Pro .NET on Amazon Web Services

Guidance and Best Practices for Building and Deployment

William Penberthy
Steve Roberts

Apress®

Pro .NET on Amazon Web Services

Guidance and Best Practices for Building and Deployment

William Penberthy
Steve Roberts

Apress®

Pro .NET on Amazon Web Services: Guidance and Best Practices for Building and Deployment

William Penberthy
Seattle, WA, USA

Steve Roberts
Seattle, WA, USA

ISBN-13 (pbk): 978-1-4842-8906-8
ISBN-13 (electronic): 978-1-4842-8907-5
https://doi.org/10.1007/978-1-4842-8907-5

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Jonathan Gennick
Development Editor: Laura Berendson
Coordinating Editor: Jill Balzano

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media LLC, 1 New York Plaza, Suite 4600, New York, NY 10004. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

This book is dedicated to you, our readers.

Hopefully, we have saved you at least one bout of
“Why isn’t this working?”

Table of Contents

About the Authors

William Penberthy has over 25 years' experience in software development (almost 17 of which is .NET) and brings a pragmatic approach to software development. With much of that time spent in consulting, he has worked on many different projects and used many different designs and approaches. In 2021, he joined Unify Consulting where he works with clients in optimizing their software development process and building distributed systems for the cloud.

Steve Roberts is a Senior Developer Advocate for .NET and PowerShell development at AWS. Based in Seattle, Washington, Steve worked for eight years as a Senior Development Engineer on the AWS tools for .NET before switching focus to a developer advocacy role. He was the development lead for the AWS Tools for PowerShell and the AWS Tools for Azure DevOps and contributed to the AWS Toolkits for Visual Studio and Visual Studio Code, and the AWS SDK for .NET. Prior to joining AWS, Steve had over 20 years' experience as a developer focused on IDE tools and integrations.

About the Technical Reviewer

Peter Himschoot works as a lead trainer, architect, and strategist at U2U. He has a wide interest in software development that includes applications for the Web, Windows, Clean Architecture, Domain-Driven Design, and Security. He has trained thousands of developers, is a regular speaker at international conferences, and has been involved in many web and mobile development projects as a software architect. He has been a Microsoft Regional Director (from 2003 to 2019) and cofounded the Belgian Visual Studio User Group (VISUG) in 2006, which is a group of trusted advisors to developer and IT professional audiences and to Microsoft.

Acknowledgments

Kudos to you, the reader, for deciding to learn and try something new. Though, of course, for those of you experienced developers, this is not something new. It's amazing what we, the authors, learned while writing this book – and we thought we were already experts. For those of you that are just starting out your life in software development, know that this is something you will need to continue to do for your entire career as this field changes about as quickly as the water in a river.

We would also like to thank Jonathan Gennick and Jill Balzano from Apress who had the unenviable job of herding the cats. We would also like to mention our appreciation to Brian Beach and Peter Himschoot who had the onerous job of doing our technical review.

Lastly, but certainly not leastly (ha – let's see if we can get it through at least once), is the support for our wives, Jeanine and Deb, who allowed us to spend way too much of our free time working on this project.

Introduction

Cloud computing can be thought of as a way to deliver on-demand computing services. These computing services can be as high level as complete applications, such as Salesforce, or as low level as storage or application processing. Access to these services is generally through the public Internet. The concept of the cloud has taken off, and there is research that says more than one third of all IT spending worldwide is on cloud computing and that number will continue to grow. This book is about a piece of that overall cloud computing pie, the combination of one of the most popular software development frameworks, .NET, and the first and largest cloud computing provider, Amazon Web Services (AWS).

It sounds straightforward when you look at that previous sentence until you really start to think through what that means; it sets the expectation that we will talk about the union of .NET functionality and AWS functionality. That is simply not possible as that union is huge and writing about it would take several volumes and likely be out of date before we even finished the second one. Instead, in this book, we are trying to identify some of the most commonly identified development scenarios, and we will use those to guide our journey.

