O'REILLY®

Compliments of Pivotal®

Modernizing .NET Applications

A Field Guide for Breathing New Life into Your Software

Richard Seroter

Modernizing .NET Applications

A Field Guide for Breathing New Life Into Your Software

Richard Seroter

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

Modernizing .NET Applications
by Richard Seroter

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*.

Acquisitions Editor: Brian Foster
Production Editor: Nan Barber
Copyeditor: Rachel Monaghan
Proofreader: Octal Publishing, LLC

Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

November 2018: First Edition

Revision History for the First Edition
2018-11-07: First Release

978-1-492-04357-7

[LSI]

Table of Contents

Preface: The .NET Renaissance

.NET is far from dead. Although JavaScript, Go, and Swift have gathered plenty of developer attention, .NET remains a dominant framework. The 2018 StackOverflow Developer Survey polled more than 100,000 developers. In the results, developers said C# was the eighth "most loved" language, and .NET Core was the fifth "most loved" framework. Analyst firm RedMonk looks at GitHub projects and StackOverflow discussion to create its language rankings, and C# has been the fifth most popular language for years now. Companies around the world have major existing investments in .NET, and its popularity remains high.

But it hasn't been entirely smooth sailing. With .NET's coupling to Windows environments, .NET apps haven't had access to the bleeding edge of server automation or application deployment. Configuration management tools have only recently supported Windows in earnest. Public clouds are now making a legitimate effort to woo .NET developers, but that wasn't the case even five years ago. And many of the most exciting microservices patterns have been tougher to implement with the available .NET tools.

This situation has left you with some tough choices. Should you abandon .NET and do your new development in a more open source, Linux-centric language? Should you invest the bare minimum to keep existing .NET apps online but freeze new development? A few years ago, that was a fair concern. However, with the introduction of .NET Core, the availability of new libraries, and some fresh architecture patterns, you have a viable path forward. I'm excited about it. You can confidently build new applications with .NET, while reengaging plans to upgrade the .NET apps you

have. Don't believe me? Let me prove it to you over the course of this book.

Acknowledgments

There are a few folks I'd like to thank for their support on this little endeavor. First, the O'Reilly team has been exceptional. This book is so much better because of their close involvement.

My colleagues at Pivotal are truly best in class and motivate me to do my best work. Our field-facing folks influenced my thinking with all their practical insight into what customers want to accomplish. Our engineering organization includes so many talented people who want to make Windows and .NET great for developers. And I work with the best marketing team on the planet. Special thanks to my terrific boss, Ian Andrews, for always giving me the latitude to take on these crazy projects.

Last but not least, I'm grateful for my supportive family. My wife, Trish, children Noah, Charlotte, and Elliott, and two pups inspire me more than they'll ever know.

CHAPTER 1
Why App Modernization Matters

The top two most-watched cable television networks are Fox News and MSNBC, respectively. In third place is the popular children-oriented network, Nickelodeon. What's the fourth most-watched cable network? Maybe ESPN for sports fans? How about one of those classic television or movie channels? Nope. It's Home and Garden TV (HGTV), a quirky how-to channel focused on home improvement. I'd like to believe its popularity comes from the fact that we humans inherently like to fix things. We often prefer to upgrade something instead of starting over. I think most developers feel the same way about our software. There's value in improving what we have versus walking away.

Okay, but how much does your approach to software really resemble those shows on HGTV? I see at least three relevant parallels.

Stating your goals out loud

One thing I like about these home-improvement shows is that the hosts don't hide their intentions. On *Fixer Upper*, the point is to help couples buy an affordable house and then make the repairs needed for them to truly enjoy it. In *Love It or List It*, one host wants the family to leave their old place behind, whereas the other does the necessary upgrades to try to convince the family to stay put. The host of *Rehab Addict* believes that old homes have a beauty that is brought out with (sometimes major) repairs, and she wants to prove it.

When assessing your software portfolio, it's so important to know what you're after. Say it out loud. Are you purely trying to

save on infrastructure costs? If so, you're likely to choose a different path than if you're looking to make it easier to add functionality. Got issues with performance and scale? I'd bet you'll make architecture choices that wouldn't be necessary if the application were handling the load just fine.

Recognizing that the extent of intrusiveness often correlates with value
: A fresh coat of paint is a good thing. But no one's delusional enough to think that painting the interior of a house will triple its value. It's nice, but superficial. Remodeling a kitchen? That's a different story. To achieve exponential leaps in value, you often have to make intrusive changes.

  The same goes for your apps. Cleaning up the user interface is a terrific thing to do, but rarely does that solve issues of scale, security, or stability. Are you looking to add test coverage to your code, or swap out an enterprise service bus for a lightweight broker? This means getting elbow-deep into the code, but it tends to pay higher dividends.

  This goes back to your goal. If cost savings are what you're after, you don't *need* to generate significant value. Containerize it, throw it in a cloud somewhere, and move on. Now, if you're trying to continuously deliver your app as a way to establish deeper customer loyalty, you'll pursue more extensive avenues that give you that higher order of value. This book focuses on scenarios for which you're after significant value from your existing .NET apps.

Respecting the choices made before you got there, but don't be handcuffed by them
: I find it amusing when an HGTV host goes into an older house and chuckles at outdated carpet or a kitchen that looks like a 1950s sitcom set. But I'd be willing to bet that those decisions made total sense at the time. In that era, with that builder or homeowner, that kitchen was perfect. The new homeowner can recognize that reality, but doesn't hesitate to make the changes needed to make the home more suitable today.

  If you're like me, you often look at your old code and sadly shake your head. *What was I thinking?* However, that code probably represented the best of our skills, our knowledge, and our project demands at the time. Don't apologize for that. Many folks dislike the term *legacy software* because it's used as an

insult. But that legacy software is running your company. Respect the fact that whoever built the first (or second, or third!) version of an app created something that must have been valuable enough to warrant yet another look.

What Is Modernization?

As I said earlier, you need to respect what came before. But just as you would want to update a house to current standards and styles, this book is about modernizing your .NET applications. What do I mean by "modernizing"? Your options with existing applications fall onto a spectrum, as shown in Figure 1-1.

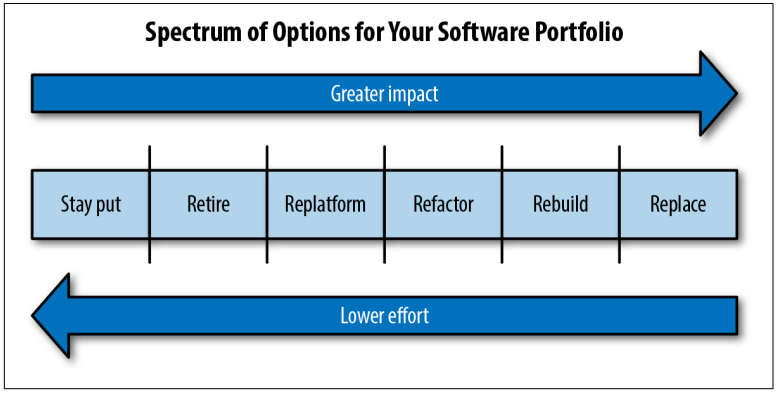

Figure 1-1. The spectrum of options for your existing apps

First, you might choose to *stay put*. That means keeping the application where it currently resides and avoiding changes. Some software might be at the end of its life, so any effort isn't worth it. Or, it's of low business value and your time and attention belong elsewhere.

When I say *replatforming* I mean taking an app, as is, and running it elsewhere. Maybe you're taking your app from a virtual machine to a container. Or from a commercial web server to an open source one. There might be some light code changes, but only the basics required to get the software to run on the next platform.

The next part of the spectrum is *refactoring*. Here, you make light or heavy changes to the code as part of an effort to improve it. You might also swap out major components like database engines or message brokers. Here, you also reconsider previous architectural choices. All of this takes time, but it yields more lasting benefits by

retiring technical debt or increasing the functional capability of the software.

Next, you could choose to *rebuild* the .NET app. The work required to refactor it might be more than it would take to start fresh. There's no shame in committing to a rebuild, but for some teams, it's the first option they consider. It can be difficult to estimate the true level of effort to rebuild an application, and it can be challenging to incrementally deliver value if you're replacing a heavily used system.

Besides the previous options, you can also choose to *retire* or *replace* your .NET application. I'd bet that you have some software that's dutifully maintained but actually no longer relevant. It's important to constantly retire applications that aren't needed anymore. And sometimes, you replace a homegrown, custom-built system with a Software as a Service (SaaS) or commercial one.

For the purposes of this book, when I say "modernization," I'm considering the far end of replatforming, through the refactoring and rebuild stages. That's where core modernization happens.

Why Modernize?

Why go through the effort of changing the apps already in production? If it ain't broke, don't fix it, right? I never liked that saying. Sometimes, the current state seems functional on the surface but is hiding some underlying weakness.

I count at least five reasons that modernization is worth the effort:

Consolidating your environments
: Server sprawl is a real thing. When I was a solution architect and needed hardware, I'd ask for the biggest box I could because I knew getting it resized later was painful. The result? That server sat at 2% utilization. What a waste. I'm sure your (on-premises or cloud) infrastructure is full of underutilized servers, forgotten testing environments, and insecure jump boxes for which everyone knows the password. By modernizing your software, you make it more amenable to high-density, on-demand hosts. That's a cost savings, and it makes you more secure.

Adding new functionality
: I'm not sure about where you work, but I'd think that software exists to, you know…do stuff. And nowadays, that often means new features for users, the ability to handle mobile clients, dynamic integration with other systems or companies, resilience in the face of bursty traffic, and a secure-by-default architecture. A major reason that organizations invest in modernization is to unlock new value from existing assets.

Upgrading and patching your dependencies
: Sun Microsystem's Scott McNealy once quipped that "technology has the shelf life of a banana." I'd be willing to bet that some piece of your software stack requires an update every two weeks. Depending on your architecture and level of automation, it can be impossible to keep up. One goal of modernization is to make it easier to update your components.

Automating more pieces of delivery
: The demands put on you by your company and customers are nearly unsustainable without a commitment to automation. Are you still manually testing the code, running security scans, or double-clicking an MSI to deploy software? That has to change, and modernization affords you the opportunity to inject strategic points of automation into your system.

Reigniting your passion for technology
: We're in the golden age of technology. I believe that. We've never been able to use so little effort to create more impactful experiences with technology. So if you're not enjoying what you're doing right now, one reason could be the slog of dealing with hard-to-maintain systems that everyone depends on. Modernizing your software so that it works for you, not the other way around, is an important reason to make this investment.

What We Cover in This Book

This book is about assessing your existing collection of .NET applications and improving them. We look at improving how you design, build, and run these apps.

In Chapter 2, we'll explore what the current state looks like. It's important to take an inventory of the current state so that we know

what we're working with. Only then can we zero in on the right tactics to employ.

Chapter 3 digs into the modern demands of a .NET software developer. What are you asked to create? When we understand what's demanded of us, we can put our focus on the work that matters.

Chapter 4 explains what "cloud-native" means and why it's an objective for many people embarking on modernization efforts. The cloud-native paradigm offers some crisp criteria that can guide your work.

The introduction of .NET Core represents a watershed moment for .NET developers. But is it suitable for every app? In Chapter 5, we look at where .NET Core came from and how to choose between it and the .NET Framework.

With new paradigms come new patterns and antipatterns. Many of the things that were acceptable on Windows or classic .NET apps are now holding you back. In Chapter 6, we look at what you need to unlearn.

Part of modernization might involve introducing or exchanging some core pieces. Chapter 7 outlines the components you should strongly consider if you're after agility, scalability, and velocity.

You've never had more places to run .NET software. Chapter 8 catalogs all the major options and how you might decide what types of .NET apps run where.

Modernization is about much more than what's in the code. In Chapter 9, we dig into proven modernization strategies that help you decompose monoliths, upgrade your architecture, and much more.

I hope you'll be energized after finishing this book. We close in Chapter 10 with some actionable steps for getting control over your .NET portfolio and extracting new value from your existing assets.

CHAPTER 2

What You Have Running Right Now

In Chapter 1, we established that there are business benefits, technical benefits, and human benefits to running adaptable, maintainable software. Sounds like a place you want to be! But if your company is like most, you're not starting with a blank slate. No, you have years' worth (decades' worth, even) of .NET software that runs your company. That's your starting point. Before we shift to the new-and-exciting aspects of modern .NET, it's important to catalog our current state. Why? It's difficult to clearly see the value of new paradigms unless you recognize the pain of where you are today. And to get a sense for what "better" looks like, you must have a baseline.

In this chapter, we look at a few areas that reflect your (likely) current state. For each, I outline the implications and your motivation to change.

You Have Many Different .NET Project Types

There isn't a one-size-fits-all approach to .NET projects. From the beginning, Microsoft offered different types of projects for each scenario. Need a component? Create a class library. Building a website or web service? Choose from a few options. Building a smart client for the desktop or a background service for the server? Yup, we have those handled, too.

Anyone using .NET for a decade or longer has these types of applications running somewhere:

Windows Forms application
: Run thick-client applications on the desktop. These are user-driven applications built with rich UI controls provided out of the box, bought from a third party, or custom built. These applications were everywhere in the enterprise until high-speed internet became ubiquitous.

Windows Services
: Define long-running background jobs for Windows environments. Many server-side software products installed their components as Windows Services. It was also easy for developers to build these to perform tasks like monitoring file shares for new documents.

Windows Presentation Foundation (WPF) applications
: The successor to Windows Forms. These apps offered a UI framework driven by declarative models written with Extensible Application Markup Language (XAML) and .NET code.

Console applications
: Terminal applications that are often invoked via command lines (with parameters) or other simple tasks that require only text input and output.

ASP.NET Web Forms application/site
: Quickly develop web applications using the original .NET web framework. Based on HTML, server-side controls, and server code, these applications replicated the Windows Forms developer experience. Even though building SOAP web services was easy with this model, it was also quite basic and limiting.

Windows Communication Foundation (WCF) services
: WCF is a hyper-extensible framework for building SOAP and RESTful web services. WCF implemented a variety of WS* standards in an attempt to be a cross-platform framework. Although powerful, WCF led teams down a configuration-heavy, complex path.

ASP.NET MVC applications
: Build web applications and APIs based on the popular model-view-controller design pattern. This open source framework

remains a popular choice and has likely become the default option within your organization.

I didn't even include unsupported Windows Workflow or Silverlight applications in this list, but I'd be willing to bet that you have a handful of those applications stashed somewhere!

What That Means

It's good to have choices. But as a result of evolving choices in the .NET ecosystem, you now have a mishmash of programming models and skill sets throughout your organization. Even though all of the aforementioned are technically supported by Microsoft, you'll struggle to find Windows Forms experience, whereas ASP.NET MVC skills are plentiful in the market.

Why You Want to Change

You could lift and shift some of your classic .NET application types to a new Infrastrucutre as a Service (IaaS) or container environment. However, that doesn't change much for the better. If the app is poorly performing, with a lousy interface and a 10-year-old codebase, clouds or containers won't fix that! Ideally, you consolidate your desired skill set by retiring old programming models and modernize applications so that the transition to cheaper runtimes is easier.

You Have Lots of Windows-Specific Hooks in Your .NET Software

Given that .NET applications ran *only* on Windows for the past two decades, you'd be forgiven for coupling to that operating system.

Without thinking about it, we used capabilities like Integrated Windows Authentication to verify Active Directory users. We added strong-named assemblies to the Global Assembly Cache (GAC) so that everyone on the server could share them. For years, it's been standard to use the ubiquitous Windows Registry for software settings, installation location, and more. And Windows-based software often required drivers or MSI-installed Windows Services to run successfully.

What That Means

Individually, those Windows-specific hooks aren't *bad*, per se. But using them *does* mean that you have some nontrivial refactoring to do if you want to use .NET Core on Linux or are starting to experiment with containers.

Applications with the previous Windows-centric characteristics also tend to be difficult to scale or instantiate on the fly. All that preparation of the operating system means that adding a new server to a cluster isn't simple. And if someone wants a quick development or testing environment built from scratch, there's a lot of friction to making that happen quickly.

Why You Want to Change

Portability and interoperability matter more than ever. When our software is more portable, it's easier to move quickly through development and testing environments into production. Portable apps care less about their host infrastructure, giving you the freedom to use cheaper hosts or alternate operating systems.

Most of the hooks mentioned earlier don't directly affect interoperability. They're local to the software itself, so from the outside looking in, they're implementation details. The exception? Integrated Windows Authentication. In a world in which software runs across Linux and Windows hosts, on mobile devices or full-featured computers, and in public and private infrastructure, cross-platform identity strategies rule. If you're pinned to a Windows-only authentication model, this limits your flexibilty and forces you to add awkward shim solutions.

You Have Stale Environments That Aren't Regularly Updated

Quick—think of a place running your software that hasn't been updated in months or years. It's probably not difficult. I know there are "Hello world" apps of mine running in dozens of locations. Windows Server environments haven't historically been easy to automate, so the whole "treat servers like cattle versus pets" movement passed over many companies. And I'm not just talking about production environments running dusty apps and unpatched operating systems. I'm also referring to the hidden sprawl of sandboxes, per-

formance testing environments, and proof-of-concept clusters. Rarely are your current .NET software or the Windows hosts that run it updated at the rate it should.

