative nature of TDD.

You don't need those kinds of people in your life.

The small iterations of writing just enough code to make a test pass is the part of the discipline that not only makes it work, but builds high confidence levels in tested code. I also find that the practice of writing *just enough* code to make something pass allows me to avoid creating bloated APIs and lets me refine my APIs and interfaces as I test.

Example 1-7 shows the updated `TeamsController` class to support the new test.

Example 1-7. Updated src/StatlerWaldorfCorp.TeamService/Controllers/TeamsController.cs

```
using System;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using System.Linq;
using StatlerWaldorfCorp.TeamService.Models;

namespace StatlerWaldorfCorp.TeamService
{
  public class TeamsController
  {
    public TeamsController() {
    }

    [HttpGet]
    public IEnumerable<Team> GetAllTeams()
    {
      return new Team[] { new Team("one"), new Team("two") };
    }
```

```
  }
}
```

There are very few negative tests we can do for a simple GET method that operates on a collection without parameters, so let's move on to the method for adding a team.

To test this, we're going to query the team list; we'll then invoke a new `CreateTeam` method, and then we're going to query the team list again. Our assertion should be that our new team is in the list.

In the strictest adherence to TDD, we wouldn't preemptively change things unless we did so to make a test pass. However, to keep the listings in the book down to a reasonable size I decided to bypass that. So far, our controller hasn't inherited from a base class, nor has it been returning anything that allows us to control the HTTP response itself (it's been returning raw values).

This isn't going to be sustainable, so we're going to change the way we're defining our controller methods and reflect our *desire* for this new pattern in the failing test shown in Example 1-8.

Example 1-8. TeamsControllerTest.cs—the CreateTeamAddsTeamToList test

```
[Fact]
public async void CreateTeamAddsTeamToList()
{
  TeamsController controller = new TeamsController();
  var teams = (IEnumerable<Team>)
    (await controller.GetAllTeams() as ObjectResult).Value;
  List<Team> original = new List<Team>(teams);

  Team t = new Team("sample");
  var result = await controller.CreateTeam(t);

  var newTeamsRaw =
   (IEnumerable<Team>)
     (await controller.GetAllTeams() as ObjectResult).Value;

  List<Team> newTeams = new List<Team>(newTeamsRaw);
  Assert.Equal(newTeams.Count, original.Count+1);
  var sampleTeam =
   newTeams.FirstOrDefault(
     target => target.Name == "sample");
  Assert.NotNull(sampleTeam);
}
```

The code here looks a little rough around the edges, but that's okay for now. While tests are passing, we can refactor both our tests and the code under test.

To make this test pass, we need to create the `CreateTeam` method on the controller. Once we get into the thick of that method, we'll need some way to store teams. In a real-world service, we don't want to do that in memory because that would violate the *stateless* rule for cloud-native services.

However, for testing it's ideal because we can easily manufacture any state we like for testing. So, we'll create the `CreateTeam` method that is a no-op, and we'll see that our test now compiles but fails. To make this pass, we're going to need a *repository*.

Injecting a Mock Repository

We know that we're going to have to get our `CreateTeamAddsTeamToList` test to pass by giving the test suite control over the controller's internal storage. This is typically done through mocks or through injecting fakes, or a combination of both.

I've elided a few of the iterations of test-driven development necessary to get us to the point where we can build an interface to represent the repository and refactor the controller to accept it.

We're now going to create an interface called `ITeamRepository` (shown in Example 1-9), which is the interface that will be used by our tests for a fake and eventually by the service project for a real persistence medium, but we won't code that yet. Remember, we're not going to code anything that doesn't convert a failing test into a passing one.

Example 1-9. src/StatlerWaldorfCorp.TeamService/Persistence/ITeamRepository.cs

```
using System.Collections.Generic;

namespace StatlerWaldorfCorp.TeamService.Persistence
{
  public interface ITeamRepository {
       IEnumerable<Team> GetTeams();
       void AddTeam(Team team);
  }
}
```

We could probably try and predict something more useful than a void return value for `AddTeam`, but right now we don't need to. So let's create an in-memory implementation of this repository interface in the service project, as in Example 1-10.

Example 1-10. src/StatlerWaldorfCorp.TeamService/Persistence/MemoryTeamRepository.cs

```
using System.Collections.Generic;

namespace StatlerWaldorfCorp.TeamService.Persistence
```

```
{
  public class MemoryTeamRepository :  ITeamRepository {
    protected static ICollection<Team> teams;

    public MemoryTeamRepository() {
      if(teams == null) {
        teams = new List<Team>();
      }
    }

    public MemoryTeamRepository(ICollection<Team> teams) {
      teams = teams;
    }

    public IEnumerable<Team> GetTeams() {
      return teams;
    }

    public void AddTeam(Team t)
    {
      teams.Add(t);
    }
  }
}
```

If you're cringing at the sight of a static collection as a private member of a class, then that's a *good thing*—you can smell bad code when you're within range. This is, however, code just good enough to make a test pass. If we were intending to use this class for anything other than tests, we'd include multiple rounds of refactoring after we had a complete test suite.

Injecting this interface into our controller is actually quite easy. ASP.NET Core already comes equipped with a scope-aware dependency injection (DI) system. Using this DI system, we're going to add the repository as a *service* in our `Startup` class, as shown in the following snippet:

```
    public void ConfigureServices(IServiceCollection services)
    {
      services.AddMvc();
      services.AddScoped<ITeamRepository, MemoryTeamRepository>();
    }
```

Using the services model, we can now use *constructor injection* in our controllers and ASP.NET Core will automatically add an instance of the repository to any controller that wants it.

We use the `AddScoped` method because we want the DI subsystem to create a new instance of this repository *for every request*. At this point we don't really know what our actual backing repository is going to be—SQL Server, a document database, or maybe even another microservice. We do know that we want *this* microservice to be

stateless, and the best way to do that is to start with per-request repositories and only switch to singletons if we have no other alternative.

Property Versus Constructor Injection

The debate over which method is best will continue raging until long after human beings are even writing code. I prefer constructor injection because it makes the dependencies of a class *explicit*. There's no magic, no detective work involved, and constructor injection is *much* easier to test with mocks and stubs.

Now that we've got a class we can use for our repository, let's modify the controller so that we can inject it by adding a simple constructor parameter:

```
public class TeamsController : Controller
{
    ITeamRepository repository;

    public TeamsController(ITeamRepository repo)
    {
      repository = repo;
    }

    ...
}
```

Note that there are no attributes or annotations required to enable this parameter for dependency injection. This may seem like a triviality, but I've grown quite fond of this fact when working with large codebases.

Now we can modify our existing controller method so that it uses the repository instead of returning hardcoded data:

```
[HttpGet]
public async virtual Task<IActionResult> GetAllTeams()
{
  return this.Ok(repository.GetTeams());
}
```

Next we can make our existing tests pass by going back into our test module and pre-populating the repository with a set of test teams (our tests assume two teams). The test for the collection's getter method will use whatever we supply in the repository so we can make reliable assertions.

It's worth reiterating that our goal with controller tests is to test only the responsibility of the controller. At this point, that means we're *only* testing to make sure that the appropriate methods are being called on the repository. We could have used a mock-

ing framework to avoid creating a custom repository, but the in-memory version is so simple we decided not to incur the overhead of mocking.

Mocking Frameworks

While I don't use mocks much in this book, I have played around with various mocking frameworks available for .NET Core. At the time of this writing my favorite was Moq, but feel free to explore on your own to find one that suits your needs.

Just remember the cardinal rule of tools also applies to libraries. They should make your life *easier*, but you should be able to get by without them. If you can't test something without a complicated mock and simple fakes won't do, maybe the class design needs to be refactored.

Completing the Unit Test Suite

I'm not going to bloat the pages in this book by listing every line of code in all of the tests. To finish the unit test suite, we're going to continue with our iterative process of adding a failing test and then writing just enough code to make that test pass.

The source code for the full set of tests can be found in the `master` branch on GitHub.

The following is an overview of some of the features of the code enabled through TDD:

- You cannot add members to nonexistent teams.
- You can add a member to an existing team, verified by querying the team details.
- You can remove a member from an existing team, verified by querying team details.
- You cannot remove members from a team to which they don't belong.

One thing you'll note about these tests is that they don't dictate the *internal* manner of persisting teams and their members. Under the current design, the API doesn't allow independent access to people; you have to go through a team. We might want to change that in the future, but for now that's what we're going with because a functioning product can be refactored, whereas a beautiful yet nonexistent product cannot.

To see these tests in action, first build the main source project, then go into the *test/StatlerWaldorfCorp.TeamService.Tests* folder and issue the following commands:

```
$ dotnet restore
...
$ dotnet build
...
$ dotnet test
Build started, please wait...
```

```
Build completed.

Test run for /Users/kevin/Code/microservices-aspnetcore/ \
teamservice/test/StatlerWaldorfCorp.TeamService.Tests/bin/Debug/ \
netcoreapp1.1/StatlerWaldorfCorp.TeamService.Tests.dll(
  .NETCoreApp,Version=v1.1)
Microsoft (R) Test Execution Command Line Tool Version 15.0.0.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...
[xUnit.net 00:00:01.1279308]   Discovering: StatlerWaldorfCorp.TeamService.Tests
[xUnit.net 00:00:01.3207980]   Discovered:  StatlerWaldorfCorp.TeamService.Tests
[xUnit.net 00:00:01.3977448]   Starting:    StatlerWaldorfCorp.TeamService.Tests
[xUnit.net 00:00:01.6546338]   Finished:    StatlerWaldorfCorp.TeamService.Tests

Total tests: 18. Passed: 18. Failed: 0. Skipped: 0.
Test Run Successful.
Test execution time: 2.5591 Seconds
```

Happily, it appears that all 18 of our unit tests have passed!

Creating a CI Pipeline

Having tests is great, but they don't do anyone any good if they aren't run *all the time*, every time someone commits code to a branch. Continuous integration is a key aspect of being able to rapidly deliver new features and fixes, regardless of your team size or geographic makeup.

In the previous chapter, we created a Wercker account and we went through all of the steps necessary to use the Wercker CLI and Docker to automate testing and deploying our applications. It should now be incredibly easy to take our fully unit-tested codebase and set up an automated build pipeline.

Let's take a look at the *wercker.yml* file for the team service, shown in Example 1-11.

Example 1-11. wercker.yml

```
box: microsoft/dotnet:1.1.1-sdk
no-response-timeout: 10
build:
  steps:
    - script:
        name: restore
        cwd: src/StatlerWaldorfCorp.TeamService
        code: |
          dotnet restore
    - script:
        name: build
        cwd: src/StatlerWaldorfCorp.TeamService
        code: |
          dotnet build
    - script:
        name: publish
        cwd: src/StatlerWaldorfCorp.TeamService
        code: |
          dotnet publish -o publish
    - script:
        name: test-restore
        cwd: test/StatlerWaldorfCorp.TeamService.Tests
        code: |
           dotnet restore
    - script:
        name: test-build
        cwd: test/StatlerWaldorfCorp.TeamService.Tests
        code: |
          dotnet build
    - script:
        name: test-run
        cwd: test/StatlerWaldorfCorp.TeamService.Tests
        code: |
          dotnet test
    - script:
        name: copy binary
        cwd: src/StatlerWaldorfCorp.TeamService
        code: |
          cp -r . $WERCKER_OUTPUT_DIR/app
deploy:
  steps:
    - internal/docker-push:
        cwd: $WERCKER_OUTPUT_DIR/app
        username: $USERNAME
        password: $PASSWORD
        repository: dotnetcoreservices/teamservice
        registry: https://registry.hub.docker.com
        entrypoint: "/pipeline/source/app/docker_entrypoint.sh"
```

The first thing to notice is the choice of `box` in the configuration. This needs to be a docker hub image that already contains the .NET Core command-line tooling. In this case, I chose `microsoft/dotnet:1.1.1-sdk`. This may change depending on which version is the most current as you're reading this, so be sure to check the official Microsoft docker hub repository for the latest tags and check the GitHub repository for this book to see what boxes are being used for tests.