Before we move into the juicy details of using .NET on AWS, let us first go over some of the fundamental differences between working with a cloud service and working with "your own servers." There are multiple different ways in which companies manage their "own servers." Some lease hardware from an infrastructure or hosting provider and are responsible for everything above the iron, including OS patches, systems drivers, installed applications, and everything in between. If those companies want to run virtual machines, then they have to purchase the management software and maintain that as well. Other companies purchase the server hardware outright and thus add details like network cards or other "pieces of metal" into the areas for which they are responsible. Regardless of the approach that you take, your company will have some system management responsibilities.

That does not necessarily change when you move to the cloud. However, it becomes easier to define the responsibilities that each party has. AWS calls this definition of responsibilities the *Shared Responsibility Model*. This model defines what parts of the overall operational burden will be owned by AWS and what parts of the burden will be owned by the customer. This is important to consider when building your applications as you should be building them in such a way as to help minimize the operational burden that you, as a developer/customer, have when working with AWS services. Not only will this minimize your operational burden, it also, oddly enough, tends to decrease your actual cloud service cost because it allows for a much finer control over the various areas used by your application, such as processing, memory usage, and storage. We promise there will be more on this later when we go into the details of running your .NET application on AWS!

We will be using a sample application through our journey. We called it *TradeYourTools*, and it is a simple tool-sharing application that is designed to help members of a neighborhood group document their available hand and electrical tools and to provide a simple process to reserve a tool and arrange their pickup and return. The application is starting as an ASP.NET MVC v4.6 application that uses ASP.NET Identity for registration. It has a Microsoft SQL Server back end that stores tool data, including pictures, user information, and any uploaded pictures. It works but be warned; it is not pretty – neither one of us claims to be a UX expert!

We will be altering this application through our journey. We are assuming that you are an experienced .NET developer and thus will not be spending any effort talking about the fundamentals of what we are doing. To take full advantage of the sample application, however, requires that you have access to a .NET integrated development environment (IDE). The most used IDE is Microsoft Visual Studio. There are several versions of Visual Studio, each with differing sets of functionality and price points. JetBrains offers a competing IDE, Rider, and there is always the ubiquitous Visual Studio Code, a cross-platform IDE that also supports .NET development. Most of the screenshots will be using Visual Studio unless we are specifically talking about a different IDE.

You will also enhance your experience if you have access to a running SQL Server instance so that you can work with a local, fully running application. The sample application will come with a database creation script that will create all of the necessary tables and relationships as well as load some sample data that will help you understand the application and how all of the pieces go together. While this will be useful, it is by no means required. If you must choose between an IDE and a database, choose the IDE – you will spend more time there!

You can download a trial version of JetBrains Rider at `www.jetbrains.com/rider/`. You can download both Visual Studio and Visual Studio Code at `https://visualstudio.microsoft.com/`. The Community Edition of Visual Studio is free and is our recommended choice if you do not currently have access to a .NET IDE. You can download Microsoft SQL Server Developer 2019, a full-featured free edition of SQL Server that is licensed for use as a development and test database.

PART I

Getting Started

CHAPTER 1

The Core Essentials

Getting started is always the hardest part of new systems, and AWS is no different. Approaching the breadth of services and the functionality available in the AWS cloud can be a daunting prospect! What services should you consider first for hosting your application? How do terms you may be familiar with, such as virtual machines, map to services? How do you, as a developer, authenticate to provision and work with resources? How does your application code authenticate to be able to call AWS services?

In this chapter, we'll look at the essentials – what you really need to know about AWS to get started. We are not going to go particularly deep – that's for later sections of the book – but this chapter will give you an overview of essential services and functionality that will be useful if you've never worked with AWS before. In addition to core services, we will take our first look at the AWS Management Console and use it to set up a "development user" identity as a recommended best practice. You'll use this user identity to work with AWS services, resources, and the tools (outlined in Chapters 2 and 3) during the remainder of the book.