What That Means

A stale environment is an insecure one. I'd be willing to bet that something in your software stack becomes vulnerable every two weeks. Unpatched Windows servers are catnip to hackers. Yes, even servers that aren't sitting in your internet-facing DMZ. If someone finds a home for malicious software on a never-changing, lonely server, it might be months or years before you notice.

Let's also not forget about vulnerabilities in your software itself. If you use a lot of external dependencies in your web or smart client apps—admittedly, many classic .NET apps were light on non-Framework dependencies—it's your responsibility to plug those holes regularly. Finally, if you have rarely touched infrastructure and apps, it's likely that you're using long-lived credentials. This also increases your risk exposure.

Why You Want to Change

You have this seemingly impossible task of moving faster while becoming safer. But nobody is satisfied with the status quo. There's that nagging feeling about those old, unpatched servers and instances of software. As we all build more software, we need a different approach. Instead of hoping that you aren't hacked, you want to switch to a proactive stance; one in which you create and destroy servers quickly, patch software constantly, and rotate credentials regularly.

You Have Monolithic Architectures and Complex Deployments

The easiest applications to build are those that have tight coupling. It's *work* to pull out C# from a codebehind page and put it into a remote web service. Adding a message broker introduces complexity. Logging to the local machine's Event Log is *simple*. A lot (most?) of the software I've written in my career has been a monolith. By that, I mean software that is part of one codebase, with collapsed

application tiers (e.g., data access code embedded in user experience tier); that is, compiled, deployed, and scaled as a single unit.

Monoliths aren't inherently awful. Architects like Simon Brown advocate for "modular monoliths," and Martin Fowler makes good arguments for a "monolith first" approach. There are legitimate reasons to avoid the premature optimization of microservices. That said, I suspect that your current monoliths require a delicate series of steps for deployment, which limits how often you do it.

What That Means

If you have a monolithic .NET application, you probably aren't reusing many of its custom components in other applications. It's also likely that you coarsely scale the application. When handling more demand, you're scaling the host for many of the components, even if all the components aren't consuming more resources. It also can be challenging for you to make targeted updates to the software without redeploying the entire thing.

When you have complex monolithic apps, you typically deploy them less often. At least that's my experience. That's because integration testing is more costly, and the installation routines are more intricate.

Why You Want to Change

Who cares if deploying your monolithic .NET app is complicated? You need to do it only once or twice a year! That doesn't reflect the rising demand for regular updates, however. As we established earlier, stale apps and environments are the enemy. They're indicative of increasingly irrelevant software and vulnerable servers. We want to move faster.

By rethinking our architecture, we stand to deploy in smaller batches, run different components on differently sized hosts, scale only the components that need it, and cater to more parallel software development team.

You Have Apps Without Deployment Pipelines

Are you envious of these software-driven companies that can ship software any time they want? What's their secret?

You probably look at the web-scale giants who update their software every 10 minutes and think "That's crazy, we don't need anything like that." It's likely true that your enterprise software or APIs won't have features added daily. But you've seen so far that continually updating .NET apps and infrastructure is about *more* than just features. You want to close security holes, patch software bugs, and more. How do companies ship so often? Deployment pipelines are a big piece of that puzzle.

What That Means

If your apps aren't on a pipeline, they typically consist of pockets of automation (such as continuous integration) intermixed with endless handoffs and manual processes. The problem is that you're only as fast as your limiting factor. If your quality assurance (QA) team has a manual review stage, it barely matters if you automate the steps before and after. If you automate the steps before QA, you'll just pile up work for them. Automating the steps after? Those stages will be starved for work because they can handle much more than upstream processes can hand them.

Even if speed isn't your biggest concern, quality should be something you care about. When your apps are delivered manually, they tend to be more error prone and subject to ad hoc adjustments. This results in inconsistencies that bite you later.

Why You Want to Change

I would contend that ".NET apps on pipeline" is one of the most important metrics you can track. Such an investment in automation is worth it. Consider what happens when you automate your integration testing: developers get faster feedback and have smaller batches of changed code to check when problems arise.

Another benefit? When deployments are automated, it reduces the perceived cost of deployment to zero. This means that teams feel empowered to quickly fix bugs, patch vulnerabilities, perform A/B testing on new feature ideas, and respond to the needs of related teams. Small batches, constantly delivered. That's one of the secrets of the cloud-natives!

You Have Apps That Aren't Ready for Higher-Order Cloud Runtimes

If you have .NET apps created in 2006 or earlier, you probably weren't thinking of cloud computing when designing them! Many first-generation cloud platforms were fairly prescriptive on what could run there, and none of them were Windows-friendly. That might have stunted the introduction of cloud-native patterns to many organizations. A lot of custom-built enterprise .NET apps use design patterns suitable for an on-premises environment that doesn't change quickly or handle unpredictable load.

What That Means

If you have a lot of "traditional" .NET applications in your portfolio, it limits which cloud runtimes make sense. Specifically, apps that don't have cloud-native characteristics aren't a great fit for anything besides virtual machines—basic lift-and-shift exercises. In Chapter 1, we discussed the fact that there's limited value in that effort. Some value, but limited. Ideally, your apps can take advantage of more on-demand services with elastic scale. And if your software has complex deployment routines, a cloud platform won't magically make that easier.

Why You Want to Change

Why do people like you embrace the cloud-computing model? Because agility matters. You gain competitive advantage when you can ship useful technology more often and make it easier to use. Cloud computing ushered in this era of on-demand access to seemingly limitless resources, along with a host of novel services for databases, machine learning, and more. If your app isn't cloud-ready—whether that app is going to a public or private cloud environment—you'll never get more than superficial value.

Summary

Depressed? You shouldn't be! We all start from somewhere, and your existing app portfolio got you to where you are today. Next up, let's examine what you're being asked to create today and what you need to pay attention to as you make your software decisions.

CHAPTER 3

The .NET Software You're Asked to Create

Every company is powered by *people*, not software. Still, what an amazing time to be a software developer! We've never been able to create more powerful experiences all while expending less effort. And it's remarkable to see the role that software plays in every modern business. Pick an industry: education, healthcare, manufacturing, real estate, gaming, finance, retail; you name it. You'll find records systems, marketplace platforms, streaming media, booking systems, and tons more. To satisfy the needs of your company, you're probably running .NET software in local datacenters, colocation facilities, and public clouds. In this chapter, we look at the categories of software you're asked to build today. This exploration matters because it determines what capabilities we need out of our new and modernized software.

Behind-the-Firewall Enterprise Apps

Today, an increasing amount of custom software targets an outside audience. But I'm not seeing a drop-off in demand for new and updated software used by internal staff.

To be sure, just because something isn't internet-accessible doesn't mean it's on your own infrastructure. Enterprise apps might run on-premises, in a colocation facility, or even in a public cloud with isolated networks and a private connection.

Let's talk about the apps themselves. You're getting requests for new standalone apps. Demand comes when teams outgrow their complicated Microsoft Excel spreadsheet solutions, for instance. Or, when new business dictates fresh tools to collect, edit, and share information. A lot of enterprise applications also extend existing commercial software. This might be interfaces that add functionality to an existing Enterprise Resource Planning (ERP) system, mainframe environment, or industry-specific software. Think of a simplified intake form that lets employees add product inventory without accessing the complex back-office system of record.

You're creating all of these enterprise apps in a few different ways. Many of you still crank out desktop apps using Windows Forms or WPF. Web apps are clearly becoming a default option. And there's the workflow-driven custom apps built in "low code" platforms like OutSystems and Pega.

Real-Time Processing Systems

Why all this enterprise attention on developing software? Because we expect the companies that we deal with to make our lives easier. Nowadays, that's through technology. One way companies deliver value through technology is by reacting faster to changing customer needs. This puts a premium on systems that provide up-to-date insight into your business.

Do you want to find out tomorrow that you're out of flu shots in one of your sickest cities? In a hot real estate market, can you afford to discover newly listed homes a week after they went up for sale? What happens if you discover your hotel is grossly overbooked after a partner website sends over a dozen reservations at once?

The heyday of nightly batch processes and weekly data file uploads is over. Your business units are asking for real-time systems that quickly ingest and process data. Now, there's still a place for complex batch analytics. But increasingly, companies use those to complement real-time behavior. For instance, you might generate machine learning models using the rich data in your warehouse. Those models then become accessible to your messaging or event stream–processing engines as new data arrives.

Here's a class of application for which you might find yourself using more bleeding-edge technologies and architecture patterns; use an

event stream processor like Apache Kafka or Amazon Kinesis for high-speed ingest. Or stand up lightweight message brokers like RabbitMQ or NATS to route data to target systems. Maybe you'll introduce TensorFlow to build and train machine learning models. How do people get the results of this real-time processing? Your web applications might use SignalR for server-to-client push scenarios, or you could introduce notification engines like Microsoft Azure Notification Hubs to send mobile alerts. When you start building real-time processing systems, there's a great chance that you'll discover a whole range of new technologies and architectures.

Public-Facing Web Applications

Unlike a decade ago, you're probably spending a fair amount of time building web applications used by *outside* users, whether customers or partners.

For many companies, this represents a big departure from the software they're used to building. Instead of creating internal applications supporting hundreds or thousands of people, you're building internet-facing apps targeting an unpredictable global audience. You're asked to build not only static sites to advertise new brands or one-off promotions, but also full-featured platforms for collecting data, selling products, serving up media, or aggregating information from dozens of sources. And all of this for a population expecting 24×7 availability and split-second latency. Gulp!

The predominant way to build these public-facing web apps today is to rely more heavily on client-side JavaScript rather than stashing compute-intensive logic on the server. Data retrieved from API calls is asynchronously loaded on the page. All of this requires a different architecture for storing, updating, and retrieving data—not to mention the adjustments to web application and API design. But I suspect you'll find yourself creating *more* of these types of applications, not fewer. Now's a great time to absorb and implement modern techniques for these applications.

Mobile-Friendly Solutions

There's no doubt that the mobile experience is crucial for nearly every business today. It's a deciding factor for plenty of consumers when they're choosing banks, grocery stores, airlines, and even

health clubs. The information available at our fingertips is breathtaking. Is your company delivering its key consumer, partner, and employee services via mobile?

When I say "mobile," I consider both native apps and mobile-friendly web experiences. Your stakeholders probably ask for both. Embracing the native application model is powerful but brings with it new considerations for application coding, software delivery, and notification strategy. If you're satisfied offering a web-only experience to customers, you still need to consider low-bandwidth consumers, response time, and spotty connections. Either way, your architecture and toolchain is undergoing a refresh to accommodate this increasingly dominant way of consuming your company's services.

APIs for Internal Apps and Partners

What's powering all of these modern applications—web, streaming, mobile, or otherwise? Application programming interfaces, or APIs. APIs make it simpler for applications to talk to one another. You interact with APIs constantly whether you know it or not. Every time you see an embedded YouTube video or Twitter tweet, post a message to a Slack channel, or read email from an Exchange Server, it's thanks to APIs.

We often associate APIs with public RESTful web services invoked over HTTP. But APIs can be private, use non-HTTP TCP ports, and exchange a variety of payload formats. Heck, the operating system on your computer is *full* of APIs.

Your business stakeholders want more collaborative systems. No software is an island. You could create a tight coupling between application databases, but that hampers your flexibility later on. You could directly embed native API calls to communicate between systems, but again, that's unwanted coupling. No, I'd be willing to bet that you're creating more web service APIs than virtually any other type of software right now. As you construct more decoupled microservices architectures, you invest in well-designed APIs that abstract away the details of the underlying system.

Building web APIs requires a handful of new considerations. Do you need an API gateway for mediation? You know, for things like authorization, token transformation, caching, and rate limiting?

Will you have a database per service, or figure out how to share data stores among various services? Can your web service handle unexpected traffic from mobile clients, partner systems, and internal applications? The web services you built in 2007 are probably in need of a refresh to handle today's demands!

Summary

In this chapter, we looked at the categories of software that your company cares about most right now. Some of those represent things you've been building for years, like internal applications. Others, such as streaming systems, are fairly new in most companies. Is there a set of criteria you can measure against as you consider how to modernize .NET applications to fit into this new world? Yes, and in Chapter 4, we look at what it means to be "cloud-native" and why you should care.

CHAPTER 4

What Does Cloud Native Look Like?

So far, we've looked at what .NET apps you're running today, and what you've been asked to build tomorrow. What is the one constant running through almost every request you now get? Make the software more scalable, more adaptable to change, more tolerant of failure, and more manageable. That's the essence of what it means to be "cloud-native." In this chapter, we look at the ideas behind cloud-native architectures, and why it matters to your .NET applications.

Defining Cloud Native

You'll find many different definitions of cloud native. The charter of the Cloud Native Computing Foundation states that cloud-native systems are "container packed," "dynamically managed," and "microservice oriented." That's too implementation centric for my taste, but its official definition of cloud-native is more on point:

> Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.
>
> These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.

Pivotal uses a definition along those lines:

> Cloud-native is an approach to building and running applications that exploits the advantages of the cloud computing delivery model. Cloud-native is about how applications are created and deployed, not where.

Joe Beda, one of the creators of Kubernetes, takes it a step further:

> At its root, Cloud Native is structuring teams, culture and technology to utilize automation and architectures to manage complexity and unlock velocity.

There's truth in all of these. But what you should take away from these definitions is that cloud native refers to how, not where. It's about achieving better business outcomes through empowered teams that deliver more scalable, resilient, operable software.

Why Cloud-Native Matters

Why should your .NET apps be cloud-native? Does it really matter and is it worth the effort? What it boils down to is getting better at software: designing it, building it, running it.

For the purpose of this discussion, let's equate "good at software" with "delivering cloud-native software." By the end of this chapter, I hope you'll agree with me.

Let's look at five reasons why you need to be good at software.

Customers Expect It

You know what's not cool anymore? Maintenance windows. Or annual software releases. Sluggish performance? Can't have that. No, you expect every business you deal with—whether it's your bank, neighborhood social network, streaming media provider, or your own employer—to deliver digital experiences that are always available, constantly updated, secure by default, and blazingly fast. That's virtually impossible to achieve with software we wrote 10 years ago!

It Helps You Meet the Demands to Operate at Scale

If becoming cloud natives requires us to create more software and machinery to support it, we absolutely need to evolve our approach to operations. Organizations want to flip their spending ratio and invest more on innovation and less on maintaining. Noble goal! That cannot happen without doubling down on automation and reducing toil. And you can't introduce massive amounts of automa-

tion without a culture, team structure, codebase, and platform that accommodates it.

It Gives You More Business Options

When you're good at software, all of a sudden you have fresh opportunities. Telecommunications companies can react quickly to unmet need by reconfiguring mobile data plans. Automobile companies can start ride-sharing or rental services. Manufacturing companies have the choice to sell machine data to third parties. And companies in all sectors can expand into new markets, run quick experiments, and find new revenue streams—all possible, and likely, when you get good at software.

Your Competitors Are Improving

It's rare to find pure monopolies today. Most of us have a choice of who we do business with in most aspects of our lives. This consumer control puts companies on notice: if you don't give me the service I want, I'll switch to someone who will! Your alternative might be a traditional competitor, customer-centric startup, or, increasingly, an internet giant with deep pockets and an eye for expansion. If you don't learn how to deliver valuable software that meets or exceeds expectations, you'll enter an irreversible decline.

It Makes Your Life Better

If you want, ignore all the preceding arguments. If for no other reason, improve your software acumen so that you enjoy work again. This is the best time *ever* to build software. Never have we been able to do so much with so little (relative) effort. If you're miserable at work, something's wrong. Become a cloud native so that you can offload mind-numbing operational tasks and get regular shots of dopamine from seeing people use the software you've built. Ship early and often, and use platforms that "run" the software effectively.

Characteristics of Cloud-Native Apps

Okay, I have you hooked. This all sounds great, you say. Can I just containerize my app and it will become cloud native? No, no you can't. Let's talk about what a cloud-native app *looks* like.

They Meet the 15-factor Criteria

How you *package* your software doesn't make it cloud native. It's about the software itself! The now-famous 12-factor criteria calls out traits for scalable, modern apps. It includes items like explicitly declared dependencies, stateless processes, scale out versus scale up, fast startup and graceful shutdown, and treating logs as event streams. As my former colleague Kevin Hoffman says in his book *Beyond the 12 Factor App* (O'Reilly):

> The goal of these 12 factors was to teach developers how to build cloud-ready applications that had declarative formats for automation and setup, had a clean contract with the underlying operating system, and were dynamically scalable.

Kevin added three factors to the standard list: API-first design, heavy use of telemetry, and security through authentication and authorization. Even though you don't need to blindly adhere to all 15 factors, the more of them that you comply with, the more cloud-ready your software will be.

They're Decoupled and Designed for Change

Microservices: you can't stop hearing about them! Let's ignore the hype. In reality, the point of this architectural paradigm is to decompose hard-to-change monolithic systems. Those decomposed components, or microservices, typically align with a business domain. A microservice isn't defined by how many lines of code it contains, but by its single-purpose focus.

As you decompose systems into these bounded contexts, you get some benefits:

- You now have a smaller deployment surface. Make targeted changes, and deploy each change without bundling up the entire system.
- Microservices help you to scale your teams. Instead of all engineers working on a set of interwoven components, smaller teams can narrow their focus and work on the change cycle that works for them.
- By teasing apart your system, you get the opportunity to introduce new technologies with a limited blast radius. Maybe the entire system doesn't need to use a document database instead

of a relational one, but it makes sense for this particular microservice.