In some cases we can skip certain steps and go directly to testing, but if a step is going to fail, we want it to be as small as possible so we can troubleshoot it. You can execute all of these build steps on your development workstation, assuming you have the Wercker CLI installed and a running Docker installation. Just execute the *buildlocal.sh* script that you can find in this chapter's GitHub repository. This script contains the following code and will execute the same build locally that Wercker will execute remotely:

```
rm -rf _builds _steps _projects _cache _temp
wercker build --git-domain github.com \
   --git-owner microservices-aspnetcore \
   --git-repository teamservice
rm -rf _builds _steps _projects _cache _temp
```

Integration Testing

The most official definition of integration testing that I've been able to find indicates that it is the stage of testing when individual components are combined and tested as a group. This phase occurs *after* unit testing and *before* validation (also called acceptance) testing.

There are some subtleties about this definition that are important. Unit tests verify that your modules do what you expect them do. An integration test should not verify that you get the *right* answers from the system; it should verify that all of the components of the system are connected and you get *suitable* responses. In other words, if you're performing complex calculations using components already covered by unit tests, your integration tests need not retest those components. Integration tests would simply verify that you can invoke your web server, trigger the right RESTful endpoint, invoke the complex calculator, and get an appropriate response.

One of the hardest parts of integration testing usually ends up being the technology or code involved in spinning up an instance of the web hosting machinery so that you can send and receive full HTTP messages.

Thankfully, this has already been taken care of for us with the `Microsoft.AspNetCore.TestHost.TestServer` class. We can instantiate one of these and build it with whatever options we like and then use it to create an instance of an `HttpClient` that is preconfigured to talk to our test server. The creation of these two classes is usually done in an integration test's constructor, as shown in this snippet:

```
    testServer = new TestServer(new WebHostBuilder()
                        .UseStartup<Startup>());
    testClient = testServer.CreateClient();
```

Note that the `Startup` class we're using here is the exact same one we're using in our main service project. This means that the dependency injection setup, configuration sources, and services will all be *exactly* as they would be if we were running the real service.

With the test server and test client in place, we can test various scenarios, like adding a team to the teams collection and querying the results to ensure that it's still there. This gives us a chance to fully exercise the JSON deserialization and use our service the way a completely external consumer might, as shown in Example 1-12.

Example 1-12. test/StatlerWaldorfCorp.TeamService.Tests.Integration/SimpleIntegrationTests.cs

```
public class SimpleIntegrationTests
{
  private readonly TestServer testServer;
  private readonly HttpClient testClient;

  private readonly Team teamZombie;

  public SimpleIntegrationTests()
  {
      testServer = new TestServer(new WebHostBuilder()
              .UseStartup<Startup>());
      testClient = testServer.CreateClient();

      teamZombie = new Team() {
          ID = Guid.NewGuid(),
          Name = "Zombie"
      };
  }

  [Fact]
  public async void TestTeamPostAndGet()
  {
     StringContent stringContent = new StringContent(
         JsonConvert.SerializeObject(teamZombie),
         UnicodeEncoding.UTF8,
         "application/json");

     HttpResponseMessage postResponse =
        await testClient.PostAsync(
          "/teams",
          stringContent);
     postResponse.EnsureSuccessStatusCode();
```

```
        var getResponse = await testClient.GetAsync("/teams");
        getResponse.EnsureSuccessStatusCode();

        string raw = await getResponse.Content.ReadAsStringAsync();
        List<Team> teams =
            JsonConvert.DeserializeObject<List<Team>>(raw);
        Assert.Equal(1, teams.Count());
        Assert.Equal("Zombie", teams[0].Name);
        Assert.Equal(teamZombie.ID, teams[0].ID);
    }
}
```

Once we're satisfied that this test works properly, we can continue adding more complex scenarios to ensure that various scenarios are supported and working properly.

With our integration tests ready to roll we can update our *wercker.yml* file to execute the integration tests by adding a few script executions:

```
    - script:
        name: integration-test-restore
        cwd: test/StatlerWaldorfCorp.TeamService.Tests.Integration
        code: |
          dotnet restore
    - script:
        name: integration-test-build
        cwd: test/StatlerWaldorfCorp.TeamService.Tests.Integration
        code: |
          dotnet build
    - script:
        name: integration-test-run
        cwd: test/StatlerWaldorfCorp.TeamService.Tests.Integration
        code: |
          dotnet test
```

For such a simple service as this one, it might seem like we've gone to some needless trouble in creating a separate project for our integration tests and using separate CI pipeline build steps.

However, developing habits and practices that you use even on the smallest projects will pay off in the long run. This is one of them. When we get to the stage where we're building services that rely on other services, we're going to want to start up versions of those services while running integration tests. We want the ability to selectively only run unit tests versus integration tests in our pipelines so we can have a "slow build" and a "fast build" if we want. Also, separating the integration tests into their own project gives us a little bit more cleanliness and organization—some of the integration tests I've written in the past have gotten very large, especially when it comes to fabricating test data and expected response JSON payloads for complex services.

Running the Team Service Docker Image

Now that the CI pipeline is working for the team service, it should automatically be deploying a Docker image to docker hub for us. With this Docker image in hand, we can deploy it to Amazon Web Services, Google Cloud Platform, Microsoft Azure, or regular virtual machines. We could orchestrate this image inside Docker Swarm or Kubernetes or push it to Cloud Foundry.

Our options are nearly endless, but they're endless *because* we're using Docker images as deployment artifacts.

Let's run this using a command you should be pretty familiar with by now:

```
$ docker run -p 8080:8080 dotnetcoreservices/teamservice
Unable to find image 'dotnetcoreservices/teamservice:latest' locally
latest: Pulling from dotnetcoreservices/teamservice
693502eb7dfb: Already exists
081cd4bfd521: Already exists
5d2dc01312f3: Already exists
36c0e9895097: Already exists
3a6b0262adbb: Already exists
79e416d3fe9d: Already exists
d96153ed695f: Pull complete
Digest: sha256:fc3ea65afe84c33f5644bbec0976b4d2d9bc943ddba997103dd3fb731f56ca5b
Status: Downloaded newer image for dotnetcoreservices/teamservice:latest
Hosting environment: Production
Content root path: /pipeline/source/app/publish
Now listening on: http://0.0.0.0:8080
Application started. Press Ctrl+C to shut down.
```

With the port mapping in place, we can treat `http://localhost:8080` as the host of our service now. The following `curl` command issues a POST to the `/teams` resource of the service. (If you don't have access to `curl`, I highly recommend the Postman plug-in for Chrome.) Per our test specification, this should return a JSON payload containing the newly created team:

```
$ curl -H "Content-Type:application/json" \
   -X POST -d \
   '{"id":"e52baa63-d511-417e-9e54-7aab04286281", \
    "name":"Team Zombie"}' \
    http://localhost:8080/teams

{"name":"Team Zombie","id":"e52baa63-d511-417e-9e54-7aab04286281",
  "members":[]}
```

Note that the reply in the preceding snippet contains an empty array for the `members` property. To make sure that the service is maintaining state between requests (even if it is doing so with little more than an in-memory list at the moment), we can use the following `curl` command:

```
$ curl http://localhost:8080/teams
  [{"name":"Team Zombie",
  "id":"e52baa63-d511-417e-9e54-7aab04286281",
   "members":[]}]
```

And that's it—we've got a fully functioning team service automatically tested and automatically deployed to docker hub, ready for scheduling in a cloud computing environment in response to every single Git commit.

Summary

In this chapter we took our first step toward building real microservices with ASP.NET Core. We took a look at the definition of a microservice and we discussed the concept of API First and how it is an essential part of building the discipline and habits necessary to allow multiple teams to have independent release cadences.

Finally, we built a sample service in a test-first fashion and looked at some of the tools we have at our disposal for automatically testing, building, and deploying our services.

In the coming chapters, we're going to expand on these skills as we build more complex and powerful services.

CHAPTER 2

Service Discovery

Up to this point in the book, we have discussed the concepts and code required to build basic microservices, configure and consume backing services, talk to databases, and build web applications. We've even spent a great deal of time and effort discussing the Event Sourcing and CQRS patterns and how they can be applied to build massive-scale applications out of a suite of related microservices.

In this chapter we're going to continue to build on the idea that we don't simply build single services in a vacuum; that everything we build is consumed by or consumes other services.

To keep the configuration and management of large numbers of services as simple as possible, I'm going to introduce the concept of *service discovery*.

Refresher on Cloud-Native Factors

Before we get into the details of service discovery, I thought it would be worth a quick refresher on some of the original *twelve factors* of cloud-native applications that are important and relevant to the sample we'll be building: *external configuration* and *backing services*.

External Configuration

As discussed throughout this book and on the original Twelve-Factor App website, properly handling configuration is key to building applications that thrive in the cloud.

Let's start with a review of what it looks like when we aren't properly externalizing our configuration. How many times have you seen (or written) code that looks like this in your application?

```
using (var httpClient = new HttpClient())
{
    httpClient.BaseAddress = new Uri("http://foo.bar/baz");
    ...
}
```

The address of the backing service is *hardcoded* in your application code. When you commit this to your version-control system, the URL is sitting there, unaltered. This is even more problematic if you've embedded credentials in the URL. This value can't easily change from one environment to the next, and you have to recompile every time you decide to change hostnames.

When people see how problematic this is, they often move the URL out of the C# code and into a *web.config* file or a *web.<environment>.config* environment-specific configuration file. These are then checked into revision control repositories, and we naively think our problems have been solved.

Unfortunately, *any configuration checked into source control might as well be hard-coded.* You should consider any values sitting in a configuration file (*web.config*, *appsettings.json*, whatever) as *part of your code.* As such, credentials and URLs and other environment-specific settings should *never* be included in these files.

The next logical step in this evolutionary process is to move the URLs and credentials out of configuration files, out of C# files, and into *environment variables*. Written this way, our code makes it obvious what configuration parameters it needs in order to function, but it leaves the responsibility of supplying those values up to the environment.

Whether we're using raw virtual machines, Docker images, or a higher-level PaaS, we should always have ways to securely inject environment variables into our applications.

Backing Services

I've harped on this concept enough in this book that you're probably getting sick of seeing it. It is actually worth repeating this point, though everything your application needs must be treated as though it is a backing service.

Whether you need binary storage for files, a database, another web service, a queue service, or anything else, the thing you need should be *loosely coupled*, and *configured from the environment*.

There are two ways to *bind* a resource that is a backing service: static binding and dynamic (runtime) binding. So far in this book we've only discussed static binding.

Statically bound resources

Statically bound resources are the ones we've been using in all of our sample code up to this point. While we've been careful to allow for environment-based replacement of default values to connect to databases, web services, and queuing services, this binding is fixed by the environment.

Whether defined by automation tools or DevOps personnel, the binding between the service and its resource is persistent and made available to the application *at start time*, and it *does not change*.

While this certainly satisfies the external configuration requirement for cloud-native applications, it might not be flexible enough for your needs. Maybe you want something a little more dynamic and powerful.

Dynamically bound resources

A dynamically bound resource is one where the binding occurs at *runtime*. Moreover, this binding is not fixed and can actually change at runtime between requests to the application.

In addition to freeing up the developers of the application from a little bit of complexity, it also allows for even looser coupling. This dynamic, loose runtime coupling between apps and bound resources facilitates more advanced functionality like failover, load balancing, fault tolerance—all with no visible impact to the application code.

Dynamic resource binding will be the focus of the rest of this chapter.

Dynamic Binding May Require One Static Binding

Dynamic resource binding is often managed through a broker or some central point of management that keeps a catalog of services. For this to work, your application needs to know how to find the broker/manager. This is usually done, oddly enough, through a static resource binding. It is for this reason and additional complexity that you should evaluate the number of services you have and how much you need the dynamic binding before you implement discovery.

Introducing Netflix Eureka

In order to discover services at runtime, you're going to need something that serves as a *service registry*; a central catalog of services. This registry and the features it offers can vary from product to product, but by and large most service registries provide at the basic level a list of services, metadata about the services, and their endpoint(s).

You may also be able to get status or heartbeat information to help you determine if a service that claims to be online really is online.

Netflix's infrastructure is predominantly run on top of Amazon Web Services. As you might imagine from the size and complexity of the product Netflix offers and the sheer volume of concurrently connected customers, Netflix's microservice ecosystem is *vast*, to say the least.