The Essentials of AWS

AWS makes available a portfolio of over 175 services – and the portfolio continues to grow. Let's be honest – you are highly unlikely to make use of all the services in a single application! There's an old maxim – how do you eat an elephant (not that you should of course, elephants are awesome)? The answer is – one piece at a time, and the same applies to getting started on AWS. Once you have an initial understanding of a core set of services and the terms you will routinely encounter working with AWS, you can then pivot to determining how best to map them to the needs of your application.

W. Penberthy and S. Roberts, *Pro .NET on Amazon Web Services*,
https://doi.org/10.1007/978-1-4842-8907-5_1

Regions and Availability Zones

Let's begin with the layout of AWS' global infrastructure. AWS operates 24 geographic *Regions* around the world (at the time of writing; more are planned). Regions are fully isolated, and each is composed of at least two, usually three, sometimes more, isolated and physically separate *Availability Zones*. Each Availability Zone (or AZ as you'll more commonly see them mentioned) is composed of one or more physically separated data centers. Each data center has redundant power, networking, and high-bandwidth low-latency connectivity. This is different from other cloud providers who may treat a single data center in a geographic area as a region.

Regions are identified both by name, for example, *US East (N. Virginia)*, or *US West (Oregon)*, etc., and by a corresponding code. You'll often make use of this code in your applications. For the two regions just mentioned, US East (N. Virginia) is referred to as *us-east-1*. US West (Oregon) is identified as *us-west-2*.

Availability Zones are identified with a combination of the parent region code and a letter suffix – a, b, c, etc. For example, *us-west-2b* refers to an AZ in the US West (Oregon) region. Your applications and the resources they consume can exist in a single AZ, or be deployed to multiple AZs in a region for high availability. Figure 1-1 shows the logical layout of the US West (Oregon), or us-west-2, region. Also in the figure are illustrations of how one might choose to distribute applications (or not) across Availability Zones.

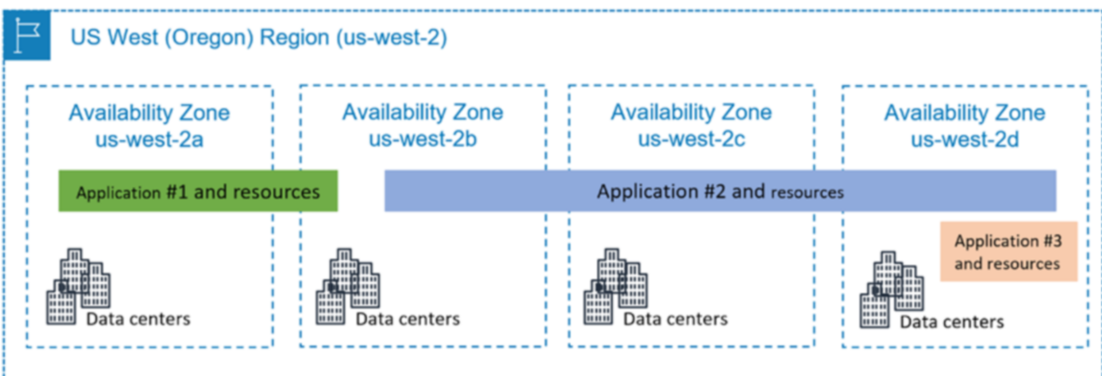

Figure 1-1. *The US West (Oregon) region and zones*

How does this global infrastructure benefit you? Regions and their Availability Zones are designed to provide you the ability to run highly available, fault-tolerant applications nearer to your customer base and to also satisfy data sovereignty laws that you may be subject to. Running applications in a region geographically closer to users helps improve latency, but you can also failover to a different region if needed. Availability Zones provide the basis of fault tolerance and scale for your applications. Assuming you've made use of Availability Zones, by deploying your application to at least two AZs in a region, a failure in a single data center – or even an entire AZ – will be transparent to your application.

This does mean, however, that you usually need to keep this infrastructure layout in mind when working with AWS. With very few exceptions, AWS services are regional, meaning that resources you create in one region are not visible in the AWS Management Console (or the various AWS tools that we'll examine in Chapters 2 and 3) when you are working in a different region. For now, just be aware that when you deploy your application resources to AWS, you'll do so in a region, and optionally across two or more AZs if you want fault tolerance. If not, and a service does not mandate a multi-AZ setup, then go ahead and use a single AZ to keep things simple.