- Microservices add value by supporting smarter deployments. Instead of coarsely scaling the entire system up or down, you have the choice of surgically scaling stressed components. This lets you keep an optimized infrastructure footprint and avoid adding capacity where it isn't needed.

To be fair, a microservices architecture isn't always the answer. You might be better off with a modular monolith, as mentioned in Chapter 2. But if you go down the path of microservices as a way to achieve the cloud agility you're after, there's a lot to consider. We discuss the specifics in upcoming chapters, but you'll have a new series of questions to answer. What's a repeatable way to uncover the boundaries of a service? How do I discover services at runtime? How can I avoid cascading failures? Is my current monitoring strategy set up for an explosion of things to monitor? Where do I start troubleshooting? Stay tuned for the answers.

They're Continuously Delivered

If your software is continuously delivered, you might be a cloud native. Unlike continuous deployment—in which changes are automatically pushed to production—continuous delivery is about going to production whenever you want. You might stage deployments for business reasons, but the current version is ready at any time.

What does it take to get here? A fair bit. It all begins with tests. My Pivotal colleagues laid this out. To go fast (through continuous delivery), you need clean code. Bad code slows you down. To achieve clean code, you need to constantly refactor. To be brave enough to constantly refactor, you need confidence that you won't break your running software. To have confidence, you need tests.

A continuously integrated/continuously delivered (CI/CD) culture affects more than just your software team. It requires buy-in throughout the company. Cloud natives have that. There's an institutional imperative to get value into the hands of customers as quickly as possible. At many enterprises, this is a fundamental shift. It changes how you fund IT, how you arrange teams, what skills you hire for, how marketing delivers the message, and so much more. But this improvement in responsiveness is game-changing for every company that employs it.

They're Built and Run by Empowered Teams

This is where DevOps comes into the picture. And not the watered-down version of DevOps in which you just rename your release engineering team or add some fancy monitoring dashboards. No, I'm talking about a singular focus on customer value. The result? You have an aligned team that includes all the skills needed to design, build, and run the "service" offered to customers. You focus on small batches and regular releases so that you can quickly learn and improve the service. Your teams swarm on production issues, fix issues without spraying blame, and thoughtfully consider how to prevent that issue from happening again.

Cloud natives do this. They don't have a software factory where the work product is handed between silos. They don't have production support teams responsible for dozens of individual systems. And they don't arrange their work around IT projects. There's no doubt that this model represents a change in how most big companies operate today. But the name of the game is "who learns from customers the fastest," and the way to win is to organize and empower your teams to focus on the customer experience.

They're Resilient in the Face of Failure

Everything fails. You can't prevent hardware, networks, software, or facilities from going down or experiencing disruptions. It will happen. Cloud-native apps laugh in the face of failure! They not only *expect* failure, they purposely *inject it* into the system to see what happens.

Cloud natives create software services that stay online in virtually all circumstances. Do they do that by provisioning premium hardware and gold-plated databases? No. Frankly, they do it with commodity hardware and custom-built or open source software. But the key is how they *use* that technology. It's about smart redundancy. They use modern databases (and caches) that tolerate network partitions and scale rapidly. These cloud natives use well-instrumented systems and ubiquitous automation to detect problems and respond immediately. And even if all those things fail, they deploy via automation so that they can stand up cloned environments in short order.

The other resilience angle relates to intentionally trying to break things. Even in production. Chaos engineering is about "experi-

ments to uncover systemic weaknesses." This software engineering discipline is about continuous improvement and recognizing that in complex distributed systems, we need to constantly probe for weaknesses.

Thinking Beyond "Apps" for Cloud-Native Software

Thus far, we've looked at cloud-native *applications*. There's more to the story than that. I often consider at least four other areas where cloud-native comes into play: infrastructure, security, data, and integration. If you leave these out of your strategy, you'll find that you're still experiencing a constraint that limits your velocity and quality.

Can you have *cloud-native infrastructure*? Sure you can. Cloud-native infrastructure, as defined in the book of the same name, is "hidden by useful abstractions, controlled by APIs, managed by software, and has the purpose of running applications." This is about software-controlled infrastructure that results in more consistent provisioning, improved resilience, and simpler maintainability. Using a public cloud doesn't automatically mean that you're using a cloud-native infrastructure approach. Not if you log in to individual machines, build servers via tickets and portals, and colocate all your apps on a few giant servers.

Your existing security strategy might not survive a cloud-native transformation. *Cloud-native security* reflects the fact that you need to "move fast to stay safe," as Pivotal Chief Security Officer Justin Smith likes to say. Malware and advanced persistent threats are evolving faster than ever. Leaked credentials continue to cause major issues. And the monitoring-centric approach isn't good enough. It's time to become more proactive. At Pivotal, we talk about the 3 Rs:

- Quickly *repair* vulnerable software and infrastructure. In the cloud, that might mean being able to patch multiple times per day.
- *Repave* your infrastructure constantly to eliminate hiding places for malware and stay in a consistent, patched state.
- Finally, *rotate* credentials regularly. Shrink the amount of time that credentials are useful. All of these combine to reduce your risk in a cloud-native world.

You won't achieve your desired outcomes if you transform how you build apps but keep the same *data* strategy. You need a *cloud-native data* approach. Your databases and data processing must be biased toward changeabilty, scalability, resilience, and manageability. That means offering different types of databases—relational, key/value, document, graph, caches, and more—for different microservices. When you start having databases scoped to a given microservice, you need to rethink how you provision, update, and manage all these instances. How you collect, transmit, store, and interact will change. Be ready!

After you "solve" the throughput issues related to infrastructure, security, apps, and data, you'll find one more holdout: integration. Much of my career was spent in the application integration space. I saw most companies invest in centers of excellence with expert resources who programmed complex, powerful integration products. The problem? Those teams (and tools) become bottlenecks. If everything can't be continuously delivered, I can move only as fast as my constraint. To rethink how you connect systems together, you'll want to introduce *cloud-native integration*. Wherever possible, integration should be self-service, distributed (not centralized), built to scale, open to changes, and delivered via automation. That's not an easy task, but it's one that will pay real dividends.

Measuring Your Progress Toward Becoming Cloud Native

Are you actually getting better at building software? Are you functioning as a cloud native yet? *How* you answer that question is critical. Measure your progress through outcomes, not output. Just as "lines of code" doesn't mean you're a more productive software developer, neither does "deploys per day" necessarily mean you're improving in the right ways. What is a useful mental model for measuring your progress?

At Pivotal, we talk about 5 S's: speed, stability, scalability, security, and savings. Are you learning and responding faster? That's *speed*. You can measure your improvement in lead time—the time from order/request to final delivery. If you are getting ideas and bug fixes to production faster, that's a tangible thing to measure. Here's one key metric: how many apps are on pipelines. That's an indicator that you can quickly deploy code. *Stability?* Keep an eye on uptime, and

resilience in the face of failures. High performers have a constantly improving mean time to recovery. Customers see less downtime, even if underlying components stumble. Companies observe *scalability* improvement in a few places. Individual systems and services handle increasing traffic with consistently low latency. Measure the time it takes to add or remove capacity in seconds, not weeks or months. What are the right *security* metrics to monitor? Consider how long it takes to patch apps and infrastructure. Or what percentage of apps and infrastructure are 100% up to date on patches. And don't forget about how long your servers live, or how often you cycle credentials.

Finally, if you're good at software, you're going to *save* money. Oh, you might find yourself spending more money because you create new computing environments and write more software. But the cost per unit decreases as you automate infrastructure, deliver work incrementally, and build in security up front.

Summary

There's even more to consider here. For an exceptional look at how to measure the right things in your software transformation, pick up the book *Accelerate* by Nicole Forsgren and team. It will definitely help you focus your attention in the right places. In Chapter 5, we take a closer look at how you choose between the .NET Framework and .NET Core for your cloud-native software.

CHAPTER 5

Choosing Between .NET Framework and .NET Core

I first got my hands on the .NET Framework in July 2000. Some colleagues went to Microsoft's Professional Developers Conference (PDC) and came back with the just-announced technical-preview bits. The installation bricked my laptop, but I wasn't deterred. Thus began a long love affair with one of the most powerful software frameworks ever built. But now things are changing, thanks to .NET Core. Should you still use the .NET Framework for new apps? How about when modernizing them? In this chapter, we dig into those questions.

A Bit of History Regarding the .NET Framework

Travel back with me to the year 1998. Were you building software with Microsoft technologies? First, you could build apps or components with C/C++ and use big Windows-centric pieces like Microsoft Foundation Class Library (MFC) and the Component Object Model (COM). Another option was Visual Basic. Here, you sacrificed low-level control in exchange for a very simple way to build data-driven apps with a nice graphical user interface. Or, you might have started your web development career building Active Server Pages (ASP) with gobs of server-side scripting.

The .NET Framework changed the game. Microsoft introduced the Common Language Runtime (CLR). This application virtual machine executed code compiled down to an Intermediate Language (IL) and offered capabilities like thread management, garbage collection, memory management, and more. Any app written for .NET ran in the CLR.

Software written in multiple languages—including the brand new C#—had shared access to the Framework Class Library (FCL). The FCL (with the Base Class Libraries at its core) offered a unified way for different apps written in different languages to perform data access, web application development, input/output (I/O), network communication, and more. These shared libraries also defined higher-order constructs like ADO.NET, and ASP.NET Windows Forms, and, later on, things like the Windows Communication Foundation (WCF) and the Windows Presentation Foundation (WPF).

Web development took off shortly after the release of the .NET Framework 1.0 in 2002. ASP.NET offered an easy-to-use programming model, and many countries saw the explosion of broadband availability that made rich web applications usable. However, that simple programming model came at a cost. Developers were explicitly shielded from actual web programming. Most of the work was server side, with little JavaScript or HTML interaction. If you needed smart UX components, you bought them from software vendors and baked those controls into your ASP.NET app. ASP.NET Web Services let us expose a SOAP endpoint by adding a basic annotation to a function. As a result, I paid little attention to the HTTP processing pipeline. All of these .NET web applications were also closely intertwined with the Internet Information Services (IIS) web server.

Over the years, the .NET Framework fragmented to support the needs of specific platforms. Each of these platforms—including Silverlight, Windows Phone, and ASP.NET 4—diverged slightly on application model, framework surface, and runtime. Even though the .NET framework was optimized for a given platform, this fragmentation made it tough to build cross-platform systems.

Although all of these design decisions improved developer productivity, they began to become a hindrance as demand increased for

more lightweight, cross-platform applications. The built-in overhead and abstractions were getting in the way.

The Introduction of .NET Core

Announced in 2014 and released in 2016, .NET Core represents a modern take on application development. It's entirely open source. It's modular, lightweight, and compatible with Windows, Linux, and macOS.

Let's dig into the major differences between the .NET Framework and .NET Core:

- First, .NET Core is more modular. Instead of inheriting the entire .NET Framework and massive assemblies like system.web, .NET Core brings in dependencies as NuGet packages. You add only what's needed for a given app. This makes your applications smaller, less memory-hungry, and more amenable to container-based deployment.
- What's also new is the front-and-center use of a command-line interface (CLI) for .NET developers. Although there were CLI tools before, many .NET developers treated the Visual Studio development environment as their gateway to .NET development. No longer. From the .NET Core CLI, you can create projects, add packages, restore dependencies, build projects, run the app, and much more. And with the introduction of Visual Studio Code, Visual Studio for Mac, and JetBrains Rider, .NET developers have a wealth of options for building apps.
- For the web developer, ASP.NET Core ushers in a variety of changes to the programming model, as well. Gone are the *global.asax* and *web.config* files. Now, you have a new approach to middleware, application startup, and patterns like dependency injection. An ASP.NET Core web app is actually a console app that instantiates a web server. The lightweight Kestrel web server is the default option, and you have more control than ever before over the request handling pipeline. These asynchronous pipelines have access to middleware that handles static files, authentication, routing, and more. What happened to the *web.config* file from ASP.NET? An ASP.NET Core app uses an extensible configuration provider that lets you pull in configu-

ration data from a variety of sources—JSON, XML, environment variables, and in-memory .NET objects, for instance.

- Not everything from the .NET Framework made its way into .NET Core. Today, it supports ASP.NET Core (web apps), console applications, class libraries, and Universal Windows Platform (Windows Store) apps. Microsoft announced that .NET Core 3.0 would also support desktop application frameworks like Windows Forms and WPF. But don't hold you breath for classic .NET Framework services like WCF or Windows Workflow Foundation (WF) to make it over. Neither is under active development by Microsoft, and porting to .NET Core seems unlikely.

Deciding Which to Use When Modernizing .NET Apps

Pretty clear cut, eh? Drop everything and get your apps over to .NET Core? It's not that easy.

It appears that .NET Core is the future. To be sure, Microsoft continues to iterate on .NET Framework and dutifully issues updates. It's not a "legacy" platform or scheduled for retirement. It's a robust, mature framework. But Microsoft's focus is clearly on .NET Core and encouraging developers to adopt it as their base framework for all future development.

Microsoft issues some of its own guidance about when you should use each. For example, Microsoft suggests that you should stick with the .NET Framework if your app already uses it, you depend on third-party libraries or technologies that aren't available for .NET Core, or you're using a .NET technology (like WCF or WF) that isn't coming to .NET Core. Conversely, Microsoft recommends .NET Core if you have cross-platform development or deployment needs, you're deploying apps into containers, you want high performance with minimal infrastructure, and you run apps with different versions of .NET on the same machine.

Summary

Don't replatform .NET Framework apps to .NET Core just because .NET Core is new. Do it only because you're getting legiti-

mate value for the effort. Would replatforming to .NET Core enable you to deliver features faster? Would it allow you to use less expensive hardware and operating systems? Does it support the testing and release automation tools that your company depends on for apps written in other programming languages?

Even though your new .NET development might happen in .NET Core, your existing portfolio of apps might only incrementally migrate over. In Chapter 7, we look at the specific recipes for *how* to modernize those applications that benefit from an upgrade.

CHAPTER 6

The New .NET Antipatterns

Out with the old! To modernize .NET applications and get the benefits of cloud native—remember, it's about software that's more scalable, more adaptable to change, more tolerant of failure, and more manageable—we need a different approach. How we built software a decade ago was a reflection of the use cases, technologies, and knowledge we had then. Although we don't apologize for that, all of those aspects have evolved. And quickly! In this chapter, we take a look at a handful of things that we used to do with our .NET apps and why they now represent antipatterns.

These items are "antipatterns" because they make it more difficult for your software to behave in a cloud-native fashion. These patterns are not inherently awful, but they no longer reflect best practices. For each item, I call out why you should avoid it, and what you should do instead.

.NET Application Architecture Antipatterns

These antipatterns directly relate to how you build the functionality of your application. As you modernize your .NET software, these are things that demand refactoring.

In-Process State

ASP.NET makes it super easy to store and retrieve values across user requests. Are you planning on stashing customer-entered data through a multipage wizard experience? Just use the following:

```
Session["CustomerName"] = txtCustomerName.Text;
```

Although ASP.NET offers multiple storage options for session state, the default behavior is to store this information in the host server's memory. Combine this with sticky session routing from your load balancer, and you were good to go.

So what's the problem? Scalability and fault tolerance. When you pin anything to an individual server, you're asking for trouble. What happens if you're in a container that gets shut down? Or what if server instances rapidly scale out to handle load, but your user can't escape the overburdened server on which their session is trapped? Cloud natives store session information in a highly available, off-box database that all application instances share.

Integrated Windows Authentication

I'm guilty of using this a *lot* during my earlier days of coding. For corporate application users, this Windows capability made it easy to create a single sign-on (SSO) experience. Just mix Internet Explorer, Windows desktops, Active Directory, and an IIS web server for free SSO! Although this combo transparently handled the various security handshakes, it also tightly coupled you to Windows environments. At times, it also demanded interactive sessions in which a user needed to key in their credentials.

How is this in conflict with cloud-native principles? It negatively affects software scale and portability. This pattern doesn't work outside of domain-joined Windows Server environments. So there goes any Linux servers in your architecture. It also means that non-.NET applications have a trickier time authenticating and authorizing users in this environment. Microsoft's recommendation for modern web apps? Use OpenID Connect for authentication. You can still use Active Directory as an identity provider if you want, but through OpenID Connect and OAuth 2.0, you take advantage of a standard flow and interface that are usable across operating systems and programming languages.

Using Custom ISAPI Filters or IIS Modules and Handlers

There's no question that Internet Information Server (IIS) for Windows is a powerful web server. For years, Windows-based programmers took advantage of IIS extensibility to augment their web applications. We developed C-based ISAPI filters that acted on indi-

vidual sites, or all sites on the server. These filters changed incoming request data, modified responses, performed custom logging, and much more. In later versions of IIS (starting with 7.0), we could use .NET to build modules, which affect all requests, or handlers, which affect specific request paths or extensions.

Now? Stop creating these. It makes it more difficult to change your software later, and affects manageability. We want software that doesn't require any preconfiguration on the target host. This ensures more consistent, speedy deployments and scaling exercises. Everything our software needs to run is part of the deployment package, and scoped to *just* that deployment. You can't expect that serverwide configurations or ISAPI filters are preinstalled anywhere. Put any HTTP request handling into code that's part of your app. Or, take advantage of the built-in request handling capabilities included in your application platform or service mesh. Just get those capabilities out of the host!