While AWS has plenty of functionality available for load balancing at the edge, and it has a naming service (Route 53) that can function as a full DNS service, neither of these services are entirely appropriate for mid-tier service naming, registry, and load balancing.

With Eureka, Netflix built its own internal product to manage the service registry that allowed for failover and load balancing. It's still using a far more advanced version of this product internally, but to our gain and enjoyment, Netflix has open sourced the core of its functionality. You can find this code on GitHub.

From a developer's perspective, your service code interacts with a Eureka server by registering itself when it starts up. If you need to discover and consume other backing services, then you can ask the Eureka server for some or all of the service registry. Your service also emits a *heartbeat* to the Eureka service at some interval (usually 30 seconds). If your service fails to send a heartbeat to Eureka after some number of intervals, it will be taken out of the registry.

If there are multiple instances of your service running, and consumers of your service are talking to Eureka to find your service, then they will stop getting the URL of the service that was taken out of the registry due to heartbeat failure and will simply refer to the service instances that are up and running.

Figure 2-1, available on the Eureka at a Glance documentation page on GitHub, illustrates how Netflix has deployed Eureka and how it expects it to be deployed in a typical organization.

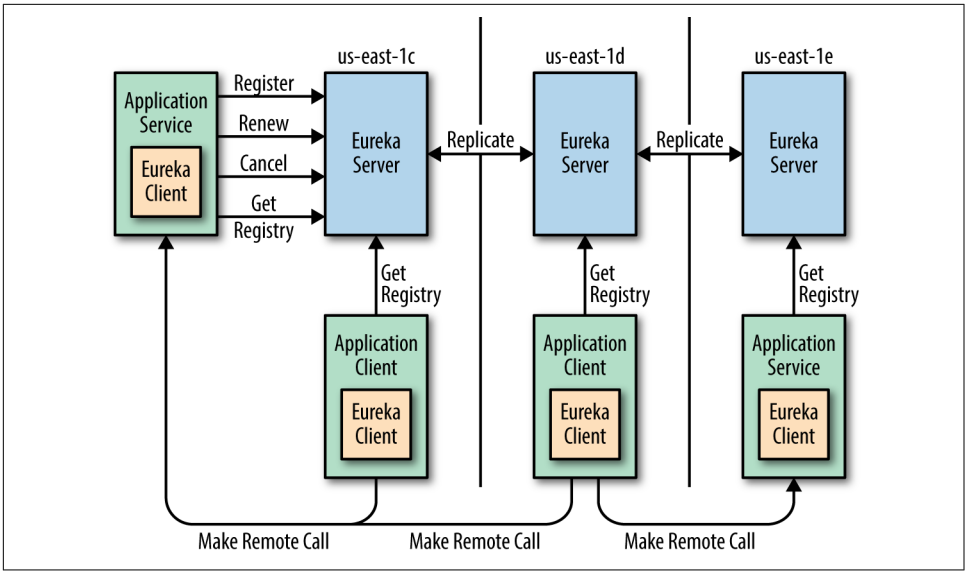


Figure 2-1. Typical Eureka deployment within AWS

Eureka is a powerful product with a lot of features that we simply don't have the room to discuss at length in this book. I encourage you to read up on the documentation and more advanced features if you think your project or organization might benefit from these features.

Eureka is also not the only player in the service registry and discovery game. There are a number of other companies and products available that provide everything from the bare-bones service registry to more full-featured registry, discovery, and fault-tolerance functionality.

Just a few of the more popular products are:

`etcd`
: Pronounced "etsee-dee," `etcd` is a low-level distributed key-value store accessible via HTTP. As such, you actually need to bolt on additional tools to make it serve as a service discovery and registry mechanism. You'll often see tools like `etcd` combined with `registrator` and `confd` to bring them up to par with other products like Consul and Eureka.

Consul
: Consul is full-featured service discovery tool that and also provides a key-value store for configuration. It uses a gossip protocol to form clusters.

Marathon
: Marathon is a full-fledged container orchestration system for Mesos and DC/OS. As such, it does *a lot* more than just service discovery. Service discovery is more like a free benefit you get by using Marathon as your container orchestration layer.

ZooKeeper
: Originally born of the Hadoop project, ZooKeeper is one of the oldest of this family of products and is mature and stable, though many argue it is showing its age and you may be better off with other products.

If you want to try out Eureka (and use it for the code later in this chapter) without the commitment of having to build it from source or install a full copy on a server, you can just run it from a docker hub image, as shown in the following command:

```
$ docker run -p 8080:8080 -d --name eureka \
   -d netflixoss/eureka:1.3.1
```

This will run a default, *nonproduction* copy of the server in your Docker virtual machine and map port 8080 from inside the container to your local machine. This means, assuming port 8080 is available, that when you statically bind an application to this Eureka server, you'll use the URL `http://localhost:8080/eureka`.

Discovering and Advertising ASP.NET Core Services

Now that we've spent some time discussing the concepts and real-world scenarios that drive the need for service discovery, let's take a look at some sample code that communicates with a Eureka server to do just that.

In our somewhat contrived sample, we're going to be building a suite of services that support an ecommerce application. The edge service is responsible for exposing a product catalog. This catalog has standard API endpoints for exposing a list of products as well as product details. There is also an inventory service that is responsible for exposing the real-time status of physical inventory. The product service will need to discover the inventory service and make calls to it in order to return enriched data when asked for product details.

To keep things simple, our sample has just these two services. In a real, large-scale application that supports mobile clients, and internal customers, communicates with multiple third-party vendors, and orchestrates multiple data flows, you could see communication between dozens or hundreds of services. As you saw in Netflix's diagram earlier in the chapter, the need to provide high availability, failover, and load balancing across regions and within regions is satisfied by multiple installations of Eureka.

Registering a Service

The first part of our sample is the inventory service, a service that needs to be dynamically discovered at runtime to provide real-time inventory status.

If you felt like it, you could communicate directly with the Eureka API yourself as it's just a set of RESTful API calls. However, it's always good to look around and see if someone else is actively maintaining a solution to the problem at hand. This saves you the trouble of reinventing the wheel.

In our case, the Steeltoe project maintains a number of client libraries for Netflix OSS projects, including Eureka. While the samples throughout this chapter will rely on the Steeltoe discovery client library, I strongly encourage you to go looking for other libraries as you read this. If you find one that suits your needs better, by all means use that one. At the time this book was written, Steeltoe was basically the only game in town for .NET Core discovery clients.

The Steeltoe library allows us to supply some configuration information using the standard .NET Core configuration system. The key things that we need to declare are the name of our application (this is how it will be identified in the registry) and the URL pointing to the Eureka server, as shown here:

```
{
 "spring": {
   "application": {
     "name":  "inventory"
   }
 },
 "eureka": {
   "client": {
     "serviceUrl": "http://localhost:8080/eureka/",
     "shouldRegisterWithEureka": true,
     "shouldFetchRegistry": false,
     "validate_certificates": false
   },
   "instance": {
     "port": 5000
   }
 }
}
```

Another key part of this configuration is the value `shouldRegisterWithEureka`. If we want our service to be *discoverable* then we must choose `true` here. The next setting, `shouldFetchRegistry`, indicates whether we want to *discover* other services.

Put another way, we need to indicate whether we're consuming registry information or producing it—or both. Our inventory service wants to be discovered and does not need to discover anything else; therefore it will not fetch the registry, but it will register itself.

We'll build our configuration the same way we always do, ensuring that we load the *appsettings.json* file with our discovery client configuration:

```
var builder = new ConfigurationBuilder()
                .SetBasePath(env.ContentRootPath)
                .AddJsonFile("appsettings.json", optional: false,
                reloadOnChange: true)
                .AddEnvironmentVariables();

Configuration = builder.Build();
```

Then we'll use Steeltoe's `AddDiscoveryClient` extension method in our `Startup` class's `ConfigureServices` method:

```
services.AddDiscoveryClient(Configuration);
```

Finally, we just need to add a call to `UseDiscoveryClient` in our `Configure` method:

```
app.UseDiscoveryClient();
```

That's it! Since we're not actually consuming any services from inside the inventory service, we're pretty much done. Obviously we need controllers and an API and to return some fabricated data, but we've covered all of those techniques extensively throughout the book. If you want to look at the rest of the code, just grab the `ecommerce-inventory` repository from the book's GitHub.

We will come back to this service later, after we've created our next service.

Discovering and Consuming Services

With a service ready to be discovered, let's turn our attention to the next service we're going to build: the *catalog*. This service exposes a product catalog and then augments product detail requests by querying the inventory service.

The key difference between this service and the others we've built so far is that this one will dynamically discover the URL of the catalog service at runtime.

We'll configure the client almost the same way we configured the inventory service:

```
"spring": {
  "application": {
    "name": "catalog"
  }
},
"eureka": {
  "client": {
    "serviceUrl": "http://localhost:8080/eureka/",
    "shouldRegisterWithEureka": false,
    "shouldFetchRegistry": true,
    "validate_certificates":  false
  }
}
```

The difference is that the catalog service doesn't need to register (since it does not need to be discovered), and it should fetch the registry so it can discover the inventory service.

Your patience going through this chapter is about to be rewarded. Take a look at the code for the `HttpInventoryClient` class, the class responsible for consuming the inventory service:

```
using StatlerWaldorfCorp.EcommerceCatalog.Models;
using Steeltoe.Discovery.Client;
using System.Threading.Tasks;
using System.Net.Http;
using Newtonsoft.Json;

namespace StatlerWaldorfCorp.EcommerceCatalog.InventoryClient
{
    public class HttpInventoryClient : IInventoryClient
    {
        private DiscoveryHttpClientHandler handler;
        private const string INVENTORYSERVICE_URL_BASE =
              "http://inventory/api/skustatus/";

        public HttpInventoryClient(IDiscoveryClient client)
        {
            this.handler = new DiscoveryHttpClientHandler(client);
        }

        private HttpClient CreateHttpClient()
        {
            return new HttpClient(this.handler, false);
        }

        public async Task<StockStatus> GetStockStatusAsync(int sku) {
            StockStatus stockStatus = null;

            using (HttpClient client = this.CreateHttpClient())
            {
                var result =
                      await client.GetStringAsync(
                        INVENTORYSERVICE_URL_BASE + sku.ToString());
                stockStatus =
                      JsonConvert.DeserializeObject<StockStatus>(
                            result);
            }

            return stockStatus;
        }
    }
}
```

The .NET Core `HttpClient` class has a variant of its constructor that lets you pass in an instance of your own `HttpHandler`. The `DiscoveryHttpClientHandler` provided

by Steeltoe is responsible for swapping the service name in your URL with the actual, runtime-discovered URL. This is what allows our code to rely on a URL like `http://inventory/api/skustatus`, which can then be converted by Steeltoe and Eureka to something like `http://inventory.myapps.mydomain.com/api/skustatus`.

Check out the full sample code for the catalog service and the inventory service.

To run the inventory service, the catalog service, and Eureka all at the same time on your computer, use the following steps.

First, start the Eureka server:

```
$ docker run -p 8080:8080 -d --name eureka \
   -d netflixoss/eureka:1.3.1
```

Then start the inventory service on port 5001:

```
$ cd <inventory service>
$ dotnet run --server.urls=http://0.0.0.0:5001
```

Depending on your computer and what's installed, you might see an error message like this:

```
 Steeltoe.Discovery.Eureka.DiscoveryClient[0]
      Register failed, Exception:
System.PlatformNotSupportedException: The libcurl library in use (7.51.0)
and its SSL backend ("SecureTransport")
do not support custom handling of certificates.
A libcurl built with OpenSSL is required.
```

If this happens and you cannot update your version of `curl` with `openssl` and are still having trouble on your Mac, then you can just run the Linux version of the inventory service right from the book's published docker hub image.

This particular problem adds ammunition to my argument against tightly coupled service discovery libraries; I will discuss that in more depth in the next section of the chapter.

To run the service in Docker, use the following `docker run` command:

```
$ docker run -p 5001:5001 -e PORT=5001 \
  -e EUREKA__CLIENT__SERVICEURL=http://192.168.0.33:8080/eureka/ \
  dotnetcoreservices/ecommerce-inventory
```

When you provide the configuration overrides here, make sure you use your actual machine's IP address. When running inside the Docker image, referring to `localhost` doesn't do anyone any good.