The AWS Free Tier

The AWS Free Tier is a collection of offers spanning 85 products (at the time of writing) to help when gaining experience with AWS services. Some offers in the Free Tier are available for the first year after you create an account (and expire thereafter – so use before you lose!); others are permanently free up to a certain amount of resource usage. Periodically, ad hoc offers may become available during a promotional period, or after a new service is launched. Some examples:

- AWS Lambda serverless compute: Up to 1 million requests, and 400,000 GB-seconds of compute time, per month – always free
- Amazon DynamoDB database: Up to 25GB of storage and 25 provisioned read and 25 provisioned write capacity units per month (this is enough to handle up to 200 million requests per month) – always free
- Amazon EC2 virtual machines: Up to 750 hours of compute on certain instance sizes, for Windows and Linux – free for the first 12 months
- Amazon S3 storage: Up to 5GB of standard storage – free for the first 12 months

Many other examples can be found at the Free Tier home page: `https://aws.amazon.com/free`. Most of the examples in this book can be performed using offers in the Free Tier to reduce or eliminate cost, and we will try to call out those services that may have a charge.

Hosting Your Code – Compute Services

So you have application code to host in the cloud. But where? And using what technology? Virtual machines (VMs)? VMs in Managed Services? Containers? Or maybe you want to adopt a serverless approach and let AWS manage the compute infrastructure in its entirety for you? There's a lot of choice, so let's quickly unpack it. In later chapters, we'll examine each of these areas in more detail.

First, virtual machines. *Amazon Elastic Compute Cloud*, or *EC2* as it's more commonly known, is the virtual machine service on AWS. A virtual machine in EC2 is an *instance*. Each instance is started from an *image*, known as an *Amazon Machine Image*, or *AMI*. EC2 provides multiple "stock" images, for both Windows and multiple Linux distributions, that you can use. These images are generally, but not always, updated monthly to ensure they include the most recent patches and other software updates. There are also 7,000+ images available through the AWS marketplace with each having various software packages pre-installed. Alternatively, you can use your own images – which can be built and snapshotted from an AWS-provided image if you so choose. You might take this route if you have specific installation or configuration requirements and can't perform these actions when launching the "stock" images provided by AWS.

Once an EC2 instance is running, you are in complete control – just as with virtual machines you might run on your own machine. You can pause (suspend), stop, or terminate them. You can also remote into them using SSH or Remote Desktop. Figure 1-2 shows an example of some instances in various states in the AWS Management Console.

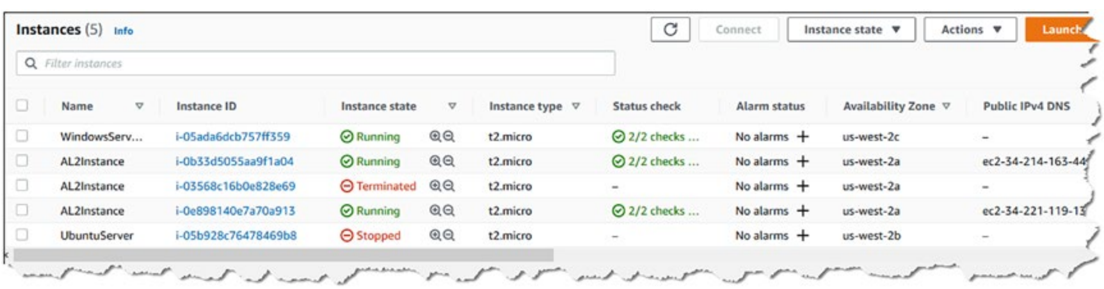

Figure 1-2. *EC2 instances in the AWS Management Console*

You are responsible for keeping instances (and custom images) up to date with patches, etc. (AWS also provides additional services that can help you do this). AWS operates a shared responsibility model, which means that AWS protects the physical infrastructure and network but you are responsible for the security on your resources.