Using the Local Disk for Storage

It's hard to think of an application that doesn't use *some* sort of storage. One straightforward choice for .NET developers is storage that's attached to the host machine. It's just so easy! Every Windows virtual machine has a *C:* drive, at minimum. Heck, the Microsoft documentation for the `File` class has you create a *C:\temp* folder to try out the code. Why *not* use this accessible storage to stash uploaded images or stream out media files?

Using local storage limits your scalability and affects your fault tolerance. In a cloud-native world, hosts are ephemeral. They live for short periods. Treat anything on a local disk as replaceable. Instead of using local storage, switch to a highly available file share or, even better, object storage. Modern object storage—in the public cloud or on-premises—offers HTTP APIs, strong durability, and impressive scale. You can still use local storage, but treat it as a scratch location for temporary content.

Building and Running Windows Services

Chapter 2 discusses the many types of .NET software. Windows Services run as long-running background jobs without a user interface. They often start when the machine starts, and can run in their own security context. These types of apps offer a useful mechanism

for executing scheduled activities—empty out an FTP share every evening—or never-ending processing, such as pulling new purchase orders from a job queue.

Why aren't these friendly to cloud-native architectures? You might find them slowing down your change rate or limiting your manageability. First, like IIS antipatterns, they're Windows specific. Second, Windows Services aren't platform managed; they're managed by the OS. A server manages its Windows Services. In a cloud-native world, platforms manage software. They schedule and monitor the workloads. And finally, Windows Services aren't a truly native part of .NET Core. There are workarounds, but as of this writing, it's not straightforward. Your best bet for background work is to create .NET Framework or .NET Core console applications that are deployed to a platform and easily scale to meet demand.

Leveraging the MSDTC

Back in the day, I spent many hours wrangling with the Microsoft Distributed Transaction Coordinator (MSDTC), a native Windows component for executing two-phase-commit transactions across distributed resources. The capability is tantalizing: create an all-or-nothing operation set that spans databases, message queues, and file-systems. The reality didn't always match the dream. There weren't many supported backend systems, and I inevitably bumped into some strange edge case. Your modern apps might not use MSDTC, but I'd bet that you have some modernization candidates that are drenched in distributed transactions.

So is MSDTC just complicated, or actually an antipattern? It's the latter. It doesn't work with modern (cloud) data sources or messaging technologies and is Windows only. And, most important, it represents a pattern ("distributed transactions") that is counter to the cloud-native focus on scalability and fault tolerance. Transactions are difficult in the first place, but when you begin spanning (long-running) processes and geographies, it becomes a cost-prohibitive approach.

The solution takes us back up to the software design. You want single-responsibility services that might use a synchronous transaction internally, but don't require cross-service transactions. This often requires you to think of individual transactions, and asynchronously handing off to the next step. Your .NET web application

might call a service to charge the customer's credit card, commit that transaction, and then hand off to a service that queues the product for shipping. Those two activities aren't in a single distributed transaction. They commit individually, but with safeguards (e.g., retries, success indicators) to ensure that the overall order process completes.

API Calls That Require User Permission

If you use Windows, you're painfully aware of those User Account Control (UAC) notifications that pop up and ask you to confirm a request for elevated access. This happens when software running on behalf of a standard user wants to do something that requires administrative permission. On the surface, that's fine and a good practice. For server-based software for which no one is there to "approve" the request, though, it's a killer to scalability and software-run-by-software.

You'll want to adopt a model in which services have necessary access to host resources with whatever identity they operate under. Anything that requires administrator intervention is a no-no. Drop any of that .NET code that requests UAC elevation, and ensure that the app permissions are sufficient for anything that needs to be accessed on the host machine or container.

Configuration and Instrumentation Antipatterns

This group of antipatterns relates to how to store configuration data and instrument your applications.

Using web.config for Environment-Specific Values

I have a confession. I used to *love* adding key/value pairs to my ASP.NET *web.config* files. What an easy way to stash configuration settings! I could change connection strings or feature flags without recompiling the code. With a small enough web farm, I might even change these values directly in production. What was a questionable practice 10 years ago, however, is now an obvious no-no.

Putting any editable configuration values into the application package limits manageability and introduces the risk of mismatched configurations among app instances. Anything deployed as part of the

application should be versioned, and any changes should trigger a new deployment. Changing *anything* manually in an environment is a recipe for disaster. If you want the same immutable application package as you deploy between environments, you need to externalize the configuration. That means using environment variables—or even better, an external configuration store—for any values that are environment specific.

machineKey in the machine.config File

The `machineKey` is used in ASP.NET to protect Forms authentication data and view-state data. Every server node in a farm needs the same `machineKey` value. Traditionally, this value is managed via the IIS Manager, and the generated key is stored in the server-wide *machine.config* file.

When deploying .NET software to application platforms, you'll want to do something different. You might not have an IIS Manager experience handy and want to scale in a cloud-native way. This means overriding the `machineKey` value in the *web.config* file. This way, the necessary value is part of the immutable application package and is not dependent on anything on the host server.

Using the Windows Registry or Windows-Specific Logging

The Windows Registry is a database that stores a hierarchy of settings. Windows itself uses the Windows Registry to store settings, and many software packages plant values there. If you build software for Windows, you're also familiar with writing information to the Event Log. All of these represent something that's convenient for development, but a hindrance to scale and manageability.

If your code depends on the Windows Registry, you'll have a more difficult time porting the application to .NET Core. Obviously, the Registry isn't available on a Linux server. Although Windows Containers do offer access to a locally scoped Windows Registry, this database isn't a versioned configuration store, and you shouldn't use it for important values.

If you write application information to the Windows Event Log, you're restricting your manageability. Today's operators don't want to terminal into individual servers to scrape local logs. Rather, your

application should use libraries or platforms that ship logs to a central place. This improves your ability to troubleshoot problems while ensuring that you don't lose valuable data if a host goes away.

Application Dependencies and Deployment Anti-Patterns

This final section reviews anti-patterns to avoid when bundling applications and deploying them to each environment.

Global Assembly Cache Dependencies

The Global Assembly Cache (GAC) is a machine-wide collection of assemblies. When you register an assembly in the GAC, it can be used by any app on the server. Before an assembly can be registered, it must be *strong-named*—that is, signed with a key. The point of strong-naming is to create a unique name for the key and support side-by-side versioning for a given assembly. The GAC was created to eliminate DLL Hell—the classic case in which multiple apps break when a shared component is updated.

Why does the GAC go against your cloud-native principles? A 12-factor app declares its dependencies and includes what it needs to run. You can't assume anything exists on the target server. Any components needed by your application should be a part of your app. Otherwise, you're stuck with complex routines to prepare a server before deploying an instance of your app. You can't autoscale if that's the case!

Interactive Installations

You should use Windows Installers only when deploying to the GAC, according to Microsoft. That's typically an interactive deployment in which an administrator clicks a wizard to choose installation locations and deployment settings. You can have unattended installations, but this is just one example of something that limits you in your quest to become cloud native.

Avoid having any stage of software deployment require human intervention. This means that your deployment environment can't run on anything with an interactive Windows Installer. No software drivers, Windows Services, or Windows extensions that have an installation wizard. Why? Because it means you can't scale on

demand to system-generated environments. Cloud natives use automated platforms to build identical environments and keep them up to date. Ensure that everything your software depends on is bin deployable, and ready to run immediately on any automation-created server.

Summary

In this chapter, we reviewed a handful of antipatterns. Often, the hardest part about using a new technology or paradigm is unlearning what we've been doing for so long. If you've been doing the items listed here, don't feel bad. Many of these patterns were suitable at one point. But now it's time to evolve and adopt patterns that promote scale, fault tolerance, changeability, and manageability.

Chapter 7 focuses on modern libraries and services that help you modernize your .NET apps by introducing new cloud-native patterns to your code.

CHAPTER 7

New Components for Your Modernized .NET Applications

Back in Chapter 1, we talked about those home improvement shows on HGTV. Whenever one of the smiling participants hires an interior decorator to fix up their home, a lot of "stuff" leaves the house, and lots of new "stuff" finds its way in. The house isn't gutted; key pieces of furniture stay, and the structure isn't dramatically altered. But a fresh design warrants new pieces. The same goes for your modernized .NET applications. When refactoring your software, it's an opportune time to freshen up your codebase with components that reflect your new priorities. In this chapter, let's look at what you should consider adding to your modernized apps.

Open Source Data and Messaging Software

Changing your data platforms is scary. I get it. Your databases store years of records and many applications have coalesced around them. Whatever you use for integrating data—be it classic Enterprise Service Bus or commercial Extract, Transform, and Load (ETL) tools, or both—likely forms an integral part of your enterprise architecture. You better have a good reason to swap out these technologies.

In many cases, you do have such a reason. For our modern .NET apps, we're prioritizing flexiblity, portability, and performance, not to mention compatibility with the modern patterns we're implementing. Let's begin with database engines. Classic commercial database platforms are still a good bet. They're powerful and feature-

rich. They're also expensive. As you consider patterns where you have one database per microservice, that cost adds up. Also, you want database engines that you can provision and manage via APIs, thus making developer self-service and platform-managed software a reality.

Consider a few alternatives. For relational workloads, you have a variety of open source options. MySQL is one. Also, take a long look at PostgreSQL. It's a multiplatform, ACID-compliant engine that's proven at scale and quite extensible. PostgreSQL has all the familiar relational database functionality you'd expect, including synchronous and asynchronous replication, indexes, schemas, stored procedures, and triggers. Besides drivers for .NET apps, PostsgreSQL offers connectors for all other major languages.

As part of modernizing your .NET apps, you might be considering some schemaless database engines. MongoDB is a document-oriented database that you should consider. It has field indexing, a rich query syntax, built-in data aggregation functionality, multidocument ACID transactions, and high availablity through replica sets. There's solid .NET support via an official driver.

Redis is another one to look at. It's a remarkably popular in-memory key/value store that's ideal for your caching needs. It has a scalable replication strategy you can employ to scale reads or achieve data redundancy. Like the other open source databases, Redis has a rich set of supported and community-contributed language bindings. Yes, including .NET.

Now, about your messaging systems. Your classic Enterprise Service Bus (ESB) probably doesn't have an expansive API for creating or managing instances. Nor does it lend itself to "citizen integrators" who can quickly deploy their own integrations. Odds are, you have a team of specialists who keep these platforms healthy. Remember one of our key objectives when modernizing our .NET apps: speed. This means eliminating (or automating) anything that slows down the delivery of valuable features to production. If you have to wait days, weeks, or months to get integrations prioritized and shipped, you'll never achieve continuous delivery.

One way to wean yourself off the monolithic integration products is by starting small. Maybe you deploy a lightweight messaging broker for the microservices that power your software. You can introduce something like NATS for lightning-fast, fire-and-forget routing

between services. Or, use the battle-tested RabbitMQ for reliable delivery of business data between your services or systems. And consider event-processing engines like Apache Flink or Apache Kafka when you want a durable, append-only log for your event stream. Each of these technologies work well with .NET applications and offers straightforward interfaces that *every* developer can use.

Cloud-Based Data and Messaging Services

I'd be remiss if I didn't address the fact that many application modernization efforts were sparked by the rapid embrace of the public cloud. You just signed up to use a public cloud and want to avoid just lifting and shifting software over to it. If you want all the goodness of on-demand, scalable infrastructure, you need to make some changes! Fortunately, each public cloud offers some pretty special data services for your modernized .NET apps.

It's easy to find managed relational databases in the public cloud. Amazon Relational Database Service (Amazon RDS0 delivers managed instances of Microsoft SQL Server, MySQL, Oracle databases, and PostgreSQL. Google Cloud SQL supports MySQL and PostgreSQL. Microsoft Azure gives you easy access to Microsoft SQL Server, PostgreSQL, and MySQL. In each case, the cloud provider handles provisioning, configuration, backups, and more. And your .NET code "just works" with any of these because each database service supports its standard interface.

The standardization fades a bit when we look at NoSQL options in the public cloud. This is an area where each provider is innovating on its own, which means your code *must* be refactored to use it. Amazon DynamoDB is a blazing-fast nonrelational database that offers synchronous replication across regions and automates *everything*, including scaling. Azure Cosmos DB has intelligent global replication and easy scaling built in. It also offers multiple types of data models: SQL, MongoDB, Cassandra, Gremlin, and key/value. Both Amazon DynamoDB and Azure Cosmos DB have standard support for .NET apps via dedicated drivers.

If you want to do away with any care-and-feeding of messaging systems, the public cloud has you covered. Amazon Web Services (AWS) offers its SQS platform for basic, durable messaging. Google Cloud has Pub/Sub, and Azure sells its Service Bus. In each case, the engines have extremely high performance limits and simple .NET

interfaces to work with. For event stream processing, you have options like Amazon Kinesis and Azure Event Hubs. One easy way to get started with these is to make them the default choice when you're integrating cloud-hosted services.

Modern .NET Packages

How about your code itself? What should you change there when refreshing your .NET software? There are many exceptional NuGet packages to add, but I'll focus on two that make a big impact: xUnit and Steeltoe—xUnit because unit tests are the foundation of a successful delivery pipeline, and Steeltoe because it accelerates the adoption of cloud-native patterns.

xUnit

To have confidence in your deployments, you must have confidence in your code. To have confidence in your code, you must have confidence in your tests. xUnit is a unit testing framework for .NET Framework and .NET Core. It was created by the gang behind NUnit v2 and is part of the .NET Foundation.

Doing test-driven development can feel like a chore. "Let me just write the code," you might say. But if we're trying to constantly ship value to production, we can't just hand off untested code to a testing team. By thoughtfully designing declarative tests, we improve quality and make it possible to continuously integrate, and even continuously deploy, software.

When writing tests with xUnit, you have two types of unit tests: facts and theories. The xUnit authors say that "*facts* are tests which are always true" and "*theories* are tests which are only true for a particular set of data." So for theories, your tests might pass or fail based on the input data. With xUnit, you can run tests using the .NET Core CLI ("dotnet test") or even Visual Studio. And you can test your code against multiple target platforms—say, .NET Framework 4.7 and .NET Core 2.1—on each run. There's no doubt that it's a real investment to add tests to your code, but it's one that's honestly worth the cost.

Steeltoe

Even though microservices are an exciting way to decompose systems, they add complexity to your architecture. Where once there was a static, predictable landscape, you now have a dynamic environment with more moving parts. Assuming that your app warrants a microservices architecture, you're going to want some help to simplify things. Enter Steeltoe.

Steeltoe is a set of libraries created by Pivotal to bring microservices patterns to your .NET Framework and .NET Core applications. It's inspired by the vigorous microservices support in the ubiquitous Spring Framework, also maintained by Pivotal. .NET apps powered by Steeltoe can run on Windows or Linux, and on Pivotal Cloud Foundry or any application host.

We take a closer look at Steeltoe in Chapter 9, when we apply it to a few modernization recipes, but let's first outline its core capabilities.

Configuration services

.NET Core introduced a new configuration provider model, and Steeltoe takes advantage of that. .NET Core supports configuration sources like command-line arguments, JSON files, and environment variables. Steeltoe adds two more that work with .NET Framework and .NET Core apps: Cloud Foundry and Spring Cloud Config Server. The Cloud Foundry provider parses standard Cloud Foundry environment variables and makes them available to your .NET app. Spring Cloud Config Server makes it easy to serve up configurations stored in Git repos, filesystems, or HashiCorp Vault. The Steeltoe Config Server provider fetches those configurations and makes it easy to access them in your .NET code.

Service discovery

So…where are my microservices? As you scale application instances in and out, and have some healthy ones and some not, it's critically important to have fresh information about where to route a request. Netflix Eureka offers an in-memory database of service locations and responds only with healthy instances. The Steeltoe Discovery client registers your service with the registry and sends occasional heartbeats to the Eureka server to indicate healthiness. The Steeltoe Discovery client also connects your app to the registry, caches the information, and periodically updates its local cache. Your code

refers to a service's friendly name, and relies on the Steeltoe library to exchange that friendly name for a route at runtime.

Circuit breaker

Using the circuit breaker pattern, you prevent hiccups in key services from cascading failure to the rest of the system. You do this by shutting off traffic to the failing service, and providing fallback behavior until that service returns to a healthy state. Steeltoe uses Netflix Hystrix as its implementation. Calls to downstream services are wrapped in a `HystrixCommand`, which works with a fixed thread pool. If the pool is exhausted or too many downstream failures occur, it trips the circuit and triggers a fallback operation. That operation might return cached or static results until the Hystrix component determines that the offending service is back online.

Management endpoints

Observability is a key demand for your modern .NET apps. When something goes wrong, you need to be able to quickly diagnose it. Steeltoe transparently (and optionally) adds a set of powerful management endpoints to your .NET Framework or .NET Core application:

`/health`
: Returns UP or DOWN information based on built-in health contributors, or any custom ones you write.

`/info`
: Returns Git information as well as any app configuration values under the "info" key.

`/loggers`
: Lets you view and change the log level for your .NET applications.

`/trace`
: Returns the last handful of requests to your app, with metadata about the requestor.

`/refresh`
: Triggers a reload of configuration values from configuration sources.

`/env`
: Returns configuration values and keys that your app is using.

`/mappings`
: Returns all the routes exposed by the application.

`/metrics`
: Returns a wide range of CLR, HTTP client, and HTTP server metrics for your app.

`/dump`
: This is for Windows-only environments, and returns information about all the threads used by your application.

`/heapdump`
: This is also Windows-only, and generates a mini-dump of your application for later analysis.

`/cloudfoundry`
: enables integration with the Pivotal Applications Manager UI in Pivotal Cloud Foundry.