And finally, start the catalog service on port 5001:

```
$ cd <catalog service>
$ dotnet run --server.urls=http://0.0.0.0:5002
```

Now you can issue the appropriate GET requests to the product API for the product list and product details:

`GET http://localhost:5002/api/products`
: Retrieve product list

`GET http://localhost:5002/api/products/{id}`
: Retrieve product details, which will invoke the inventory service, whose URL is dynamically discovered via Eureka

DNS and Platform Supported Discovery

This chapter has basically showed you that through the use of an open source server product and a few client libraries, you can write code to consume URLs in the format `http://`*`service`*`/api` and assume that the library code will swap the word *`service`* for a fully qualified domain name or an IP address.

The main problem I have with this pattern is that it has the side effect of tightly coupling application code with a particular server and client implementation of service discovery. For example, using Eureka (including the helper classes that come with Steeltoe), there's still code that has to manually replace logical service descriptors (e.g., `inventory`) with IP addresses or fully qualified domain names like `inventory.mycluster.mycorp.com`.

Service discovery, registration, and fault detection are all things that I feel should be *nonfunctional requirements*. As such, application code shouldn't contain anything that tightly couples it to some implementation of service discovery.

Obviously, reality and pragmatism must prevail and the decision is ultimately yours, but there are ways to do service discovery without incurring some of the baggage you might get from diving into the deep end with Netflix OSS and Eureka.

Platforms like Kubernetes have plug-ins like SkyDNS that will automatically synchronize information about deployed and running services with a network-local DNS table. This means that without *any* client or server dependencies, you can simply consume a service at a URL like `http://inventory` and your client code will automatically resolve to an appropriate IP address.

When evaluating how you're going to do discovery, you should see if there might be a way to accomplish it without creating a tight coupling or dependency in your application code.

Summary

If you're building microservices today, then you're likely not building just one that runs in isolation. Figuring out a reliable way of allowing one service to be aware of

the URLs and status of all the services on which it depends is no small task. If the idea of dynamic runtime service discovery and of using such discovery to support failover and fault tolerance appeals to you, then a registry like Eureka might be for you.

Keep in mind, though, that a dynamic service registry like Eureka is just one tool among many in the vast arsenal at our disposal these days for building service ecosystems. Hopefully this chapter will have given you an idea of some of the possibilities available and provided enough details for you to decide whether you're going to use discovery with your project.

CHAPTER 3
Configuring Microservice Ecosystems

Configuration is one of the areas of architecture and implementation that are often overlooked by product teams. A lot of teams just assume that the legacy paradigms for configuring applications will work fine in the cloud. Further, it's easy to assume that you'll "just" inject all configuration through environment variables.

Configuration in a microservice ecosystem requires attention to a number of other factors, including:

- Securing read and write access to configuration values
- Ensuring that an audit trail of value changes is available
- Resilience and reliability of the source of configuration information
- Support for large and complex configuration information likely too burdensome to cram into a handful of environment variables
- Determining whether your application needs to respond to live updates or real-time changes in configuration values, and if so, how to provision for that
- Ability to support things like feature flags and complex hierarchies of settings
- Possibly supporting the storage and retrieval of secure (encrypted) information or the encryption keys themselves

Not every team has to worry about all of these things, but this is just a hint as to the complexity of configuration management lying below the surface waiting to strike those who underestimate this problem.

This chapter will begin by talking about the mechanics of using environment variables in an application and illustrate Docker's support for this. Next, we'll explore a configuration server product from the Netflix OSS stack. Finally, we'll dive deeper into `etcd`, an open source distributed key-value store often used for configuration management.

Using Environment Variables with Docker

It is actually fairly easy to work with environment variables and Docker. This book has harped on this point a number of times. Cloud-native applications need to be able to accept configuration through environment variables. While you might accept more robust configuration mechanisms (we'll discuss those shortly), environment variables supplied by the *platform* on which you deploy should the minimal level of configuration support your applications have.

Even if you have a default set of configuration available, you should figure out which settings can be overridden by environment variables at application startup.

You can explicitly set configuration values using name-value pairs as shown in the following command:

```
$ sudo docker run -e SOME_VAR='foo' \
  -e PASSWORD='foo' \
  -e USER='bar' \
  -e DB_NAME='mydb' \
  -p 3000:3000 \
  --name container_name microservices-aspnetcore/image:tag
```

Or, if you want to avoid passing explicit values on the command line, you can *forward* environment variables from the launching environment into the container by simply not passing values or using the equals sign, as shown here:

```
$ docker run -e PORT -e CLIENTSECRET -e CLIENTKEY [...]
```

This will take the `PORT`, `CLIENTSECRET`, and `CLIENTKEY` environment variables from the shell in which the command was run and pass their values into the Docker container without exposing their values on the command line, preventing a potential security vulnerability or leaking of confidential information.

If you have a large number of environment variables to pass into your container, you can give the `docker` command the name of a file that contains name-value pairs:

```
$ docker run --env-file ./myenv.file [...]
```

If you're running a higher-level container orchestration tool like Kubernetes, then you will have access to more elegant ways to manage your environment variables and how they get injected into your containers. With Kubernetes, you can use a concept called `ConfigMap` to make external configuration values available to your containers without having to create complex launch commands or manage bloated start scripts.

A deep dive into container orchestration systems is beyond the scope of this book, but this should reinforce the idea that no matter what your ultimate deployment target is going to be, *all* of them should have some means of injecting environment variables so your application *must* know how to accept those values.

By supporting environment variable injection and sticking with Docker as your unit of immutable artifact deployment, you're well positioned to run in any number of environments without becoming too tightly coupled to any one in particular.

Using Spring Cloud Config Server

One of the biggest difficulties surrounding configuration management for services lies not in the mechanics of injecting values into environment variables, but in the day-to-day maintenance of the values themselves.

How do we know when the ultimate source of truth for the configuration values has changed? How do we know *who* changed them, and how do we implement security controls to prevent these values from being changed by unauthorized personnel and keep the values hidden from those without appropriate access?

Further, if values do change, how do we go back and see what the previous values were? If you're thinking that we could use a solution like a Git repository to manage configuration values, then you're not alone.

The folks who built Spring Cloud Config Server (SCCS) had the same idea. Why reinvent the wheel (security, version control, auditing, etc.) when Git has already solved the problem? Instead they built a service that exposes the values contained in a Git repository through a RESTful API.

This API exposes URLs in the following format:

```
/{application}/{profile}[/{label}]
/{application}-{profile}.yml
/{label}/{application}-{profile}.yml
/{application}-{profile}.properties
/{label}/{application}-{profile}.properties
```

If your application is named `foo`, then all of the `{application}` segments in the preceding template would be replaced with `foo`. To see the configuration values available in the development profile (environment), you would issue a GET request to the `/foo/development` URL.

To find out more about Spring Cloud Config Server, you can start with the documentation.

While the documentation and code are targeted at Java developers, there are plenty of other clients that can talk to SCCS, including a .NET Core client that is part of the Steeltoe project (discussed in the previous chapter).

To add client-side support for SCCS to our .NET Core application, we just need to add a reference to the `Steeltoe.Extensions.Configuration.ConfigServer` NuGet package.

Next, we need to configure our application so it can get the right settings from the right place. This means we need to define a Spring application name and give the URL to the configuration server in our *appsettings.json* file (remember every setting in this file can be overridden with environment variables):

```
{
  "spring": {
    "application": {
      "name": "foo"
    },
    "cloud": {
      "config": {
        "uri": "http://localhost:8888"
      }
    }
  },
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  }
}
```

With this configuration set up, our `Startup` method looks almost exactly like it does in most of our other applications:

```
public Startup(IHostingEnvironment env)
{
   var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json", optional: false,
             reloadOnChange: false)
        .AddEnvironmentVariables()
        .AddConfigServer(env);

   Configuration = builder.Build();
}
```

The next changes required to add support for the configuration server come in the `ConfigureServices` method. First, we call `AddConfigServer`, which enables the client through dependency injection. Next, we call `Configure` with a generic type parameter. This allows us to capture the application's settings as retrieved from the server in an `IOptionsSnapshot`, which is then available for injection into our controllers and other code:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddConfigServer(Configuration);
```

```
        services.AddMvc();

        services.Configure<ConfigServerData>(Configuration);
    }
```

The class we're using here to hold the data from the config server is modeled after the sample configuration that can be found in the Spring Cloud server sample repository:

```
public class ConfigServerData
{
    public string Bar { get; set; }
    public string Foo { get; set; }
    public Info Info { get; set; }

}

public class Info
{
   public string Description { get; set; }
   public string Url { get; set; }
}
```

We can then inject our C# class as well as the configuration server client settings if we need them:

```
public class MyController : Controller
{
   private IOptionsSnapshot<ConfigServerData>
     MyConfiguration { get; set; }

   private ConfigServerClientSettingsOptions
     ConfigServerClientSettingsOptions { get; set; }

   public MyController(IOptionsSnapShot<ConfigServerData> opts,
                       IOptions<ConfigServerClientSettingsOptions>
                          clientOpts)
   {
      ...
   }

    ....
}
```

With this setup in place, and a running configuration server, the `opts` variable in the constructor will contain all of the relevant configuration for our application.

To run the config server, we can build and launch the code from GitHub if we want, but not all of us have a functioning Java/Maven development environment up and running (and some of us simply don't *want* a Java environment). The easiest way to start a configuration server is to just launch it from a Docker image:

```
$ docker run -p 8888:8888 \
 -e SPRING_CLOUD_CONFIG_SERVER_GIT_URI=https://github.com/spring-cloud-samples/ \
 config-repohyness/spring-cloud-config-server
```

This will start the server and point it at the sample GitHub repo mentioned earlier to obtain the "foo" application's configuration properties. If the server is running properly, you should get some meaningful information from the following command:

```
$ curl http://localhost:8888/foo/development
```

With a config server Docker image running locally and the C# code illustrated in this section of the chapter, you should be able to play with exposing external configuration data to your .NET Core microservices.

Before continuing on to the next chapter, you should experiment with the Steeltoe configuration server client sample and then take stock of the options available to you for externalizing configuration.

Configuring Microservices with etcd

Not everyone wants to use the Netflix OSS stack, for a number of reasons. For one, it is noticeably Java-heavy—all of the advanced development in that stack occurs in Java first, and all of the other clients (including C#) are delayed ports of the original. Some developers are fine with this; others may not like it.

Others may also take umbrage with the size of the Spring Cloud Config Server. It is a Spring boot application but it consumes a pretty hefty chunk of memory, and if you're running multiple instances of it to ensure resilience and to prevent any of your applications from failing to obtain configuration, you could end up consuming a lot of the underlying virtual resources just to support configuration.

There is no end to the number of alternatives to Spring Cloud Config Server, but one very popular alternative is `etcd`. As mentioned briefly in the previous chapter, `etcd` is a lightweight, distributed key-value store.

This is where you put the most critical information required to support a distributed system. `etcd` is a clustered product that uses the Raft consensus algorithm to communicate with peers. There are more than 500 projects on GitHub that rely on `etcd`. One of the most common use cases for `etcd` is the storage and retrieval of configuration information and feature flags.

To get started with `etcd`, check out the documentation. You can install a local version of it (it really is a small-footprint server) or you can run it from a Docker image.

Another option is to use a cloud-hosted version. For the sample in this chapter, I went over to compose.io and signed up for a free trial hosting of `etcd` (you *will* have

to supply a credit card, but they won't charge you if you cancel within the trial period).

To work with the key-value hierarchy in `etcd` that resembles a simple folder structure, you're going to need the `etcdctl` command-line utility. This comes for free when you install `etcd`. On a Mac, you can just `brew install etcd` and you'll have access to the tool. Check the documentation for Windows and Linux instructions.

The `etcdctl` command requires you to pass the addresses of the cluster peers as well as the username and password and other options *every time* you run it. To make this far less annoying, I created an alias as follows:

```
$ alias e='etcdctl --no-sync \
   --peers https://portal1934-21.euphoric-etcd-31.host.host.composedb.com:17174,\
   https://portal2016-22.euphoric-etcd-31.host.host.composedb.com:17174 \
  -u root:password'
```

You'll want to change *`root:password`* to something that actually applies to your installation, regardless of whether you're running locally or cloud-hosted.

Now that you've got the alias configured and you have access to a running copy of `etcd`, you can issue some basic commands:

`mk`
: Creates a key and can optionally create directories if you define a deep path for the key.

`set`
: Sets a key's value.

`rm`
: Removes a key.

`ls`
: Queries for a list of subkeys below the parent. In keeping with the filesystem analogy, this works like listing the files in a directory.

`update`
: Updates a key value.

`watch`
: Watches a key for changes to its value.

Armed with a command-line utility, let's issue a few commands:

```
$ e ls /
$ e set myapp/hello world
world
$ e set myapp/rate 12.5
12.5
```

```
$ e ls
/myapp
$ e ls /myapp
/myapp/hello
/myapp/rate
$ e get /myapp/rate
12.5
```

This session first examined the root and saw that there was nothing there. Then, the `myapp/hello` key was created with the value `world` and the `myapp/rate` key was created with the value `12.5`. This implicitly created */myapp* as a parent key/directory. Because of its status as a parent, it didn't have a value.

After running these commands, I refreshed my fancy dashboard on compose.io's website and saw the newly created keys and their values, as shown in Figure 3-1.

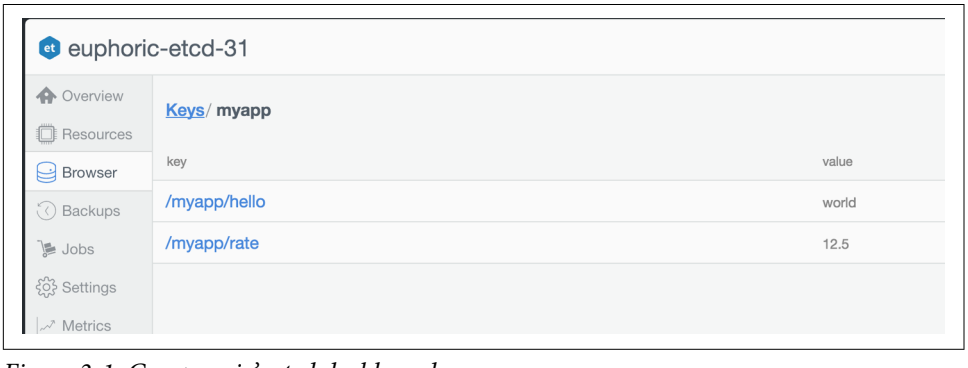


Figure 3-1. Compose.io's etcd dashboard

This is great—we have a configuration server and it has data ready for us to consume—but how are we going to consume it? To do that we're going to create a custom ASP.NET configuration provider.

Creating an etcd Configuration Provider

Throughout the book we've gone through a number of different ways to consume the ASP.NET configuration system. You've seen to how add multiple different configuration sources with the `AddJsonFile` and `AddEnvironmentVariables` methods.

Our goal now is to add an `AddEtcdConfiguration` method that plugs into a running `etcd` server and grabs values that appear as though they are a native part of the ASP.NET configuration system.

Creating a configuration source

The first thing we need to do is add a configuration source. The job of a configuration source is to create an instance of a configuration builder. Thankfully these are pretty

simple interfaces and there's already a starter `ConfigurationBuilder` class for us to build upon.

Here's the new configuration source:

```
using System;
using Microsoft.Extensions.Configuration;

namespace ConfigClient
{
    public class EtcdConfigurationSource : IConfigurationSource
    {
        public EtcdConnectionOptions Options { get; set; }

        public EtcdConfigurationSource(
          EtcdConnectionOptions options)
        {
            this.Options = options;
        }

        public IConfigurationProvider Build(
          IConfigurationBuilder builder)
        {
            return new EtcdConfigurationProvider(this);
        }
    }
}
```

There is some basic amount of information that we'll need in order to communicate with `etcd`. You'll recognize this information as mostly the same values we supplied to the CLI earlier:

```
public class EtcdConnectionOptions
{
    public string[] Urls { get; set; }
    public string Username { get; set; }
    public string Password { get; set; }
    public string RootKey { get; set; }
}
```

Creating a configuration builder

Next we can create a configuration builder. The base class from which we'll inherit maintains a protected dictionary called `Data` that stores simple key-value pairs. This is convenient for a sample, so we'll use that now. More advanced configuration providers for `etcd` would probably want the flexibility of maybe splitting keys on the `/` character and building a hierarchy of configuration sections, so `/myapp/rate` would become `myapp:rate` (nested sections) rather than a single section named `/myapp/rate`:

```
using System;
using System.Collections.Generic;
using EtcdNet;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Primitives;

namespace ConfigClient
{
    public class EtcdConfigurationProvider : ConfigurationProvider
    {
        private EtcdConfigurationSource source;

        public EtcdConfigurationProvider(
          EtcdConfigurationSource source)
        {
            this.source = source;
        }

        public override void Load()
        {
            EtcdClientOpitions options = new EtcdClientOpitions()
            {
                Urls = source.Options.Urls,
                Username = source.Options.Username,
                Password = source.Options.Password,
                UseProxy = false,
                IgnoreCertificateError = true
            };
            EtcdClient etcdClient = new EtcdClient(options);
            try
            {
                EtcdResponse resp =
                 etcdClient.GetNodeAsync(source.Options.RootKey,
                    recursive: true, sorted: true).Result;
                if (resp.Node.Nodes != null)
                {
                    foreach (var node in resp.Node.Nodes)
                    {
                        // child node
                        Data[node.Key] = node.Value;
                    }
                }
            }
            catch (EtcdCommonException.KeyNotFound)
            {
                // key does not
                Console.WriteLine("key not found exception");
            }
        }
    }
}
```

The important part of this code is highlighted in bold. It calls `GetNodeAsync` and then iterates over a *single* level of child nodes. A production-grade library might recursively sift through an entire tree until it had fetched all values. Each key-value pair retrieved from `etcd` is simply added to the protected `Data` member.

This code uses an open source module available on NuGet called `EtcdNet`. At the time I wrote this book, this was the most stable and reliable of the few I could find that were compatible with .NET Core.

With a simple extension method like this:

```
public static class EtcdStaticExtensions
{
   public static IConfigurationBuilder AddEtcdConfiguration(
       this IConfigurationBuilder builder,
       EtcdConnectionOptions connectionOptions)
   {
       return builder.Add(
        new EtcdConfigurationSource(connectionOptions));
   }
}
```

We can add `etcd` as a configuration source in our `Startup` class:

```
public Startup(IHostingEnvironment env)
{
     var builder = new ConfigurationBuilder()
         .SetBasePath(env.ContentRootPath)
         .AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)
         .AddEtcdConfiguration(new EtcdConnectionOptions
         {
             Urls = new string[] {
             "https://(host1):17174",
             "https://(host2):17174"
              },
             Username = "root",
             Password = "(hidden)",
             RootKey = "/myapp"
         })
         .AddEnvironmentVariables();
     Configuration = builder.Build();
}
```

For obvious reasons, I've snipped out the root password for the instance. Yours will vary depending on how you installed `etcd` or where you're hosting it. If you end up going this route, you'll probably want to "bootstrap" the connection information to the config server with environment variables containing the peer URLs, the username, and the password.

Using the etcd configuration values

There's just one last thing to do, and that's make sure that our application is aware of the values we're getting from the configuration source. To do that, we can add a somewhat dirty hack to the "values" controller you get from the `webapi` scaffolding:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using EtcdNet;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Configuration;

namespace ConfigClient.Controllers
{
    [Route("api/[controller]")]
    public class ValuesController : Controller
    {
        private ILogger logger;

        public ValuesController(ILogger<ValuesController> logger)
        {
            this.logger = logger;
        }

        // GET api/values
        [HttpGet]
        public IEnumerable<string> Get()
        {
            List<string> values = new List<string>();
            values.Add(
              Startup.Configuration.GetSection("/myapp/hello").Value);
            values.Add(
              Startup.Configuration.GetSection("/myapp/rate").Value);

            return values;
        }

      // ... snip ...
    }
}
```

To keep the code listing down I snipped out the rest of the boilerplate from the values controller. With a reference to `EtcdNet` in the project's *.csproj* file, you can `dotnet restore` and then `dotnet run` the application.

Hitting the `http://localhost:3000/api/values` endpoint now returns these values:

```
["world","12.5"]
```

These are the exact values that we added to our `etcd` server earlier in the section. With just a handful of lines of code we were able to add a rock-solid, remote configuration server as a standards-conforming ASP.NET configuration source!

Summary

There are a million different ways to solve the problem of configuring microservices, and this chapter only showed you a small sample of these. While you're free to chose whichever you like, keep a close eye on how you're going to maintain your configuration *after* your application is up and running in production. Do you need audit controls, security, revision history, and other Git-like features?

Your platform might come with its own way of helping you inject and manage configuration, but the single most important lesson to learn from this chapter is that every single application and service you build *must* be able to accept external configuration through environment variables, and anything more complicated than a handful of environment variables is likely going to require some kind of external, out-of-process configuration management service.

CHAPTER 4

Securing Applications and Microservices

Developers' perception of security concerns can range from true love to pure evil. In some organizations, security is a checklist that happens after an application has been developed, and in others it is such a burden that it often doesn't get done properly or is simply skipped altogether.

When building applications for the cloud—applications built around the assumption that they might not run on infrastructure you own—security *cannot* be an afterthought or some mindless checkbox on a to-do list. Security must be a first-class citizen in all development efforts for user-facing applications and services alike.

In this chapter we'll discuss security topics as they relate to cloud-native applications and develop samples that illustrate some ways we can secure our ASP.NET Core web applications and microservices.

Security in the Cloud

Securing applications that run at scale in the cloud is not as straightforward as it is when you deploy applications to a local data center where you have full control over the operating system and the installation environment.

In this section, we'll cover some of the main issues that developers often run into when trying to adapt their existing ASP.NET skills or legacy codebases to running securely in the cloud. Some of these problems might be obvious (like the lack of Windows authentication), whereas others are more subtle.

Intranet Applications

Intranet applications are everywhere and are often as complex (or more so!) than customer-facing applications. Companies build these support or line-of-business applications all the time, but when we think about building such applications running on a PaaS on top of scalable, cloud infrastructure, some of our old patterns and practices fall short.

The most notable issue is the inability to do *Windows authentication*. ASP.NET developers have been spoiled by a long history of built-in support for securing web applications with Windows credentials. In these applications, the browser-based challenge replies with details about the currently logged-in user, and the server knows how to deal with that information and the user is implicitly logged in. This is extremely effective and very handy for building apps secured against a company's internal Active Directory.

The reason this works is because the client browser and the server application are part of the same domain, or workgroup, or interoperable domains. The presence of Windows on the server and client as well as the presence of Kerberos in the middle facilitate this seamless exchange of credentials.

Whether you're running in a public cloud or your own on-premise PaaS, these platforms operate very differently from traditional physical or virtual machine Windows deployments.

The operating system that underpins your application needs to be considered *ephemeral*. It is subject to periodic and random destruction. You cannot assume that it will have the ability to join a domain; in fact, it is highly unlikely that domain joining will be a practical option. In a lot of cases, the operating systems supporting your cloud applications are frequently and deliberately destroyed. Some companies have security policies where all virtual machines are destroyed and rebuilt during rolling updates to reduce the potential surface area exposed for persistent attacks.

In the case of the code we're writing for this book, we're restricting ourselves to the cross-platform variant of .NET Core, so we can't rely on any facilities that are only available to Windows applications. This rules out integrated Windows authentication, so we'll need to find a different alternative for our cloud services.

Cookie and Forms Authentication

Anyone who has worked with legacy ASP.NET web applications should be familiar with forms authentication. This mode of authentication is where an application presents a custom UI (a form) to prompt the users for their credentials. The credentials are transmitted directly to the application and validated by the application.

When users successfully log in, they receive a cookie that marks them as authenticated for some period of time.

There is nothing about running your application on PaaS that is intrinsically good or bad for cookie authentication. However, it does create a few additional burdens for your application.