How does your application get deployed onto an EC2 instance? As with the virtual machine instance, you have complete control here too. You can write scripts to download the application binaries and other resources and place them onto the instance as it starts, or you can use services such as *AWS CodeDeploy* (among others) to perform the deployment. In the case of CodeDeploy, you simply upload the built application bundle and choose the instance(s) to be involved (by selecting the instance IDs, or some other criteria such as tags or membership in a group), and CodeDeploy does the rest.

If managing running instances sounds like more work than you would like, but you still want to retain some control, then consider services such as *AWS Elastic Beanstalk* or *Amazon Lightsail*. Both build on top of EC2 to offer virtual machines under the hood but abstract away some or most of the management and deployment aspects.

AWS Elastic Beanstalk is a service for deploying and scaling web applications and services and is the fastest and simplest way to get your application up and running on EC2 virtual machines. As a developer, your focus is on building and packaging your application. For .NET Framework applications, you use a Web Deploy package. For .NET Core and .NET 5+, you use `dotnet publish` to build the deployment package. In both cases, you upload the bundle to AWS and instruct Elastic Beanstalk to deploy it to the instance(s) in your application's *environment*. The environment contains the EC2 instance(s) and other resources employed in hosting your application code. Elastic Beanstalk takes care of provisioning and managing your infrastructure resources, and handling deployments, but you can still, if you wish, work with the underlying resources. Should you need to, and network security settings permit, it's possible to remote into the EC2 instances in the environment. Figure 1-3 shows an example of the resources and their chosen configuration in a load-balanced auto-scaled Elastic Beanstalk environment.