Service connectors

One of the 12-factor app criteria refers to bound services. To help you discover and use bound services in a Cloud Foundry environment, Steeltoe offers a handful of connectors. These connectors parse the list of bound services for your .NET app, and provide those connection details to your code. Steeltoe offers connectors for MySQL, PostgreSQL, Microsoft SQL Server, RabbitMQ, Redis, and OAuth.

Security

Steeltoe makes it simple to use Cloud Foundry's OAuth2 security services in your apps. The OAuth2 SSO provider lets you use the credentials in a User Access and Authentication (UAA) server or Pivotal single sign-on (SSO) service for authentication and authorization purposes. For accessing RESTful services, the Steeltoe JSON Web Token (JWT) provider lets you secure access to endpoints.

You can use one or all of the previous capabilities in your modernized .NET apps. Each capability is represented as a NuGet package and typically works with ASP.NET Core, ASP.NET (MVC, Web Forms, WebAPI, WCF), and console applications.

Continuous Integration and Continuous Delivery Tools

What's the most effective way to develop a sustainable path to production? Put your .NET apps on pipelines. This means automating the key steps of integrating, packaging, and deploying software to its target destination. If you do *nothing* else I've recommended in this chapter, at least do this.

There's no shortage of products in this space. There are continuous integration tools like Jenkins, CircleCI, TeamCity, AWS CodePipeline, and Visual Studio Team Services (VSTS) CI. I'm personally partial to Concourse, which offers declarative pipelines, stateless execution environments (so no messy cleanup!), and an intuitive dashboard. When doing continuous delivery, review products like GoCD and Spinnaker, but also consider the CI tools just listed because they're capable of also deploying integrated packages.

Summary

When tasked with modernizing your apps, don't miss this amazing opportunity to actually improve your software. Consider introducing new data services, code libraries, and deployment tools that stand to deliver more resilient, scalable, change-friendly apps that are sustainable for the next decade. Chapter 8takes a look at *where* to run all this modernized .NET software.

CHAPTER 8

Where to Run Your Modern .NET Applications

Have you seen the television show *Love It or List It* on HGTV? The premise is that two hosts compete to see whether they can get the homeowners to renovate and stay in their house ("love it") or fall in love with a new house and sell their current one ("list it"). There are plenty of times when the property owners find that their current abode is still the best fit. But it's not unusual for the new property to win out. Your modernized .NET software might very well keep running on its current host. However, I suspect that you'll frequently want to find it a new home. In this chapter, we look at some considerations for deciding where to run these modern .NET apps.

Choose Your Infrastructure Location

You might be tempted to boil down the infrastructure choice to "public cloud" versus "private cloud." But if you're an enterprise developer, there's more to consider than that.

Consider the full spectrum of hosting options at your disposal. If your company is like most, you're choosing among infrastructure that's on-premises, colocated, run by an outsourcing or managed service provider, or offered by one of many chosen public cloud providers. And your system might span many of those! How do you choose? Consider these six criteria:

Proximity to key systems and data sources

One of your first considerations? Where is my *other* stuff? For example, if you're modernizing a .NET web app with a tangled dependency on a shared, on-premises database, targeting a public cloud might be a mistake. You'd want your app and its database to be close to each other. And what if your system expected all components to be on the same network and your public cloud VLAN couldn't stretch to accommodate? That's a red flag. When modernizing .NET apps, think carefully about how to move the apps *and* their dependencies as a package.

Expected consumption and traffic patterns

What are the general expectations of resource usage and activity of the app? For apps with steady, predictable usage, the public cloud could be more costly and less attractive. Or if you have the need for petabytes of attached storage, it makes sense to use on-premises or colocated infrastructure where you can create massive storage volumes. Unpredictable usage is often where the cloud shines. It's easy to scale up and down, and the seemingly limitless upper bound keeps your app online under nearly every circumstance. Consider what your app is likely to experience, and decide accordingly.

Dev team makeup and skill

Who exactly is building or modernizing your .NET apps? For experienced teams that want to build and run their own software, a true cloud model is ideal. All that self-service and API-centric automation caters to that crowd. If you're not investing a lot in deployment pipelines and infrastructure automation, you might find a managed hosting provider a safer choice. Even though I do believe that your technology choices can trigger a positive change in culture and behavior, I'd also recommend carefully considering your current skill set before choosing a technology for your team.

Who is operating the software?

Do you have very distinct build, deploy, and run stages at your company? If so, that will affect where you choose to run your .NET software. If a separate team operates most of your company's software, you're likely using whatever they offer up in their service catalog. Conversely, if you know that your team is on the hook to build *and* run your software, you'll probably

choose a host that favors self-service and software-driven operations.

Application maturity
: I think we forget about this one too often. When you're experimenting, you prioritize fast feedback, and often do not worry about choosing proprietary technology. Why create the perfect, portable architecture if you don't even know whether the idea is legitimate? Depending on where your app is in its life cycle, that can affect where you run it. Some companies have used a public cloud for shiny new apps in order to measure application traffic patterns, and then eventually moved the app to a cheaper private or hosted environment after they could confidently predict consumption. Or maybe applications that are nearing their end of life get some light refactoring and shoved into a high-density public cloud environment. Although I don't believe that anyone's constantly moving apps between infrastructure hosts, over the 10 to 15 years that software is useful, you can bet that it slides around.

Strategic relationships and existing investments
: You can't ignore your current investments when embarking on a modernization project. If your company is all in on Google Cloud, you're heavily incentivized to run the updated apps there. If you have a multiyear outsourcing agreement, there's heavy pressure to utilize that. And if you have nothing but Windows Server environments (and skills!) in house, it's unlikely you'll replatform onto .NET Core on Linux.

Any one of these—or another criterion entirely—can become a deciding factor. But it's rarely a black-and-white decision, as your company's context plays a major role in what infrastructure hosts your modernized .NET application.

Choose Your Infrastructure Abstraction

Choosing *where* you run software is different than choosing *how* you run software. And your choice of runtime abstraction might actually help determine your infrastructure location, based on what each location supports. Let's look at each runtime abstraction, what it offers you, what you take responsiblity for, and what you should run there.

Hardware Abstraction

Without fail, all your software is running on physical hardware. You might be using virtualized layers on top, but hardware is still at the foundation. What do we get when we directly use physical hardware to run our software? Bare-metal machines provide predictable cost and performance. You pay a certain amount for the hardware, and there's no variable cost based on how you use it. And because you don't have virtualized workloads competing for resources, your software has access to whatever is available on the server.

So what's your responsibility when using physical hardware to run software? A lot. It's up to you to secure and isolate the machines as necessary. You need to install and manage the operating system and any application middleware. If you need additional capacity, it's on you to order and install it. And, of course, you must actually deploy, configure, and operate your software.

There's definitely a class of workload that makes sense to run on bare-metal hardware. Consider software that requires consistent performance and demands physical isolation from others. It's also a good fit when you have substantial compute and local storage demands that can't be met by virtualized infrastructure. You might still have a handful of commercial software packages that aren't supported in virtualized environments and require dedicated hardware. As we discussed earlier, since everything requires a bare-metal foundation, you'll use physical servers to run virtualization platforms.

Infrastructure as a Service Abstraction

This is the abstraction that caused cloud computing to really take off. All of a sudden, Infrastructure as a Service (IaaS) lets you get your own virtual machines anywhere in the world. Amazing! What do you get from this infrastructure abstraction? You get on-demand access to compute. No more long waits to order hardware, rack and stack it, and install an operating system. IaaS also offers elasticity where you can scale individual machines up or down, manually or automatically. This is all made possible by robust APIs that allow unattended, bulk interactions with virtual infrastructure. Finally, IaaS gives you a strong security boundary for colocated workloads that share a physical host or network. You'll find IaaS options from a wide range of public providers as well as on-premises options from VMware and OpenStack distributors.

I've observed that many companies rely on IaaS to run their commercial and custom software. One reason for that is that it's a comfortable transition from how we've traditionally hosted our software. The experience inside an IaaS-provisioned virtual machine is practically identical to a physical host. It's still your responsibility to manage the operating system, connect machinery together—think load balancing, clusters—and configure the application host. Even though IaaS makes it dramatically easier to acquire infrastructure, it doesn't fundamentally change the experience of deploying or running your software. To be sure, there are improvements to the software experience as a result of all the automation and APIs, but you haven't changed your base responsibilities.

IaaS makes for plenty of workloads. It's traditionally been the only place to run Windows-based software. Windows Server support has been slow to arrive to other infrastructure abstractions. It's also suitable for workloads that require preinstalled drivers or operating system configurations. That often applies to virtualized network appliances or commercial software packages. IaaS is also a fit when you're doing lift-and-shift migrations of commercial or custom software running on physical hardware or virtualized environments without automation. And, of course, IaaS forms the foundation of most application platforms that provide higher-level software abstractions.

Container as a Service Abstraction

Container as a Service (CaaS) options arose just as containers became a default packaging unit for modern software. Developers use this abstraction because of its sophistication running containerized workloads. You get smart resource management, scheduling of containers, rapid scaling, and support for both stateful or stateless workloads. Although Kubernetes is seemingly everywhere, there are other legitimate CaaS offerings from Mesosphere, Docker, HashiCorp, and others. CaaS experiences are available in the public cloud via managed services like AWS Elastic Container Service (ECS), or in public/private clouds via software like Pivotal Container Service (PKS). Most container platforms are only for Linux-based workloads, but Microsoft's Service Fabric caters specifically to Windows workloads. And with the introduction of Windows Server Containers, I expect you'll see future Windows support in the other popular CaaS products, too.

If you're investing in a CaaS, you've got certain responsibilities for running your .NET software. It's on you to manage a secure, patched set of base images. Your software gets merged with a parent container image, so it's critical that you use a trusted registry. Other responsibilities include adding log aggregation and app monitoring, given that most CaaS platforms focus on the infrastructure, not any app-specific considerations. Depending on which CaaS you use, you also might need to figure out how to isolate tenants (beyond using constructs like Kubernetes namespaces). And, because software like Kubernetes is updated at least quarterly, it's your responsibility to have a strategy to roll out infrastructure updates.

What should you run in your CaaS environment? Some people answer with "everything!" I don't. To be sure, it's a great runtime for short-lived workloads. Containers are amazing for quick startup and scaling. CaaS is also ideal for software that's prepackaged into containers by vendors, or delivered through the Helm package manager. CaaS is useful for stateful or stateless legacy apps that you don't want to make changes to before stuffing into a high-density runtime. And sure, you also can run Linux-based, custom-built cloud-native software there, if you prefer to operate it at the container level. Given how useful orchestrators like Kubernetes are for infrastructure services, I'm infatuated with CaaS as a runtime for *platforms* more than apps—things like database platforms, event stream–processing platforms, and closely coordinated services that make up your *own* platform.

Platform as a Service Abstraction

The concept of Platform as a Service (PaaS) has been around almost as long as IaaS. In those early days, PaaS was billed as an on-demand, application-centric runtime that hid from you all of the infrastructure complexity. That sounds great! And it *was* for many developers, but there were limits to those early platforms. They ran only in the public cloud. You had to build apps with functional and resource limitations in mind. HTTP was the only routing option. And the runtime was a black box. Some of today's cloud-native application platforms do so much more. Take Pivotal Cloud Foundry (PCF), for instance. You can completely deploy a routable app with a single command and easily scale it. But you can *also* run the platform on any infrastructure, perform TCP-based routing, deploy containers or source code, and SSH into running instances. A qual-

ity PaaS platform also runs stateful or stateless apps, hosts web apps or background jobs, aggregates application logs, and offers a catalog of backing services. All of that comes together to make ongoing app operations simpler.

What's left for *you* to do? In a PaaS, you still define application tenant boundaries—think "orgs" and "spaces" in a PCF environment. It's also your responsibility to define policies for user permissions and autoscaling. And, of course, you still need to deploy code or target your continuously integrated/continuously delivered (CI/CD) process at the PaaS endpoint. Once that code is running, it's on you to manage the application, which could mean scaling it, adding new routes, and troubleshooting issues. If you're running a software-based PaaS like PCF (versus a fully cloud-hosted platform), your team will also operate the platform itself. Fortunately, platforms like PCF come with significant automation capabilities that make it straightforward to operate with a handful of engineers.

So what workloads run in a PaaS? As with CaaS, you could answer "everything!" and I'd disagree. PaaS should be your starting point for most custom-built software. If it can't run there, look for the next abstraction down the stack. You might think, however, that PaaS is just good for web apps. But in reality, it's also a good runtime for batch jobs, streaming data pipelines, and private APIs. And you'll actually find a handful of PaaSes (like PCF or Microsoft Azure's App Service) that have Windows Server support.

Function as a Service Abstraction

How you can have even *less* responsibility for application infrastructure? Use a Function as a Service (FaaS) platform, often synonymous with serverless computing. These services offer a scale-to-zero, pay-for-consumption model that's attractive to developers. Additionally, all FaaS environments offer built-in scaling, so you don't ever need to bother with scaling the function up or down. It's also common to see FaaS products that offer easy integration with cloud services. These cloud services might trigger a function, or be on the receiving end of a function. Some folks are already building complex systems on a FaaS, whereas others are doing simpler but also powerful data change synchronizations, real-time analytics, mobile notifications, and even serving up single-page apps. Today these platforms are almost exclusively in the public cloud, but some new software-based

platforms are cropping up. Keep an eye on offerings like Pivotal Function Service for a functions-anywhere approach.

One thing to keep in mind here is that FaaS is for new software. I'm fairly confident that you don't have any software today that lifts and shifts to a FaaS. The free ride is over. Unlike previous computing paradigms that supported a straightforward migration—consider when you moved software from hardware to VMs, or many apps directly to PaaS or CaaS—FaaS represents a new way of thinking. For FaaS, you're writing tightly scoped, event-driven software that conforms to the boundaries of the FaaS. Decomposing an existing app into a set of functions is far from impossible, but it's definitely a major refactoring.

Summary

It's fairly likely that your infrastructure location and abstraction will undergo changes when you modernize your .NET apps. That's natural. If you build great .NET software with a solid architecture, your infrastructure choices don't need to be made up front. Rather, you might make these choices after you've started your application development. And that's okay. With the exception of functions, these infrastructure options shouldn't exert much influence on your upfront design. In Chapter 9, we look at a handful of modernization recipes that you can use regardless of where your software runs.

CHAPTER 9

Applying Proven Modernization Recipes

Building software is inherently a creative task. In past years, businesses thought they could build "software factories" in which interchangeable people churned out lines of code. That era is thankfully ending, as wise leaders recognize that well-designed, smartly constructed software is a business differentiator. That said, there are software patterns we can follow over and over again to accomplish foundational goals while not diminishing the overall creativity of the software delivery process. In this chapter, we look at five patterns, or recipes, for modernizing your software. By implementing these, you can accomplish many of your cloud-native goals and still maintain the freedom to innovate.

Use Event Storming to Decompose Your Monolith

You have a million-line-of-.NET-code behemoth that runs your business. The thought of deploying an update to this monster makes you break out into a cold sweat. It's chewed up and spit out any developer that tried to make improvements. Today is the day that you slay this foul beast. But how? How do you even begin to tear this creature apart and get it under control? *Event storming* is one effective approach.

Created by Alberto Brandolini, event storming is a technique for creating a domain model. When you're finished, you'll better understand what the software does and how to tackle decomposition.

Event storming is usually done in a workshop with a small group—perhaps 5 to 10 people made up of technical and line-of-business staff. It's led by someone who understands domain-driven-design (DDD). You'll need some office supplies, namely a large surface to write on (often a giant sheet of paper or wall-sized whiteboard), markers, and different colored sticky notes.

You begin by having everyone write down meaningful things that happen in the business domain. Record those *domain events* on one color of sticky note, like orange. Domain events are things like "money withdrawn," "user registered," or "order canceled." Next, you identify the cause of a given event. If that cause is a *command* or trigger, like "register user," it's written on a blue sticky note and placed next to the event. If one event causes another event, place those events next to each other. And if the cause of an event is time itself—for example, consider an event like "cart expired"—you add a note indicating "time." Figure 9-1 shows how this process might look.

Figure 9-1. Team doing an event storming workshop with Pivotal

Next up, you identify *aggregates* that take in commands and result in events, and start grouping aggregates into what's called a *bounded context*. You end up with a behavioral model that drives widespread understanding of the problem space. Developers can take this model and potentially define microservices for a given bounded context. In this manner, you can begin to identify slices of your .NET monolith that you can carve out and rebuild, piece by piece.

Externalize Your Configuration

When I had configuration data for my .NET apps, I typically stored it in keys within my *web.config* file. Oh, how wrong I was. As we discussed earlier, having any configuration values in your .NET project itself means that you need to change the package as it moves between environments and make any production changes by starting back at the source code. Not good. This recipe looks at how you externalize your configuration into remote config stores, by using Steeltoe libraries that talk to a Spring Cloud Config Server.

To begin, go to http://start.spring.io. From here, you can create the scaffolding for Spring Boot projects. Provide a group ID, artifact ID, and name for your project. Next, select the Config Server package (see Figure 9-2).