First and foremost, forms authentication requires your application to maintain and validate credentials. This means you'll have to deal with securing, encrypting, and storing confidential information. As we'll see later in the chapter, there are other options available that let us defer the maintenance and validation of identity to third parties, allowing our apps to focus solely on their core business value.

Encryption for Apps in the Cloud

Encryption is usually something that we worry about on a per-application basis. Some services use encryption and others don't. In the days of legacy ASP.NET applications, we would find the most common use of encryption in the creation of secure authentication and session cookies.

This form of encryption would make use of the *machine key* in order to encrypt cookies. It would then use the same machine key to decrypt cookies sent to the web application from the browser.

The simple phrase "machine key" should scare us to death as developers of cloud-native services. In the cloud, we can't rely on specific machines or on specific files sitting on those machines. Our application can start up inside any container at any time, hosted by any number of virtual machines on any number of continents. We simply cannot rely on the fact that a single encryption key will be distributed across every machine on which our application runs.

As we'll discuss throughout the rest of the chapter, there are a number of other areas where encryption is used. For example, tokens are often cryptographically signed, requiring the use of asymmetric keys for validation.

Where do you store your keys if you cannot rely on the existence of a persistent filesystem, nor can you rely on those keys being available in the memory of every running instance of your service?

The answer is to treat the storage and maintenance of cryptographic keys as a backing service. In other words, this service is external to your application, in the same way that state, the filesystem, databases, and additional microservices are.

Bearer Tokens

If an application isn't the central authority on the identity and authorization of its users, then it needs to be coded in such a way that it can accept *proof of identity* and

proof of authorization. There are a number of different standards that define various methods for accepting proof of identity, including OAuth and OpenID Connect (usually just referred to as OIDC) that we'll be illustrating in this chapter's samples.

The most common way to transmit proof of identity in an HTTP-friendly, portable manner is through the use of *bearer tokens*. Ideally, when an application accepts a bearer token it does so through the `Authorization` header. The following shows an example of what a bearer token might look like in an HTTP trace:

```
POST /api/service HTTP/1.1
Host: world-domination.io
Authorization: Bearer ABC123HIJABC123HIJABC123HIJ
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (X11; Linux x86_64) etc...etc...etc...
```

The value of the `Authorization` header takes the form of a single word that indicates the type of authorization, followed by some sequence of characters that contains the value of the credentials. You might be more familiar with other commonly used authorization types: `Digest` and `Basic`.

In a normal service flow that utilizes bearer tokens, the service will extract the token from the `Authorization` header. Many token formats, like OAuth 2.0 (JWT), are usually encoded in a Base64, URL-friendly format, so the first step toward validating those tokens is decoding them to get at the original content. If a token was encrypted with a private key, a service will then use a public key to validate that the token was produced by the appropriate authority.

For a detailed discussion of the JSON Web Token (JWT) format and specifications, feel free to check out the original RFC. The code samples we're going to be looking at in this chapter will make extensive use of the JWT format.

Securing ASP.NET Core Web Apps

Securing an ASP.NET Core web application involves deciding on authentication and authorization mechanisms and then using the appropriate middleware. Authentication middleware examines incoming HTTP requests, determines if the user is authenticated, and, if not, issues the appropriate challenge and redirects.

One of the most reliable ways to perform authentication in the cloud and keep your applications as focused on business logic as possible is through the use of bearer tokens.

For the samples in this chapter, we'll be focusing on OpenID Connect and bearer tokens using the JWT standard.

OpenID Connect Primer

Depending on the type of application we're building and the security requirements of that application, we have a wide variety of authentication flows that we can utilize. OpenID Connect (we'll just refer to it as OIDC from now on) is a superset of the OAuth2 standard and contains specifications and standards for the ways identity providers (IDPs), users, and applications communicate securely.

There are authorization flows that are designed specifically for single-page web applications, for mobile applications, and for traditional web applications. One of the simpler flows available for web applications is the one shown in the sequence diagram in Figure 4-1.

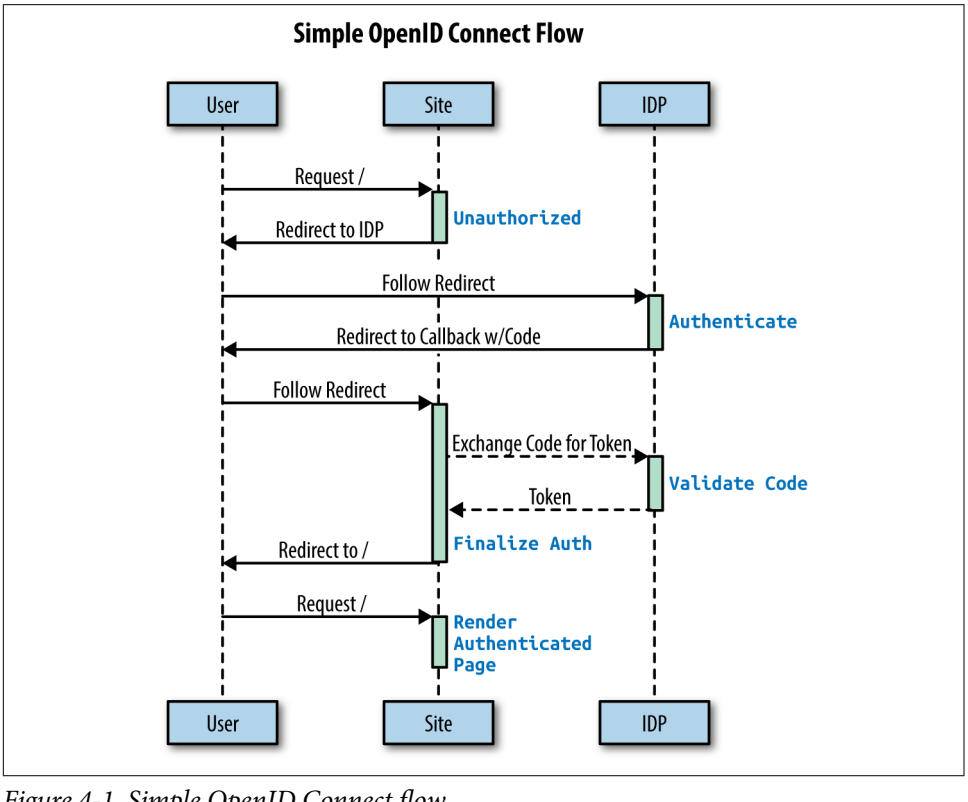


Figure 4-1. Simple OpenID Connect flow

In this flow, an unauthenticated user makes a request for a protected resource on a website. The site then redirects the user to the identity provider, giving it instructions on how to call the site back after authentication. If all goes well, the identity provider will supply to the site a short string with a very short expiration period called a *code*. The site (also referred to as the *protected resource* in this scenario) will then immedi-

ately make an HTTP POST call to the identity provider that includes a client ID, a client secret, and the code. In return, the IDP gives back an OIDC token (in JWT format).

Once the site has received and validated the token, it knows that the user is properly authenticated. The site can then write an authentication cookie and redirect to a home page or the original protected resource. The presence of the cookie is now used so the site can bypass the round-trip to the IDP.

There are more complex flows that include the concept of resources and obtaining access tokens with more chatty redirect loops, but for our sample we're going to be using the simplest flow. This flow is also one of the more secure because the claims-bearing token is never exposed on a URL, only as a short-lived string that is immediately swapped over a secure connection for a token.

OIDC References and Resources

For a more thorough examination of OpenID Connect, its history beginning with the original OAuth standard, and many other security and authentication concepts, I recommend you check out the book *Identity & Data Security for Web Development* by Jonathan LeBlanc and Tim Messerschmidt (O'Reilly). The code samples are in Node.js, but they're easy enough to read and the book is worth the read for the explanations of authentication standards alone.

It might seem a little complex now, but as you get more exposure to OIDC and you see how little code you have to implement (thanks to readily available open source middleware), it becomes less intimidating.

Securing an ASP.NET Core App with OIDC

For our first code sample in this chapter, we're going to take a simple ASP.NET Core MVC web application and secure it with OIDC. To do this, we're going to need a few things:

- An empty web application
- An identity provider
- Some OIDC middleware

Creating an empty web application

The first of the three is easily taken care of with the following command at a terminal:

```
$ dotnet new mvc
```

This creates a starter web application using MVC with a stock controller and some basic layouts, CSS, and JavaScript. We're going to use this as a starting point to add security. If you want to see the completed code sample, you can find it on GitHub.

Setting up an identity provider with an Auth0 account

Now that we have an unsecured application, we need to figure out what we're going to use as our identity provider. In an enterprise situation, we might use something like Active Directory Federation Services (ADFS). If we're already invested in Azure and running an Active Directory there, we might secure our application with Azure AD. We could also use other IDPs, like Ping Federate or Forge Rock. There are plenty of open source samples for standing up super-simple IDPs for experimentation and testing, too.

For the purposes of giving you a sample that you can use without investing in a pile of infrastructure or shelling out a bunch of money in upfront costs, we want an easy-to-use identity provider that includes a free trial period. I decided on Auth0 for this chapter, but other providers are available, like Google and Stormpath (which is merging with Okta). Using Google as an IDP will only accept Google identities, whereas Auth0, Stormpath, and their ilk can be configured to use a private database of users or accept other common OIDC identities, like Facebook and Twitter.

Take a moment now to go over to *http://auth0.com*. Once you've signed up and are at your dashboard, click the Create Client button. Make sure you choose Regular Web Application as the application type. If you choose ASP.NET Core as the implementation language, you will be taken to a quick-start tutorial with code that looks very similar to what we're going to build in this chapter.

Configuring Your Auth0 Client

It is important that you follow all of the directions at the beginning of the Auth0 .NET Core tutorial. This means changing the OAuth JWT signature algorithm under Advanced Settings to RS256.

For the sample client used in this chapter I created a connection to a private database of users, but you can choose what you like, including accepting identities from Facebook or Twitter.

Using the OIDC middleware

Thankfully we have been spared the burden of having to write all of the code that implements the redirects and the other low-level details of the OIDC standard. All we have to do is decide when we want to initiate a challenge (force a user to authenticate against the IDP) and configure the OIDC middleware.

Take a look at the modified *Startup.cs* file from our previously empty web application, shown in Example 4-1. We'll analyze what's going on next.

Example 4-1. src/StatlerWaldorfCorp.SecureWebApp/Startup.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Authentication.OpenIdConnect;
using Microsoft.AspNetCore.Http;


namespace StatlerWaldorfCorp.SecureWebApp
{
    public class Startup
    {
        public Startup(IHostingEnvironment env)
        {
            var builder = new ConfigurationBuilder()
                .SetBasePath(env.ContentRootPath)
                .AddJsonFile("appsettings.json",
                  optional: true, reloadOnChange: false)
                .AddEnvironmentVariables();
            Configuration = builder.Build();
        }

        public IConfigurationRoot Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddAuthentication(
                options => options.SignInScheme =
              CookieAuthenticationDefaults.AuthenticationScheme);
```

```
    // Add framework services.
    services.AddMvc();

    services.AddOptions();

    services.Configure<OpenIDSettings>(
      Configuration.GetSection("OpenID"));
}

public void Configure(IApplicationBuilder app,
            IHostingEnvironment env,
            ILoggerFactory loggerFactory,
            IOptions<OpenIDSettings> openIdSettings)
{
    Console.WriteLine("Using OpenID Auth domain of : " +
      openIdSettings.Value.Domain);
    loggerFactory.AddConsole(
      Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStaticFiles();

    app.UseCookieAuthentication(
      new CookieAuthenticationOptions
      {
          AutomaticAuthenticate = true,
          AutomaticChallenge = true
      });

    var options =
      CreateOpenIdConnectOptions(openIdSettings);
    options.Scope.Clear();
    options.Scope.Add("openid");
    options.Scope.Add("name");
    options.Scope.Add("email");
    options.Scope.Add("picture");

    app.UseOpenIdConnectAuthentication(options);

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
```

```
                    template: "{controller=Home}/{action=Index}/{id?}");
            });
        }

        private OpenIdConnectOptions CreateOpenIdConnectOptions(
            IOptions<OpenIDSettings> openIdSettings)
        {
            return new OpenIdConnectOptions("Auth0")
            {
                Authority =
                  $"https://{openIdSettings.Value.Domain}",
                ClientId = openIdSettings.Value.ClientId,
                ClientSecret = openIdSettings.Value.ClientSecret,
                AutomaticAuthenticate = false,
                AutomaticChallenge = false,