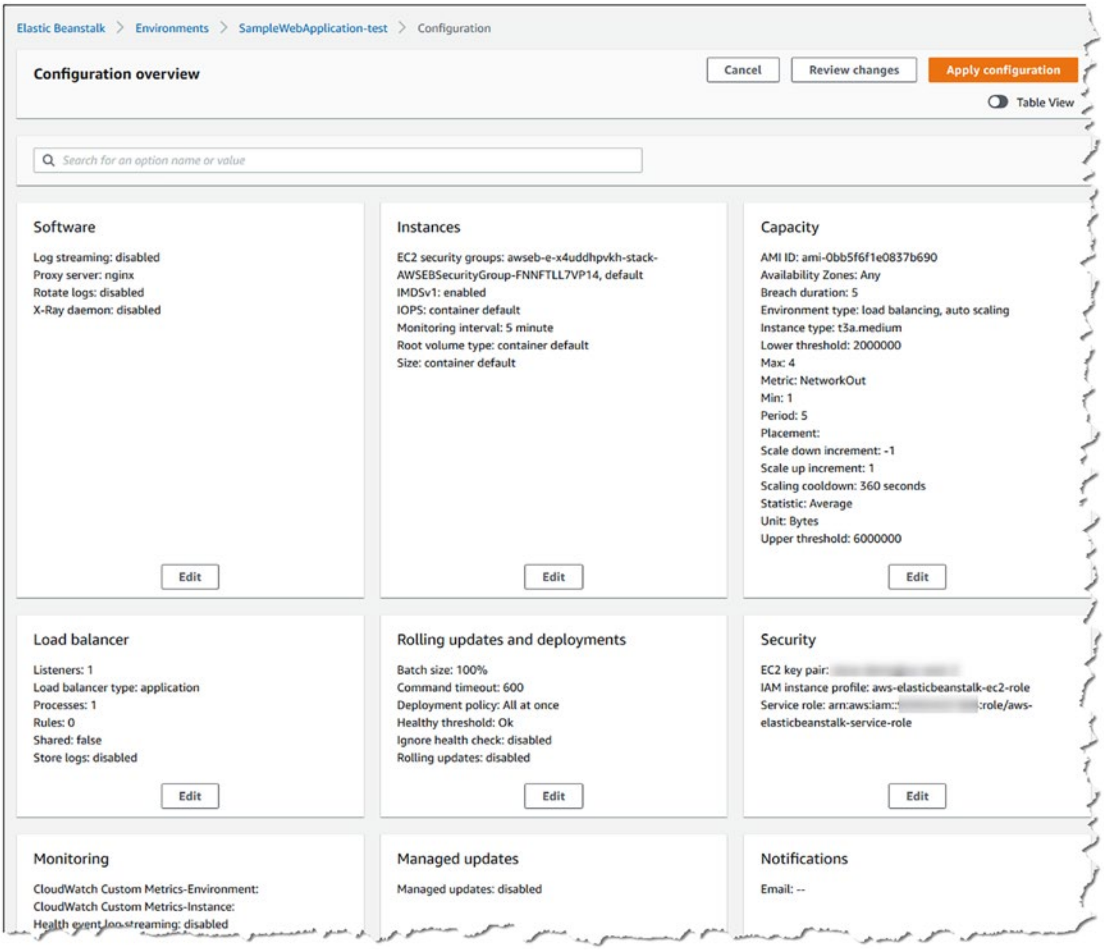

Figure 1-3. *Example Elastic Beanstalk environment and resources*

Amazon Lightsail is another abstraction on top of EC2. Like Elastic Beanstalk, Lightsail takes care of the provisioning of the underlying EC2 resources but takes a more simplified view of managing them. With Elastic Beanstalk, you can tweak and adjust almost all the settings of the underlying resources (even though the service created them for you). With Lightsail, fewer settings are exposed, but you can still remote into the underlying instances if needed. Offering an even more managed experience than Elastic Beanstalk, Lightsail is ideal for simple workloads, or just getting started on AWS, but provides the ability to “grow into” the full EC2 experience later if you need to.

Next, we have the choice of using containers. AWS provides two container services: *Amazon Elastic Container Service* (*ECS*) and *Amazon Elastic Kubernetes Service* (*EKS*). In addition, each has a serverless variant known as *AWS Fargate*, where AWS entirely operates the underlying container infrastructure on your behalf.

ECS runs Docker containers with no changes and supports both Windows and Linux containers. Note, however, that if you choose to use AWS Fargate, then currently only Linux containers are supported, so your application will need to be based on .NET Core or .NET 5+. If your Docker images are hosted in a third-party repository, you can still use ECS. Alternatively, you can upload your images to *Amazon Elastic Container Registry* (*ECR*). As with EC2 virtual machines, you can take advantage of AWS CodeDeploy to minimize downtime when updating applications by using blue/green deployments, with full support for deployment monitoring and rollback.

If you're using Kubernetes, then EKS will be of interest. With EKS, the Kubernetes control plane is run across multiple AZs, and it automatically detects and replaces unhealthy nodes. As with ECS, Windows is supported – first for worker nodes and for scheduling Windows containers. It's also possible to run Windows and Linux worker nodes together in the same cluster. As you might expect, EKS manages the availability and scalability of the control plane nodes for you, freeing you from infrastructure management and allowing you to focus on your application code and deployment topology. We will go deeper into the container offerings in a later chapter.

Finally, AWS offers a serverless compute service known as *AWS Lambda*. Lambda enables you to run your code without provisioning or managing servers – all of that is handled fully by AWS on your behalf. This allows you to focus entirely on your code, which can run in response to events such as an object being created in storage, or a request being received on a web API endpoint, and lots more (your code in Lambda can be triggered from a variety of events across 140 AWS services). You can also configure your code to run on a *cron* schedule, you can call it directly from within your own application code, or you can even invoke functions from the command line. You might find this latter feature useful in automation scripts, for example.

Note Lambda uses Linux as the underlying operating system host, so your code must target .NET Core 3.1 or .NET 5+; .NET Framework is not supported for writing Lambda functions.

Storage

Now that you have some idea of the range of compute options available to host your application code, let's move on to another common need of applications – storage.

When people think of storage in conjunction with AWS, *Amazon Simple Storage Service*, or *S3* as it's more commonly referred to, is usually what springs to mind. S3 is a service for object storage in the cloud. We use the term "object storage" to distinguish it from "file storage" – it's important to know that S3 is not a file system so it doesn't have a "real" directory structure or well-defined metadata such as the file name and date it was last modified. Obviously, it can store files, but as far as S3 is concerned, objects are just opaque blobs of data of varying lengths. To work effectively with S3, the key elements to understand are buckets, objects, and keys.