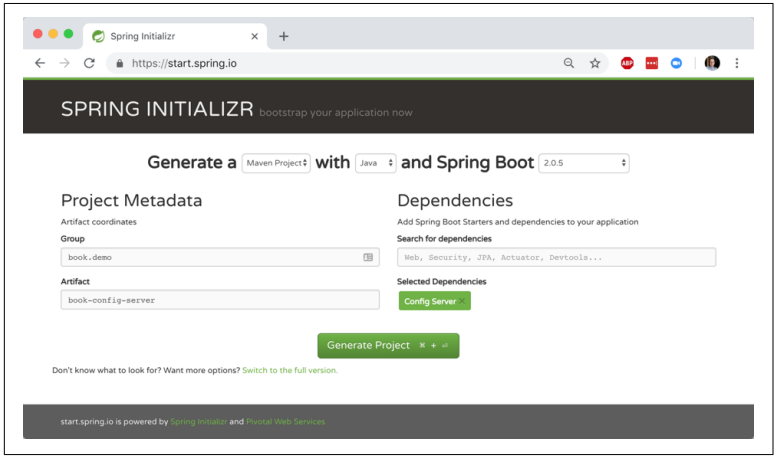

Figure 9-2. Creating the scaffolding for Spring Boot projects at start.spring.io

Download the resulting package. After ensuring that you have a current version of Java on your machine, open this Spring Boot project in your favorite code editor. Eclipse, Atom, and Visual Studio Code all have integration with Spring Tools for easier development.

There's only a single line of code you need to write, thanks to Spring Boot. Open the **.java* file in the *src/main/java* directory, and add the `@EnableConfigServer` annotation above the `@SpringBootApplication` annotation, as shown in Figure 9-3.

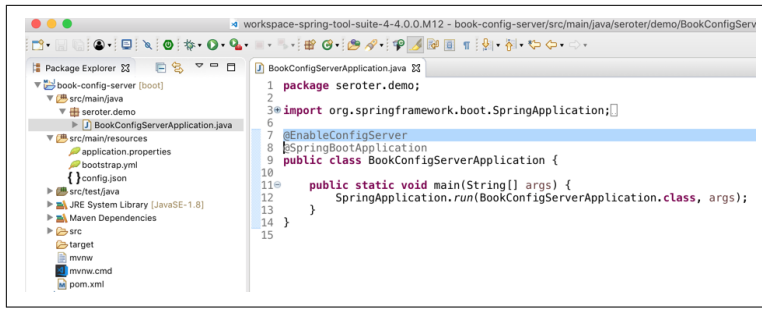

Figure 9-3. Enabling the Spring Cloud Config Server in Spring Boot code

The Spring Cloud Config Server can pull configuration values from a local filesystem, SVN repository, database, Git repository, or HashiCorp Vault backend. Powerful stuff. The Spring Cloud Config Server combines all the relevant configurations it finds into one bucket for your application to access. In the example repo that we're using here, there are three configuration files. One of them is *book-demos.properties*, which contains the following entries:

```
#greeting
greeting=Hello, O'Reilly book readers!

#logging toggles
loglevel=info

server.port=8888

spring.cloud.config.server.git.uri=/
    https://github.com/rseroter/book-demo-configs
```

That completes the Config Server setup. Start the Java project so that the server loads all the configuration files and is ready to serve them to requesting applications, as demonstrated in Figure 9-4.

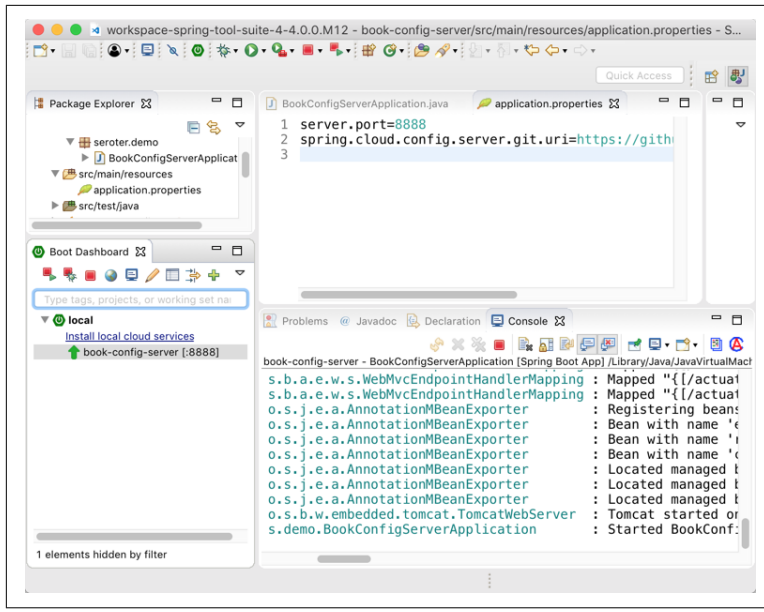

Figure 9-4. Configuring and starting up your Spring Cloud Config Server

Suppose that you have a .NET Framework console app in which you store the configuration values as static variables. You want to replatform that app to .NET Core, and refactor it to grab configuration values from the Spring Cloud Config Server.

With the latest .NET Core SDK installed, go to a command prompt, create a project directory, and run **`dotnet new console`** in that directory. Add the following packages by typing **`dotnet add package`** *`[fill in the package]`*:

- `Microsoft.Extensions.Configuration` to build up the configuration objects in code.
- `Microsoft.Extensions.Configuration.FileExtensions` helps us read from the filesystem to get the location of the Config Server.
- `Microsoft.Extensions.Configuration.Json` helps read the JSON config file on the filesystem.
- `Steeltoe.Extensions.Configuration.ConfigServerCore` pulls in the Steeltoe Config Server client.

Next, add an *appsettings.json* file to the console project. This file points your app to the Config Server and provides a "name," which helps the Config Server client read the correct configuration files. The application name typically maps to the name of the configuration files themselves:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
     }
  },
  "AllowedHosts": "*",
  "spring": {
    "application": {
      "name": "bookdemo"
    },
    "cloud": {
      "config": {
        "uri": "http://localhost:8888"
      }
    }
  }
}
```

Finally, we have our .NET code. Add the following `using` statements to the *Program.cs* file:

```
using System.IO;
using Steeltoe.Extensions.Configuration.ConfigServer;
using Microsoft.Extensions.Configuration;

static void Main(string[] args){
  //retrieve local configurations, and those from the config
  //server
  var builder = new ConfigurationBuilder()
        .SetBasePath(Directory.GetCurrentDirectory())
        .AddJsonFile("appsettings.json")
        .AddConfigServer();
  var configuration = builder.Build();

  string logLevel = configuration["loglevel"];

  Console.WriteLine("value from config is: " + logLevel);
  Console.ReadLine();
}
```

When you run the application, you see that this .NET Core console application now reads configurations from a remote store, as illustrated in Figure 9-5.

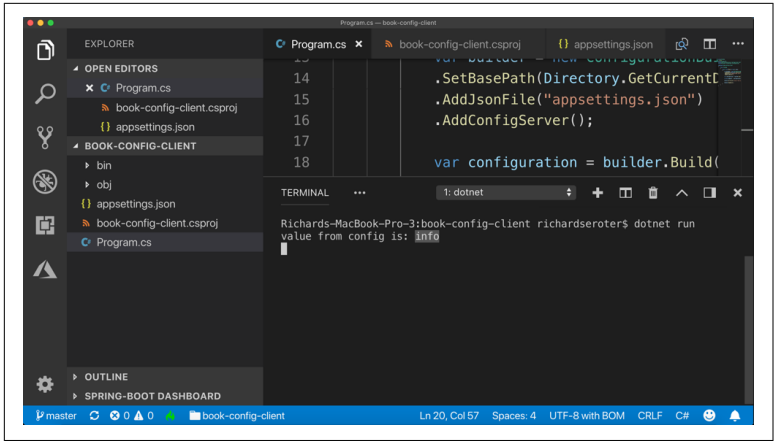

Figure 9-5. Reading configurations from a remote store

You'll also probably use this pattern with ASP.NET and ASP.NET Core web applications, so be sure to read the official Steeltoe documentation for the instructions. And, if you're a Cloud Foundry user, Steeltoe pulls any Config Store connection info directly from the environment. No local config settings needed!

Introduce a Remote Session Store

By default, ASP.NET apps use an in-memory session state provider. You designate that in the *web.config* file:

```
<sessionState mode="InProc"
    customProvider="DefaultSessionProvider">
  <providers>
    <add name="DefaultSessionProvider"
        type="System.Web.Providers.DefaultSessionStateProvider,
        System.Web.Providers, Version=1.0.0.0, Culture=neutral,
        PublicKeyToken=31bf3856ad364e35" connectionStringName
        ="DefaultConnection" />
  </providers>
</sessionState>
```

However, that's not a cloud-native approach. Any loss of host results in lost session data. We want a highly available, remote session store. This recipe looks at how to refactor your web apps to use Redis as that store. Let's assume you have some code like the following that stores a timestamp in ASP.NET session state:

```
string timestamp = Session["timestamp"] as string;
```

```
lblmsg.Text = "timestamp is: " + timestamp;

if (timestamp == null)
{
  timestamp = DateTime.Now.ToString();
  Session["timestamp"] = timestamp;
}
```

Begin by getting Redis running on your machine. You have multiple options here. Redis runs on Unix-based or Windows machines. You can also pull a Docker image instead of installing Redis yourself. Since I'm working with a "classic" Windows app, I used the Chocolatey package manage to install Redis on a Windows Server 2012 R2 machine (Figure 9-6).

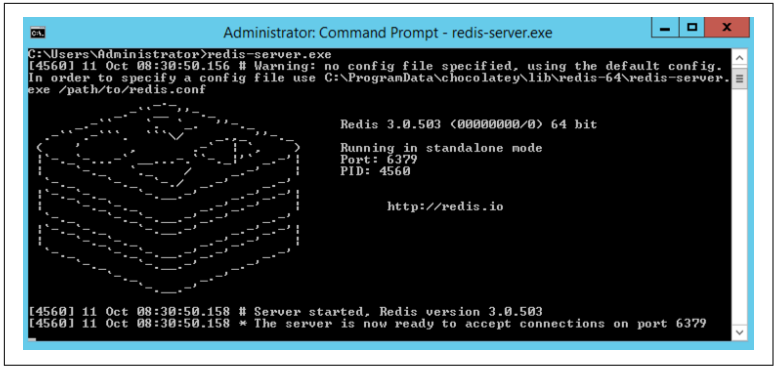

Figure 9-6. Running Redis on Windows Server

Back in our ASP.NET code, add a new NuGet package. Find Microsoft.Web.RedisSessionStateProvider (Figure 9-7). I chose version 2.2.6.

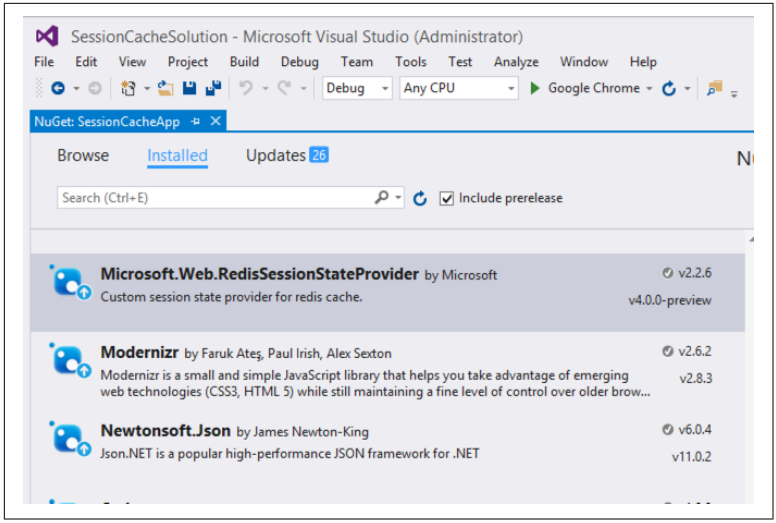

Figure 9-7. Installing a NuGet package

After the package is installed, a new section is automatically added to our *web.config* file. Comment out the previous InProc and session state provider and then configure the Redis one. Then, provide the host port, and any other values specific to your Redis installation:

```
<sessionState mode="Custom"
    customProvider="MySessionStateStore">
  <providers>
    <add name="MySessionStateStore"
            type="Microsoft.Web.Redis.RedisSessionStateProvider"
            host="127.0.0.1" port="6379" accessKey="" ssl="true"
            />
  </providers>
</sessionState>
```

Start the ASP.NET application; you'll see that the timestamp values are stored and retrieved from our Redis-based session state. Stopping and starting your application doesn't "purge" the session state, because it's stored in Redis instead of in memory. Querying the "KEYS" of the Redis instance shows the session information for the connected clients (see Figure 9-8).

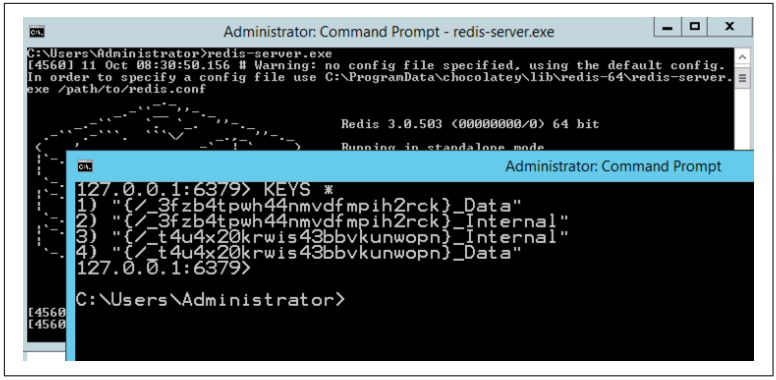

Figure 9-8. Querying session information stored in Redis

Move to Token-Based Security Schemes

One of the stickier issues when you're considering a replatforming and migration of ASP.NET applications is security—specifically, untangling Integrated Windows Authentication from your code. As we saw earlier, using this scheme is an antipattern because it's not cross-platform or particularly friendly to other programming languages. A more cloud-native option would be OpenID Connect. In this recipe, we remove the Integrated Windows Authentication from our app, and replace it with OpenID Connect.

For example purposes, I created a new ASP.NET Web Forms app (Figure 9-9) with Integrated Windows Authentication enabled. This represents our "classic" app infested with bad practices.

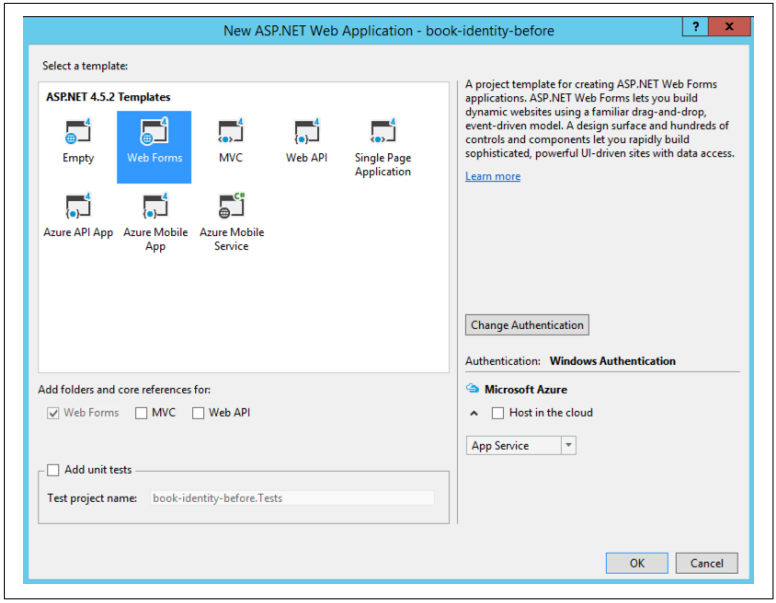

Figure 9-9. ASP.NET Web Forms app with Integrated Windows Authentication enabled

When creating or reviewing a project that uses Windows Authentication, you'll see something like this in your *web.config* file:

```
<authentication mode="Windows" />
<authorization>
  <deny users="?" />
</authorization>
```

Your Windows-infused code has operations annotated with `Authorize` or `PrincipalPermission` statements, or checks to see if the user is in a particular role. This takes advantage of the .NET identity system. With Integrated Windows Authentication and the IIS web server, a user's roles are typically populated by Active Directory. Here's my *Default.aspx.cs* file.