                ResponseType = "code",
                CallbackPath = new PathString("/signin-auth0"),

                ClaimsIssuer = "Auth0",
                SaveTokens = true,
                Events = CreateOpenIdConnectEvents()
            };
        }

        private OpenIdConnectEvents CreateOpenIdConnectEvents()
        {
            return new OpenIdConnectEvents()
            {
                OnTicketReceived = context =>
                {
                  var identity =
                    context.Principal.Identity as ClaimsIdentity;
                  if (identity != null) {
                    if (!context.Principal.HasClaim(
                       c => c.Type == ClaimTypes.Name) &&
                       identity.HasClaim( c => c.Type == "name"))
                         identity.AddClaim(
                           new Claim(ClaimTypes.Name,
                             identity.FindFirst("name").Value));
                    }
                  return Task.FromResult(0);
                }
            };
        }
    }
}
```

The first thing that looks different from the samples in previous chapters is that we're making an options class called `OpenIDSettings` available as a service by reading it

from the configuration system. This is a plain class that just exposes properties for holding the four pieces of metadata needed for every OIDC client:

Authorization domain
: The root hostname of the IDP.

Client ID
: An ID issued by the IDP. You can see this on your client configuration page in Auth0.

Client secret
: There is a button on your Auth0 client configuration page to copy this value to the clipboard.

Callback URL
: This tells the IDP how to redirect the user back to your site after authentication. This value must be configured in the list of authorized callback URLs in your Auth0 client configuration.

Because of the sensitive nature of this information, we haven't checked our *appsettings.json* file into GitHub, but Example 4-2 shows what it looks like.

Example 4-2. appsettings.json

```
{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  },
  "OpenID": {
      "Domain" : "bestbookeverwritten.auth0.com",
      "ClientId" : "<client id>",
      "ClientSecret": "<client secret>",
      "CallbackUrl": "http://localhost:5000/signin-auth0"
  }
}
```

The next two things we do in our new startup class are tell ASP.NET Core that we want cookie authentication and that we want OpenID Connect authentication. Remember that we use OIDC to *determine* who the users are and if they're authenticated, and we'll use cookies to *remember* who they are until the cookies expire.

Another key piece of code is in the `CreateOpenIdConnectEvents` method. Here we define a function that is invoked after we get an authentication ticket back from the

IDP. We use this to sift through the claims on the ticket, and if we find a name claim, we add it to the current claims identity using a well-known constant for the appropriate claim type. This has the effect of translating the OIDC token `name` claim into the `Name` property on the `ClaimsIdentity`. Without this bit of code, we would appear to authenticate but the user's name would be null.

Microsoft Claims Versus OpenID Claims

The issue at hand is that the identity system for ASP.NET Core relies on the `ClaimTypes.Name` constant (`http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name`) to determine the name of the user. However, the name of the claim that corresponds to user name on an OpenID JWT token is simply `name`. Any time we merge OIDC identity and ASP.NET identity, we have to translate claims like this.

If we were to run the application right now, it would appear as though we've done nothing. The application still accepts anonymous authentication and there's nothing that triggers an authentication flow with the Auth0 IDP.

To facilitate this new functionality, we're going to add an account controller as shown in Example 4-3. This controller exposes actions for logging in, logging out, and rendering a view that displays all of the claims on the user identity.

Example 4-3. src/StatlerWaldorfCorp.SecureWebApp.Controllers.AccountController

```
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http.Authentication;
using Microsoft.AspNetCore.Authorization;
using System.Linq;
using System.Security.Claims;

namespace StatlerWaldorfCorp.SecureWebApp.Controllers
{
    public class AccountController : Controller
    {
        public IActionResult Login(string returnUrl = "/")
        {
            return new ChallengeResult("Auth0",
             new AuthenticationProperties() {
              RedirectUri = returnUrl
             });
        }

        [Authorize]
        public IActionResult Logout()
        {
```

```
            HttpContext.Authentication.SignOutAsync("Auth0");
            HttpContext.Authentication.SignOutAsync(
              CookieAuthenticationDefaults.AuthenticationScheme);

            return RedirectToAction("Index", "Home");
        }

        [Authorize]
        public IActionResult Claims()
        {
            ViewData["Title"] = "Claims";
            var identity =
              HttpContext.User.Identity as ClaimsIdentity;
            ViewData["picture"] =
              identity.FindFirst("picture").Value;
            return View();
        }
    }
}
```

Take a look at the code for the `Logout` action. ASP.NET Core supports multiple authentication schemes simultaneously. For our sample, we're supporting both cookies and a scheme called "Auth0" (we could have easily named it something more generic, like "OIDC"). When the user logs out of the application, we want to ensure that awareness of both logins is purged.

There's also a new action called `Claims`. This action searches through the user identity (which is castable to `ClaimsIdentity`) for the claim named `picture`. Once this method finds the `picture` claim, it puts the value in the `ViewData` dictionary.

Varying Support for Claims

Not all IDPs are going to give you a claim called `picture`. Auth0 will give us one if it has been able to figure out the user's picture from the sign-in method (e.g., when users sign in with custom accounts that have email addresses that match registered Gravatars, as in the upcoming example).

Always make sure you get a list of all of the claims guaranteed to be available from your IDP before you write any code that relies on those claims.

Example 4-4 contains the code for the `Claims` view that iterates through the claims collection and renders the claim type and value in a table, as well as displaying the user's picture.

Example 4-4. Claims.cshtml

```
<div class="row">
    <div class="col-md-12">

        <h3>Current User Claims</h3>

        <br/>
        <img src="@ViewData["picture"]" height="64" width="64"/><br/>

        <table class="table">
            <thead>
                <tr>
                    <th>Claim</th><th>Value</th>
                </tr>
            </thead>
            <tbody>
                @foreach (var claim in User.Claims)
                {
                    <tr>
                        <td>@claim.Type</td>
                        <td>@claim.Value</td>
                    </tr>
                }
            </tbody>
        </table>
    </div>
</div>
```

And finally, Figure 4-2 shows what the */Account/Claims* page looks like when I log in to the application with an account bearing my email address.

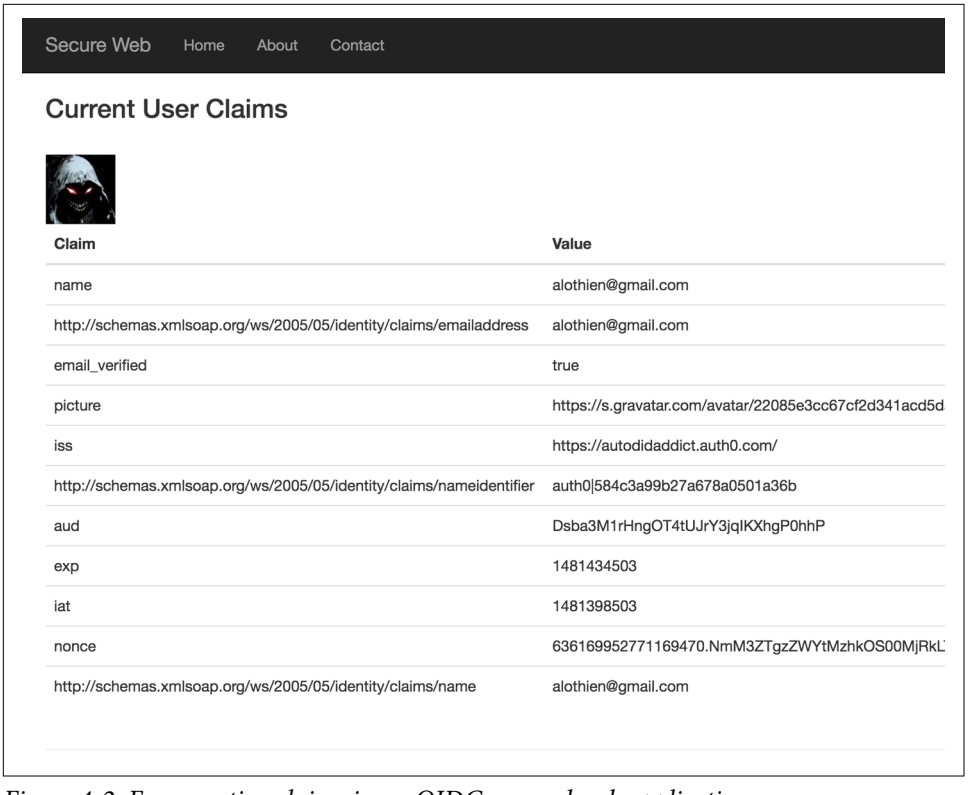


Figure 4-2. Enumerating claims in an OIDC-secured web application

At this point we've gone from the empty scaffolding of an ASP.NET Core web application and connected it to a cloud-friendly third-party identity provider. This relieves our application of the burden of manually managing authentication and allows us to take advantage of bearer tokens and the OIDC standard.

While I normally advocate *avoiding* scaffolding and templates because of their long-storied history of bloat, in this case it wasn't so bad because the template contained stylesheets and layouts we would have had to create anyway.

OIDC Middleware and Cloud Native

I've mentioned a few times how reliance on OS-specific security features will end up causing you a lot of problems in the cloud. There are a number of things that can cause problems when trying to run an application on an elastically scaling platform, and even our shiny new OIDC middleware is subject to some of these issues.

If you're not running this application on Windows, then you might have seen a warning message that looks something like this during startup:

```
warn: Microsoft.Extensions.DependencyInjection.DataProtectionServices[59]
      Neither user profile nor HKLM registry available.
      Using an ephemeral key repository.
       Protected data will be unavailable when application exits.
warn: Microsoft.AspNetCore.DataProtection.Repositories.EphemeralXmlRepository[50]
      Using an in-memory repository.
      Keys will not be persisted to storage.
```

The core of this problem is the use of encryption keys and data protection. In the traditional world of big, bloated Windows servers for .NET applications, we could rely on things like the OS for managing encryption keys.

Imagine that instead of running 1 instance of this application on your laptop, you're running 20 instances of it in the cloud. Unauthenticated users hit instance 1 with no code or token. They get redirected to the IDP, and when they come back to the application they hit instance 2. If information used in the OIDC flow is encrypted by instance 1 and cannot be decrypted by instance 2, you're going to have some catastrophic application failures at runtime.

The solution is to treat the storage and retrieval of security keys as a backing service. There are a number of third-party products, like Vault, that specialize in this functionality, or you can use a distributed cache like Redis and store short-lived keys there.

You've already seen how to use the Steeltoe libraries for application configuration and service discovery when working with the Netflix OSS stack. You can also use a NuGet module from Steeltoe called `Steeltoe.Security.DataProtection.Redis`. This module is designed specifically to move the storage used by the data protection APIs off of the local disk (which is not cloud native) and into an external Redis distributed cache.

Using this library, we can configure external data protection in our `Startup` class's `ConfigureServices` method as follows:

```
services.AddMvc();

services.AddRedisConnectionMultiplexer(Configuration);
services.AddDataProtection()
        .PersistKeysToRedis()
        .SetApplicationName("myapp-redis-keystore");

services.AddDistributedRedisCache(Configuration);

services.AddSession();
```

And then we just call `app.UseSession()` in our `Configure` method to finish setting up external session state.

To see this in action without the rest of the security integration in this chapter, check out the sample in the Steeltoe GitHub repository.

Securing ASP.NET Core Microservices

Securing a microservice with no UI (usually called "headless") automatically rules out any authentication flow that requires direct interaction with a user or a browser capable of facilitating a redirection flow.

In this section, we'll talk about the options available for securing microservices and we'll illustrate one of these options by building a secured service with bearer tokens.

Securing a Service with the Full OIDC Security Flow

One commonly used option for securing a service that supports an OIDC-secured website is to simply implement one of the OIDC authentication flows designed specifically for services.

In this flow, illustrated in Figure 4-3, the user goes through the authentication flow we've already discussed, using browser redirects and interaction with the website and IDP. Once the website establishes a validated claims identity, it will then request an *access token* from the IDP by presenting the identity token as well as information about the desired resource.

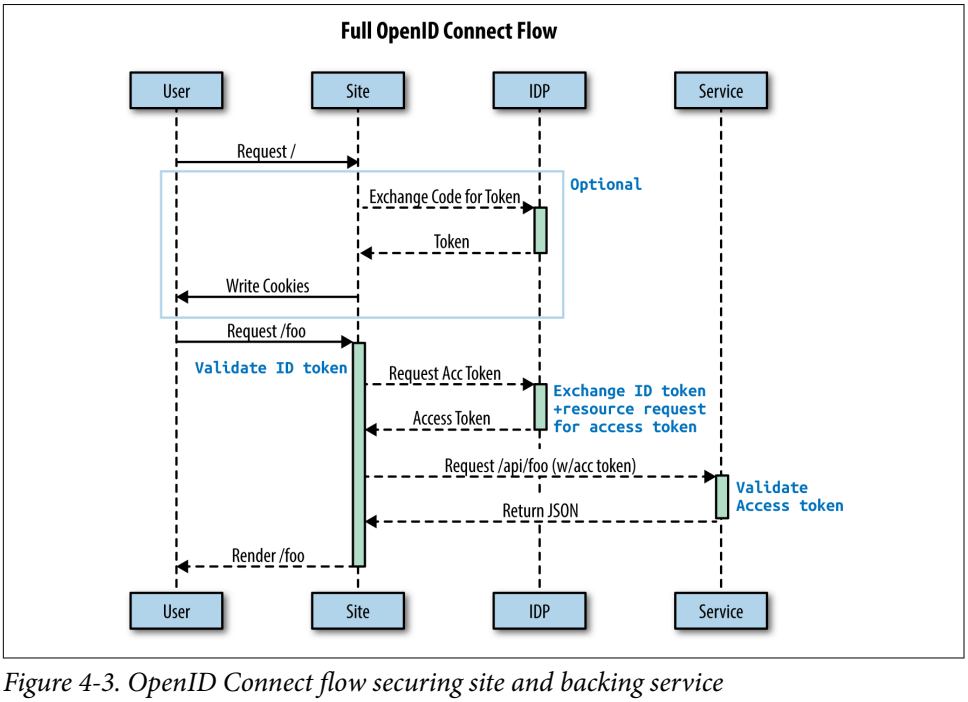


Figure 4-3. OpenID Connect flow securing site and backing service

Essentially, the site will be asking the IDP, "Can user *X* access resource *Y*? If so, give me a token asserting this." The token we get back can be validated by a service. If the

access token presented by the site doesn't grant the user the permission to do what is desired with the resource in question, the service will reject the HTTP call with a 401 Unauthorized or a 403 Forbidden.

If you don't need every single service call to come embedded with the concept of an interactive user as well as whether or not that person has read or write access to the resource, then this authorization flow may be more than you need. A major disadvantage.

The first is that every single access token requires validation. Some of this can be done simply by opening up the token, but in many scenarios the access token is sent directly to the IDP for verification. This makes the IDP an even more integral part of every transaction flowing through your system, which also makes it qualify as a risk and central point of failure.

It's up to you whether you adopt this strategy, but if you just need to know if the consuming application (be it a service or GUI) is permitted to make calls against the backing service API, then there is a much simpler approach that I'll cover next.

Securing a Service with Client Credentials

The client credentials pattern is one of the simplest ways to secure a service. First, you communicate with the service only via SSL, and second, the code consuming the service is responsible for transmitting credentials. These credentials are usually just called a username and password, or, more appropriate for scenarios that don't involve human interaction, a client key and a client secret. Any time you're looking at a public API hosted in the cloud that requires you to supply a client key and secret, you're looking at an implementation of the client credentials pattern.

It is also fairly common to see the client key and secret transmitted in the form of custom HTTP headers that begin with the `X-` prefix; e.g., `X-MyApp-ClientSecret` and `X-MyApp-ClientKey`.

The code to implement this kind of security is actually pretty simple, so we'll skip the sample here. There are, however, a number of downsides to this solution that stem from its simplicity.

For example, what do you do if a particular client starts abusing the system? Can you disable its access? What if a set of clients appear to be attempting a denial of service attack? Can you block all of them? Perhaps the scariest scenario is this: what happens if a client secret and key are compromised and the consumer gains access to confidential information *without* triggering any behavioral alerts that might get them banned?

What we need is something that combines the simplicity of portable credentials that do not require communication with a third party for validation with some of the

more practical security features of OpenID Connect, like validation of issuers, validation of audience (target), expiring tokens, and more.

Securing a Service with Bearer Tokens

Through our exploration of OpenID Connect, we've already seen that the ability to transmit portable, independently verifiable tokens is the key technology underpinning all of its authentication flows.

Bearer tokens, specifically those adhering to the JSON Web Token specification, can also be used independently of OIDC to secure services without involving any browser redirects or the implicit assumption of human consumers.

The OIDC middleware we used earlier in the chapter builds on top of JWT middleware that we get from the `Microsoft.AspNetCore.Authentication.JwtBearer` NuGet package.

To use this middleware to secure our service, we can first create an empty service using any of the previous examples in this book as reference material or scaffolding. Next, we add a reference to the JWT bearer authentication NuGet package.

In our service's `Startup` class, in the `Configure` method we can enable and configure JWT bearer authentication, as shown in Example 4-5.

Example 4-5. Startup.cs

```
app.UseJwtBearerAuthentication(new JwtBearerOptions
{
    AutomaticAuthenticate = true,
    AutomaticChallenge = true,
    TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuerSigningKey = true,
        IssuerSigningKey = signingKey,
        ValidateIssuer = false,
        ValidIssuer = "https://fake.issuer.com",

        ValidateAudience = false,
        ValidAudience = "https://sampleservice.example.com",

        ValidateLifetime = true,
    }
});
```

We can control all of the different types of things we validate when receiving bearer tokens, including the issuer's signing key, the issuer, the audience, and the token lifetime. Validation of things like token lifetime usually also requires us to set up options

like allowing some range to accommodate clock skew between token issuer and the secured service.

In the preceding code, we've got validation turned off for issuer and audience, but these are both fairly simple string comparison checks. When we validate these, the issuer and the audience must match *exactly* the issuer and audience contained in the token.

Let's say our service is a stock management service running in support of a store called *alienshoesfrommars.com*. We might see an issuer value of `https://idp.alienshoesfrommars.com` and the audience would be the service itself, `https://stockservice.fulfillment.alienshoesfrommars.com`. While convention dictates that the issuer and audience are both URLs, they do not need to be live websites that respond to requests in order to validate the token.

You may have noticed that we *are* validating the issuer signing key. This is basically the only way that we can ensure that a bearer token was issued by a known and trusted issuer. To create a signing key that we want to match the one that was used to sign the token, we take a secret key (some string) and create a `SymmetricSecurityKey` from it, as shown here:

```
string SecretKey = "seriouslyneverleavethissittinginyourcode";
SymmetricSecurityKey signingKey =
  new SymmetricSecurityKey(
    Encoding.ASCII.GetBytes(SecretKey));
```

As the string indicates, you should *never store the secret key directly in your code*. This should come from an environment variable or some other external product like Vault or a distributed cache like Redis. An attacker who is able to obtain this secret key will be able to fabricate bearer tokens at will and have unfettered access to your no-longer-secure microservices.

Security and Key Rotation

If the keys used to sign your bearer tokens change every couple of minutes or every hour, then even if someone were to be able to capture this key and fabricate tokens, they would only be able to do so for a short period of time.

Both off-the-shelf and custom-built products usually contain some strategy for keeping a shallow key history around so that validators can check bearer tokens against the current key as well as the previous key, or a consumer retry is built into the client code that fetches a new key upon getting a 401 or 403 from a service.

An intruder gaining access to your system may be inevitable, but using techniques like key rotation, short token expiration periods, and minimal allowances for clock

skew can mitigate the risk or eliminate the damage someone can do with captured keys.

That's pretty much all we need to start securing our services with bearer tokens. All we need to do is drop the `[Authorize]` attribute on controller methods that need this, and the JWT validation middleware will be invoked for those methods. Undecorated methods will allow unauthenticated access by default (though you can change this behavior as well).

To consume our secured service, we can create a simple console application that creates a `JwtSecurityToken` instance from an array of `Claim` objects, then sends those as an `Authorization` header bearer token:

```
 var claims = new[]
 {
    new Claim(JwtRegisteredClaimNames.Sub, "AppUser_Bob"),
    new Claim(JwtRegisteredClaimNames.Jti,
      Guid.NewGuid().ToString()),
    new Claim(JwtRegisteredClaimNames.Iat,
        ToUnixEpochDate(DateTime.Now).ToString(),
        ClaimValueTypes.Integer64),
 };

var jwt = new JwtSecurityToken(
    issuer: "issuer",
    audience: "audience",
    claims: claims,
    notBefore: DateTime.UtcNow,
    expires: DateTime.UtcNow.Add(TimeSpan.FromMinutes(20)),
    signingCredentials: creds);

var encodedJwt = new JwtSecurityTokenHandler().WriteToken(jwt);

httpClient.DefaultRequestHeaders.Authorization =
       new AuthenticationHeaderValue("Bearer", encodedJwt);

var result = httpClient.GetAsync("http://localhost:5000/api/secured").Result;
Console.WriteLine(result.StatusCode);
Console.WriteLine(result.Content.ToString());
```

Here's a secured controller method in our service that enumerates the claims sent by the client. Note that this code would never even be executed if the bearer token hadn't already been validated according to our middleware configuration:

```
[Authorize]
[HttpGet]
public string Get()
{
   foreach (var claim in HttpContext.User.Claims) {
      Console.WriteLine($"{claim.Type}:{claim.Value}");
   }
   return "This is from the super secret area";
}
```

Since the JWT validation middleware has already been written for us, there's very little work for us to do in order to support bearer token security for our services. If we want to have more control over which clients can call which controller methods, we can take advantage of the concept of a *policy*. Policies are just custom bits of code that are executed as predicates while determining authorization.

For example, we could define a policy called `CheeseburgerPolicy` and create a secured controller method that not only requires a valid bearer token, but it also requires that said token meets the criteria defined by the policy:

```
[Authorize( Policy = "CheeseburgerPolicy")]
[HttpGet("policy")]
public string GetWithPolicy()
{
 return "This is from the super secret area w/policy enforcement.";
}
```

Configuring this policy is done easily in the `ConfigureServices` method. In the following sample, we create `CheeseburgerPolicy` as a policy that requires a specific claim (`icanhazcheeseburger`) and value (`true`):

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddMvc();
   services.AddOptions();

   services.AddAuthorization( options => {
     options.AddPolicy("CheeseburgerPolicy",
         policy =>
           policy.RequireClaim("icanhazcheeseburger", "true"));
     });
}
```

Now if we modify our console application to add a new claim of this type with a value of `true`, we will be able to invoke regular secured controller methods as well as controller methods secured with the `CheeseburgerPolicy`.

The policy can require specific claims, usernames, assertions, or roles. You can also define your own requirement that implements `IAuthorizationRequirement` so that you can add your own validation logic while not polluting your individual controllers.

Summary

Security isn't an afterthought. It's also something that cannot be condensed into a single chapter with complete coverage. This chapter provided you with some basic guidance around securing web applications with OIDC and how to secure microservices with JWT bearer token technology.

In both cases the middleware is already written for us, so all we had to do was figure out how to configure and invoke the appropriate code and understand how the various security technologies work. If you have more than a passing interest in identity and secure communications in the cloud, I strongly suggest that you check out some books and online references on the subject.

About the Author

Kevin Hoffman has been programming since he was 10 years old, when he was left alone with a rebuilt Commodore VIC-20 and a BASIC programming manual. Ever since then, he has been addicted to emerging technologies, languages, and platforms.

He has written code for just about every industry, including biometric security, waste management, guidance systems for consumer-grade drones, financial services, and a bunch more. He's written over a dozen books on computer programming and has presented at a number of user groups and conferences, including Apple's WWDC and ScalaDays.

These days Kevin teaches development teams how to migrate and modernize their enterprise applications to thrive in the cloud with the latest cloud-native patterns, practices, and technology. Kevin is currently building cloud-native applications, patterns, and practices for Capital One.