Buckets are the top-level organizational unit and are URL addressable – this means that buckets must conform to DNS name rules and, most importantly, they must have a globally unique name. If I have a bucket named `myapplicationbucket` in my account, you cannot have another bucket with the same name in your account.

Objects exist solely within buckets and represent the data you want to store. Objects can be any size up to 5 terabytes currently. You can add metadata tags to objects, as well as various types of access controls. Furthermore, it's possible to add life cycle policies to objects. One example of this would be a policy where older or less frequently accessed objects are automatically moved to different storage classes to take advantage of lower pricing.

A *key* is used to identify the different objects within a bucket. At its core, S3 is a key-value store operating at massive scale. Keys are probably what cause most people to think S3 is a file system when they first encounter it – object keys do in fact look like Linux file system paths. For example, consider the following key: `application.images/frontpage/logo.jpg`. Notice the / delimiter in the key – this divides the key into *prefixes* that resemble, but are not, subfolders. S3 keys are a flat structure, and there is no concept of sub-buckets or subfolders. However, using prefixes can help infer a logical structure. In fact, the AWS Management Console and the tools we'll meet in Chapter 2 and beyond use the prefixes to emulate a file system when working with S3. Figure 1-4 shows an example of object keys and prefixes in a bucket (note too that the console uses the term "Folder" for a prefix).

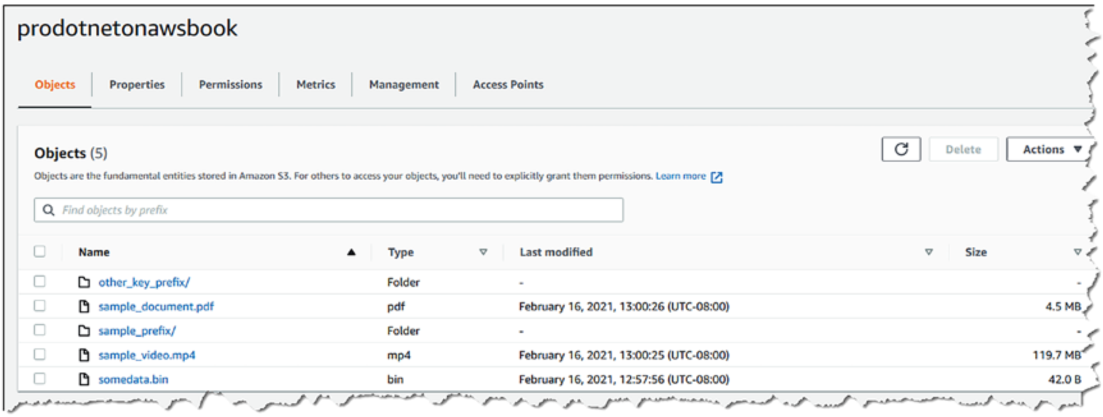

Figure 1-4. *Object keys and prefixes in an S3 bucket*

Listing 1-1 shows the output of a command (from the AWS Tools for PowerShell, which we will introduce in Chapter 2) to list the keys in the bucket. Here, you can see how the prefixes are in fact just parts of keys identifying the objects in storage and are not expressing any form of storage hierarchy.

Listing 1-1. Listing object keys

```
C:\> (Get-S3Object -BucketName prodotnetonawsbook).foreach("Key")

other_key_prefix/
other_key_prefix/image.png
sample_document.pdf
sample_prefix/
sample_prefix/file1.txt
sample_prefix/file2.txt
sample_video.mp4
somedata.bin
```

Even if you don't make use of S3 directly from your application, you'll still find that many AWS services make use of S3 themselves. For example, when you deploy your application to Elastic Beanstalk, the deployment bundle representing your built application must first be uploaded to an S3 bucket. Tooling usually handles this for you, behind the scenes, but it is something to be aware of.