```
protected void Page_Load(object sender, EventArgs e)
{
  //pull user name
  string username = User.Identity.Name;

  //check roles within a c# method
  bool rolecheck = User.IsInRole("WebAdmins");

  //variable that holds value from protected call
```

```
    string result = String.Empty;

    //call "protected" operation
    try
    {
      result = GetSecureData();
    }
    catch(System.Security.SecurityException ex)
    {
      result = ex.Message;
    }

    //print results
    lblMessage.Text = String.Format(@"The username is {0},
    it is {1} that they are a WebAdmin, and result of protected
    call is {2}",
      username,
      rolecheck.ToString(),
      result);
  }

  [PrincipalPermission(SecurityAction.Demand, Role="WebAdmins")]
  private string GetSecureData()
  {
    return "super secret info!";
  }
```

In this recipe, we use Azure Active Directory (Azure AD) to expose an OpenID Connect (OIDC) endpoint for authentication. Then, we refactor our code to use OIDC and still retain the functionality we had previously. Note that you can perform variations of this pattern using Active Directory Federation Services, or identity services like Okta or Auth0. You can also swap in Pivotal's Single Sign-On (SSO) tile and Steeltoe's identity libraries for the custom code I use next. But here, I wanted to demonstrate a vanilla recipe that you could easily try yourself.

Provision an Azure AD instance in the public cloud, as demonstrated in Figure 9-10. This is a fairly straightforward process and can fit within its free usage tier.

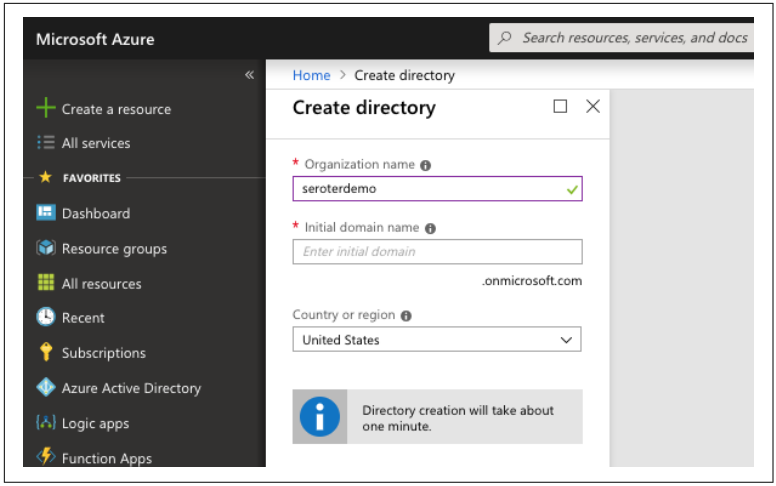

Figure 9-10. Provisioning an Azure AD instance

Using the "app registrations" section of Azure AD, register a new application with a callback URL to your application (Figure 9-11). Note that my URL maps to the localhost instance I'm using in this recipe.

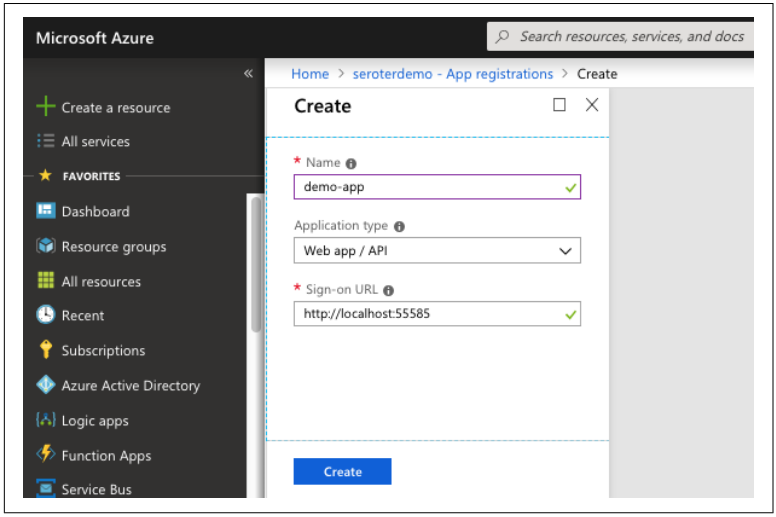

Figure 9-11. Registering a new application with a callback URL

After creating the registered app, you can see values like the application ID and object ID. The application ID maps to the client ID in the OAuth world, so save that value for future use. We have a couple

key configurations to set in Azure AD before switching to code. First, under Settings, we set the Reply URLs. This value instructs Azure AD where to redirect the client and also helps verify that the source and destination domain are the same. Figure 9-12 shows that the value I set equaled my primary site URL.

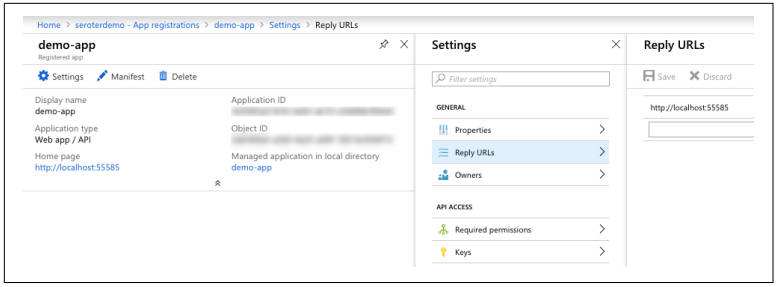

Figure 9-12. Setting key configurations in Azure AD

Back on the starting page of the registered app, you'll find a button labeled "manifest." Here, we can view and edit the identity configuration, including the setting to return the "groups" associated with the user. I changed the groupMembershipClaims to "SecurityGroup" so that I had access to these claims after signing in from my ASP.NET application.

Finally, to demonstrate that group/role functionality, I returned to the Azure AD instance and added multiple groups, and added my user to each of them (Figure 9-13). Be aware that each group has an object ID (represented as a GUID), and that's the value returned to the client as a claim ID.

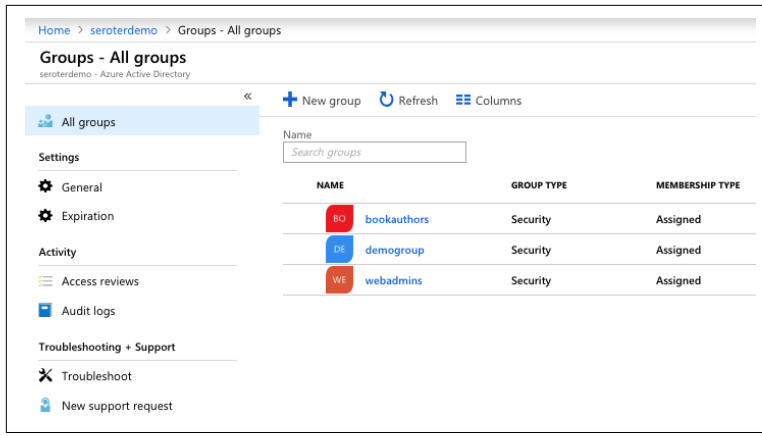

Figure 9-13. Adding groups to an Azure ID instance

Next up in the recipe is refactoring our code. First, I added three NuGet packages to my project:

- `Microsoft.Owin.Security.OpenIdConnect`
- `Microsoft.Owin.Security.Cookies`
- `Microsoft.Owin.Host.SystemWeb`

These packages bring in the necessary Open Web Interface for .NET (OWIN) middleware components for doing cookie-based SSO.

Before adding code, we update our configuration with some application settings. Specifically, we're adding keys used by our code to redirect to Azure AD. In the *web.config*, add the following:

```
<appSettings>
  <add key="ClientId"
          value="[app id from registered app]" />
  <add key="Tenant"
          value="[name of tenant].onmicrosoft.com" />
  <add key="AzureADInstance"
          value="https://login.microsoftonline.com/{0}" />
  <add key="PostLogoutRedirectUri"
          value="http://localhost:55585" />
</appSettings>
```

That uses the application ID from our Azure AD–registered application, our tenant name, and the redirection URL, which should map to the reply URL we set up in Azure AD. Given our previous pat-

tern, you could also use a Config Store to stash and retrieve these values.

Next, we need to add a `Startup.cs` class that configures all the OpenID Connect authentication. Right-click the project folder, choose "Add" and select "OWIN Startup class." Name the class Startup.cs. There are four "using" statements automatically added at the top, and introduce these eight additional ones.

```
using System;
using System.Threading.Tasks;
using Microsoft.Owin;
using Owin;
//added
using System.Configuration;
using System.Globalization;
using Microsoft.Owin.Security;
using Microsoft.Owin.Security.Cookies;
using Microsoft.Owin.Security.OpenIdConnect;
using Microsoft.Owin.Security.Notifications;
using Microsoft.IdentityModel.Protocols.OpenIdConnect;
using Microsoft.IdentityModel.Tokens;
```

At the top of the `Startup.cs` class, we define variables that pull configuration values from the *web.config* file:

```
private static string clientId =
    ConfigurationManager.AppSettings["ClientId"];
private static string aadInstance =
    ConfigurationManager.AppSettings["AzureADInstance"];
private static string tenant =
    ConfigurationManager.AppSettings["Tenant"];
private static string postLogoutRedirectUri =
    ConfigurationManager.AppSettings["PostLogoutRedirectUri"];

string authority = String.Format(CultureInfo.InvariantCulture,
                                 aadInstance,
                                 tenant);
```

The heart of this class is the `Configuration` operation, which triggers a cookie-based authentication:

```
public void Configuration(IAppBuilder app)
{
  app.SetDefaultSignInAsAuthenticationType(
       CookieAuthenticationDefaults.AuthenticationType);
  app.UseCookieAuthentication(new CookieAuthenticationOptions());

  app.UseOpenIdConnectAuthentication(
    new OpenIdConnectAuthenticationOptions{
      ClientId = clientId,
```

```
        Authority = authority,
        PostLogoutRedirectUri = postLogoutRedirectUri,
        RedirectUri = postLogoutRedirectUri,
        ResponseType = OpenIdConnectResponseType.IdToken,
        Notifications =
            new OpenIdConnectAuthenticationNotifications
        {
          AuthenticationFailed = context =>
          {
            context.HandleResponse();
            context.Response.Redirect(
                 "/Error?message=" + context.Exception.Message);
            return Task.FromResult(0);
          }
        }
      }
    );
  }
```

How do we initiate the login event? Let's add an explicit login command to our application. In the `Site.Master` definition, add a `LoginView` to the top menu. I placed it after the tag that defined the navigation menu:

```
<asp:LoginView runat="server" ViewStateMode="Disabled">
  <AnonymousTemplate>
    <ul class="nav navbar-nav navbar-right">
      <li>
        <a
          href="Site.Master"
          runat="server"
          onserverclick="btnLogin_Click">Login</a>
      </li>
    </ul>
  </AnonymousTemplate>
  <LoggedInTemplate>
    <ul class="nav navbar-nav navbar-right">
      <li>
        <asp:LoginStatus runat="server"
          LogoutAction="Redirect"
          LogoutText="Logout"
          LogoutPageUrl="~/"
          OnLoggingOut="Unnamed_LoggingOut" />
      </li
    </ul>
  </LoggedInTemplate>
</asp:LoginView>
```

In the codebehind of the `Site.Master`, I added four `using` declarations:

```
//added
using Microsoft.Owin.Security;
using Microsoft.Owin.Security.OpenIdConnect;
using System.Web.Security;
using Microsoft.Owin.Security.Cookies;
```

Then, I added functions for the login and logout command. Notice that the login command triggers an authentication request:

```
protected void btnLogin_Click(object sender, EventArgs e)
{
  if (!Request.IsAuthenticated)
  {
      HttpContext.Current.GetOwinContext().Authentication
         .Challenge(
         new AuthenticationProperties { RedirectUri = "/" },
         OpenIdConnectAuthenticationDefaults.AuthenticationType);
  }
}

protected void Unnamed_LoggingOut(
    object sender, LoginCancelEventArgs e)
{
  Context.GetOwinContext().Authentication.SignOut(
      CookieAuthenticationDefaults.AuthenticationType);
}
```

The final step? Refactoring that *Default.aspx.cs* code that checks user roles. First I updated the "secured" operation to no longer demand a role, but still demand an authenticated user:

```
[PrincipalPermission(SecurityAction.Demand)]
private string GetSecureData()
{
  return "super secret info!";
}
```

Next, to do security checks, I added three `using` statements at the top of the class:

```
//added for OIDC
using Microsoft.Owin.Security;
using Microsoft.Owin.Security.OpenIdConnect;
using System.Security.Claims;
```

Then, I changed the code that looked up the username, looked up user roles, and called the `secure` function. Notice that while we get the user's group assignments as claims, we don't automatically get the friendly name. You could use another Azure AD API lookup to translate it, if needed:

```
string username = User.Identity.Name;
//using object ID, as we don't get the friendly name in the claim
const string bookGroup = "2c4e035e-9aba-4ea9-be4b-fc67bd762242";
string rolestatus = "no";
string result = String.Empty;
var userClaims =
    User.Identity as System.Security.Claims.ClaimsIdentity;

//look for claim associated with the desired user role
Claim groupDevTestClaim = userClaims.Claims.FirstOrDefault(
          c => c.Type == "groups" &&
          c.Value.Equals(
             bookGroup,
             StringComparison.CurrentCultureIgnoreCase));

if (null != groupDevTestClaim)
{
  rolestatus = "yes";
}

try
{
  result = GetSecureData();
}
catch (System.Security.SecurityException ex)
{
  result = ex.Message;
}

//print results
lblMessage.Text = String.Format@(
  "The username is {0}, {1} they are a book author,
  and result of protected call is {2}",
  username,
  rolestatus,
  result);
```

Don't forget to "turn off" Windows Authentication in your *web.config* file. Simply switch `authenticationMode` to `None` and delete the authorization block.

With that, the ASP.NET application is refactored to use a cross-platform, non-Windows-specific way to authenticate and authorize users. When starting up the application and clicking login, I'm immediately redirected to log in, as shown in Figure 9-14.

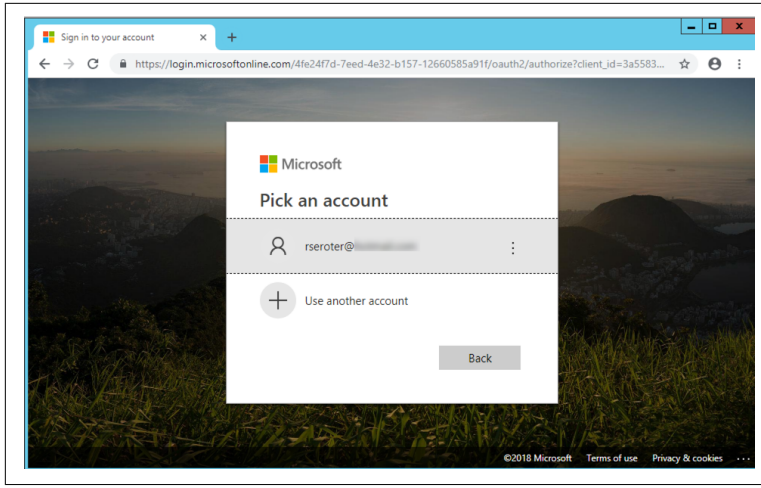

Figure 9-14. Being redirected to log in

After logging in, I'm redirected to the application, and my code executes to confirm that I'm logged in and authorized, as depicted in Figure 9-15.

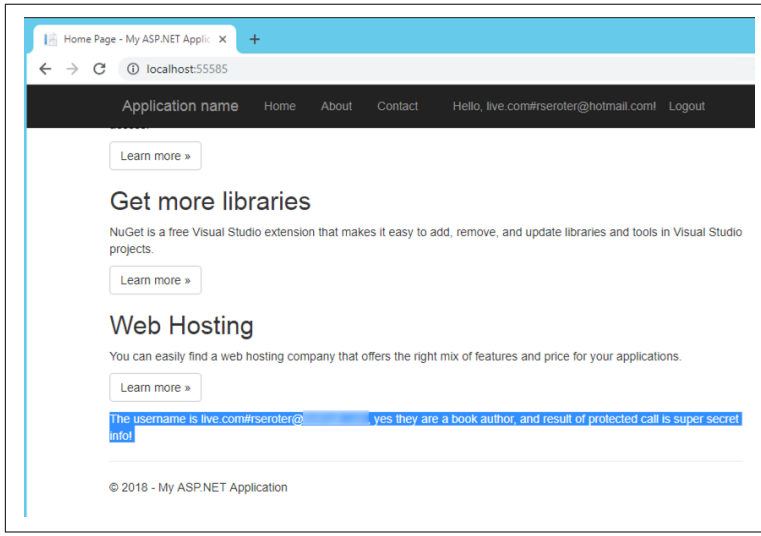

Figure 9-15. Confirming login and authorization

Put .NET Core Apps on Pipelines

If you do *nothing else* that I recommended in this book, do this. By putting your apps on pipelines, you set yourself up for a repeatable path to production. This recipe walks you through the necessary steps to put an ASP.NET Core app on a Concourse pipeline that deploys the app to Pivotal Cloud Foundry. Even if your destination is somewhere else or you're using the .NET Framework, this pattern should be useful.

Concourse is a declarative, pipeline-oriented CI/CD system. It uses the concepts of resources, jobs, and steps. *Resources* are the things you'll use in your pipeline. A resource might be a Git repo or an Azure Blob Storage account. *Jobs* determine what your pipeline actually does. They explain how the resources pass through the pipeline and how you visualize the flow. A job is made up of *steps*. Each step may grab a resource or execute a low-level task. Tasks execute in ephemeral containers, so when a pipeline is finished, there's no mess to clean up.

A complete .NET Core pipeline should: fetch source code; run tests, including unit, integration, smoke, and performance tests; and then deploy the software. That deployment phase consists of generating a build artifact and pushing that artifact to the target environment.

In this recipe, we build a basic pipeline that runs unit tests and publishes the result to Cloud Foundry. We begin with an ASP.NET Core application that has some xUnit test baked in. My *TestClass.cs* file defines a couple of basic unit tests against a simple web controller (`ValuesController`), as shown in Figure 9-16:

```
using Xunit;
using tested_core_app.Controllers;

namespace unittests {
  public class TestClass {
    private ValuesController _vc;

    public TestClass() {
      _vc = new ValuesController();
    }

    [Fact]
    public void Test1(){
      Assert.Equal("pivotal", _vc.Get(1));
    }
```

```
        [Theory]
        [InlineData(1)]
        [InlineData(3)]
        [InlineData(9)]
        public void Test2(int value) {
          Assert.Equal("public", _vc.GetPublicStatus(value));
        }
      }
    }
```

Local unit tests confirm that our code passes. I then checked this code into a GitHub repository so that my deployment pipeline could "see" it.

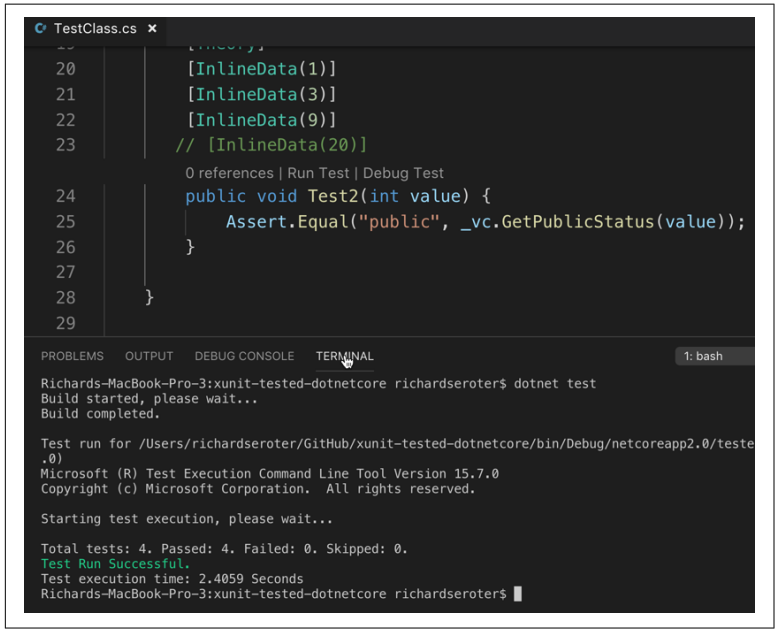

Figure 9-16. Running local unit tests

Note that although Concourse can use Windows-based worker nodes to execute tasks, this recipe uses a Linux-based runtime. The simplest way to get Concourse up and running on any OS is via Docker Compose. After Concourse is up and running, you can log in to the *fly* CLI to create and manage pipelines; see Figure 9-17.

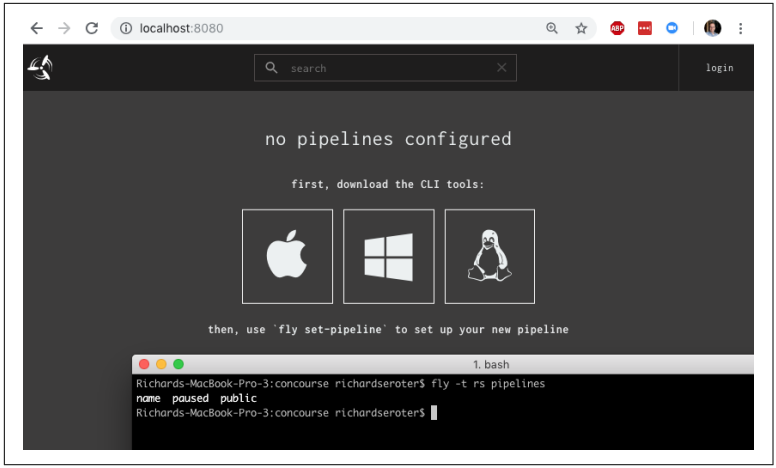

Figure 9-17. Concourse, up and running

Concourse pipelines are defined in a YML format. We define a GitHub repo and Pivotal Cloud Foundry endpoint as the two resources:

```
---
resources:
- name: seroter-source
  type: git
  source:
    uri: https://github.com/rseroter/xunit-tested-dotnetcore
    branch: master
- name: pcf-on-aws
  type: cf
  source:
    api: https://api.run.pivotal.io
    skip_cert_check: false
    username: [username]
    password: [password]
    organization: [org]
    space: development
```

The pipeline then has two jobs. The first executes unit tests. Notice that it uses a Microsoft-provided Docker image to host the tests. Next, it runs a `dotnet` test command to execute the xUnit tests:

```
jobs:
- name: aspnetcore-unit-tests
  plan:
    - get: seroter-source
      trigger: true
    - task: run-tests
      privileged: true
```

```
    config:
      platform: linux
      inputs:
      - name: seroter-source
      image_resource:
        type: docker-image
        source:
          repository: microsoft/aspnetcore-build
      run:
        path: sh
        args:
        - -exc
        - |
            cd ./seroter-source
            dotnet restore
            dotnet test
```

If that job passes, the second job kicks off. Note that there's no data passing directly between jobs. You can share files between tasks in a job, but not between jobs. In a real-world scenario, you'd likely drop the results of the first job into an artifact repository and then pull from that repository in the next job. In this recipe, we simply grab the source code again from the GitHub repo (if the tests pass) and push it to PCF:

```
jobs:
- name: aspnetcore-unit-tests
  [..]
- name: deploy-to-prod
  plan:
    - get: seroter-source
      trigger: true
      passed: [aspnetcore-unit-tests]
    - put: pcf-on-aws
      params:
        manifest: seroter-source/manifest.yml
```

Deploying a pipeline is easy. From the fly CLI, you provide the name of your pipeline and point to the *pipeline.yml* file.

```
fly -t rs set-pipeline -p book-pipeline -c pipeline.yml
```

The result? A green pipeline, as shown in Figure 9-18, if all the unit tests pass.

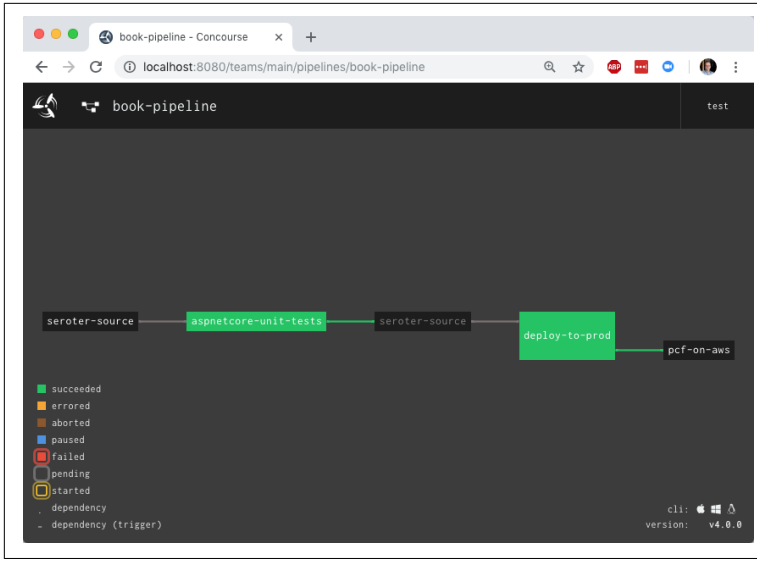

Figure 9-18. Success!

And we get a running app on PCF (see Figure 9-19). From this point on, any check-in to my code on GitHub triggers the pipeline and pushes my code to Cloud Foundry. What a straightforward way to automate the path to production!

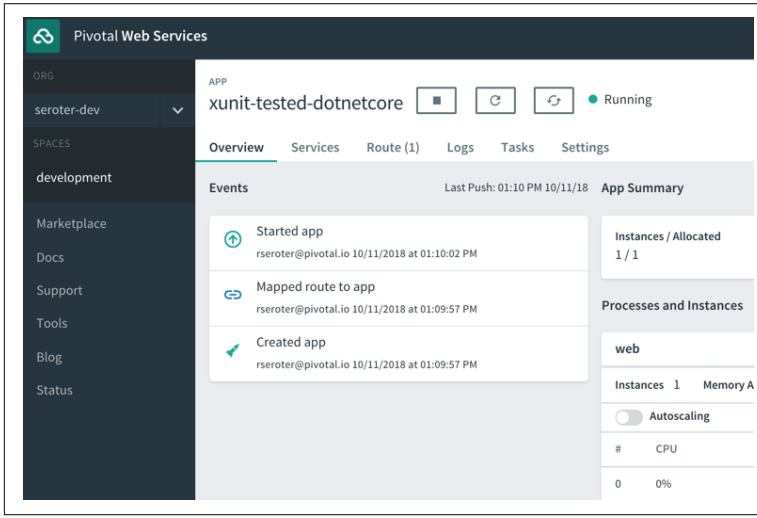

Figure 9-19. The app up and running on PCF

Summary

All of these recipes can help you quickly modernize your .NET apps and remove constraints that prevent those apps from being cloud-native. Instead of simply moving apps unchanged from one host to the next, consider investing in modernization.

CHAPTER 10

Your Call to Action

When I finish watching a fixer-upper show on HGTV, I go on with my day and pay no heed to what I observed. I don't want you to do that after you read this book! No, you have some clear next steps to ensure that you regain control of your .NET portfolio and start realizing new value from your existing assets. I'd like you to do four things.

Step 1: Assess Your Portfolio

First, you need to get a handle on what's in front of you. Scour your landscape to get a sense of which .NET project types you have. Go back to Chapter 2 and list your software for each category.

Consider a few vectors when plotting your portfolio. For example, you might want to create a chart that grades technical debt on the y-axis, and business value on the x-axis (see Figure 10-1). The circles that you plot represent the .NET apps, and the circle size corresponds to how well that app represents what's in your overall portfolio. That last factor matters because you want to tackle modernization of apps that generate patterns you can use elsewhere.

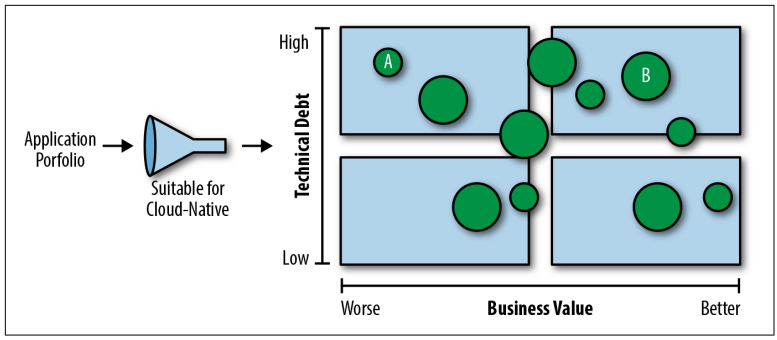

Figure 10-1. Plotting your application portfolio

In Figure 10-1, circle A is a poor candidate for modernization. It has low business value and a high degree of technical debt, and upgrading it doesn't help us elsewhere in the portfolio. Contrast that with circle B, which still has some serious technical debt to overcome, but it has strong business value and would have modernization patterns you can apply elsewhere.

After creating your inventory of .NET apps and doing some light scoring, it's time to choose a handful for initial modernization. You want to choose among the options that are useful to your business, and can generate repeatable modernization patterns.

Step 2: Decide on a Modernization Approach

There's no one-size-fits-all approach to modernization. Some .NET software doesn't need to be fully cloud native. Some light refactoring might be all you need. Other applications require a more comprehensive rewrite to introduce the necessary cloud-native attributes you crave. And some software will simply be retired because it's no longer adding any value.

Figures 10-2 and 10-3 show some useful visual models for thinking about what to do with your existing .NET software. Figure 10-2 considers what to do based on application priority—that is, whether you plan to invest, maintain, or divest the .NET app.

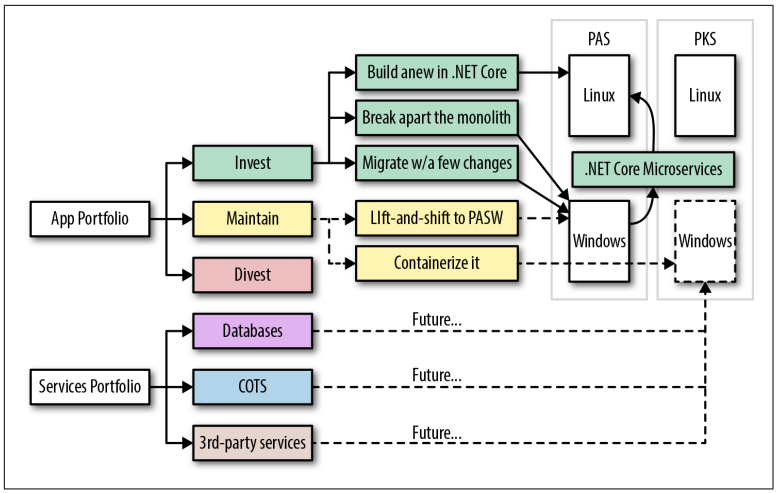

Figure 10-2. A modernization model based on app priority (created by William Martin at Pivotal)

You can see there that for apps you're investing in or maintaining, there are a handful of strategies and target runtimes. Figure 10-3 puts the focus on the type of .NET software and how that influences what you do with it. You can see that for web apps and .NET console apps, there's a clear path. The only ones you're currently "stuck" with are desktop apps.

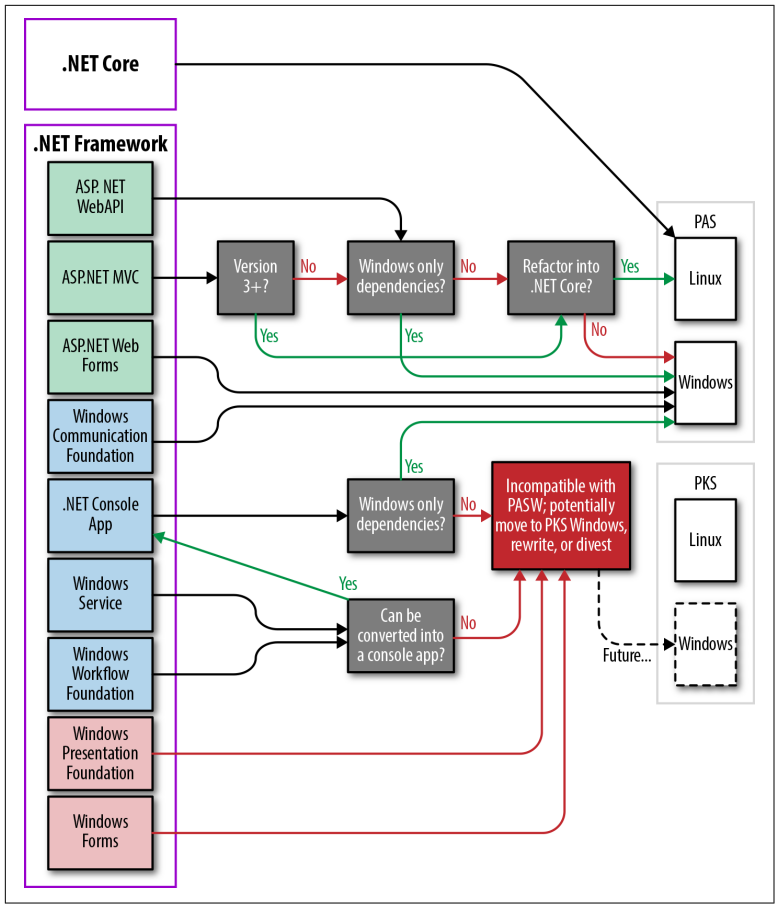

Figure 10-3. A modernization model based on software type (created by Shawn Neal and Shanfan Huang at Pivotal)

As you plan your modernization, you have questions to ask yourself. Are you creating a microservices architecture? How exactly are you going to decompose existing applications? What new components are you going to introduce? What's the ideal place for this modernized .NET software to run? Hopefully this book has helped you think through those answers!

Step 3: Modernize Your Initial Set of Apps

It's critical that you start this process small. Don't set up a plan to modernize 300 apps this year. There's going to be a learning process.

Based on a value-stream assessment or some other technique, choose a mix of apps (see step 1) that add business value and offer the most learning opportunities. Don't forget to mix in one or two apps that offer some "quick win" success to motivate your team!

Once you choose your initial set of target apps, assemble a team that will stick together to modernize them. First, try to collect some upfront metrics that measure your current state. Think of things like how long it takes to update the app today, its uptime numbers, and load limits. This will come in handy later. Next, focus on defining useful standards for developer tooling and project setup, and immediately get those applications into CI/CD pipelines. As each week goes by, do a retrospective on what you've learned and any surprises you've encountered.

Step 4: Record Your Patterns and Spread the News

If you're part of this modernization effort, it's critical to document your journey. Your goal is to make this a repeatable process, and one that many can perform. Write down the core patterns that you use, and stash them on a wiki or some other sharable place. At Pivotal, when we do application transformation projects with customers, we create and leave behind a "cookbook" of all the recipes we created together. This means that future teams can follow those recipes, and easily add new ones.

A Final Note

I hope you found this book helpful. Like a home improvement project, your .NET modernization is a journey. It requires upfront assessment of the task at hand and careful consideration for what work you want to take on. At the same time, both home improvement and .NET modernization projects offer unique opportunities to make meaningful changes. Don't miss the chance to add new capabilities while making your software more sustainable and performant.

You have an important job to do, and I know that you'll tackle it with gusto. Best of luck!

About the Author

Richard Seroter is a senior director of product for Pivotal. He's also an 11-time Microsoft MVP for cloud/integration, an instructor for Pluralsight, the lead InfoQ.com editor for cloud computing, and author of multiple books on application integration strategies. As a product director, he gets to lead product, partner, and customer marketing and help shape Pivotal's position and messaging. He maintains a regularly updated blog at *seroter.wordpress.com* on topics of architecture and solution design, and you can also find him on Twitter as @rseroter.