O'REILLY®

Developing on AWS with C#

A Comprehensive Guide on Using C# to Build Solutions on the AWS Platform

Early Release
Raw & Unedited

Compliments of
aws

Noah Gift &
James Charlesworth

Developing on AWS with C#

A Comprehensive Guide on Using C# to Build Solutions on the AWS Platform

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Noah Gift and James Charlesworth

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

Developing on AWS with C#
by Noah Gift and James Charlesworth

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*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Suzanne McQuade
Development Editor: Melissa Potter
Production Editor: Beth Kelly
Copyeditor: TO COME
Proofreader: TO COME

Indexer: TO COME
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

Month Year: First Edition

Revision History for the Early Release
2022-05-24: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492095873* for release details.

978-1-492-09581-1

[???]

Table of Contents

CHAPTER 1

Migrating a Legacy .NET Framework Application to AWS

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *mpotter@oreilly.com*.

In the previous chapters we have seen some of the exciting tools and services that AWS gives us as developers. We are next going to take a look at what we can do with some of our legacy .NET applications and explore what is made possible by moving them to the cloud.

Software development is not, as I'm sure you are uncomfortably aware, a pursuit solely of greenfield projects, clean repos, latest toolsets and tidy backlogs. Organizations of all sizes can have legacy code, some of which may still be running on premises; Internal tools, APIs, workflows, applications that are actively used but not actively maintained. Migrating these to the cloud can provide your organization with cost savings, increased performance and a drastically improved ability to scale.

In this chapter you will learn how to choose, plan and execute the migration of a web application running on IIS and built on either .NET Framework or .NET Core/6+

With the release of .NET 5 in November 2020 Microsoft has renamed .NET Core to simply ".NET". In these next chapters we will refer to .NET Core and all future versions as .NET and the previous, legacy version of the framework as .NET Framework

Choosing a Migration Path

Every .NET application will have a different path to the cloud and, while we cannot create a one-size-fits-all framework for migrating a legacy .NET application, we *can* learn from the migrations of those that came before us. In 2011, the technology research company Gartner identified 5 migration strategies for migrating on premises software to the cloud[1]. These were known as The 5 R's and over the years have been refined, adapted and expanded as new experiences emerged, growing to encompass all the challenges you might face migrating and modernizing a legacy web application.

For migrating some of your code to AWS we now have 6 R's, any of which could be applied to a legacy .NET web application running on IIS and built in either .NET Framework or the more recent incarnation *.NET*.

- Rehosting
- Replatforming
- Repurchasing
- Rearchitecting
- Rebuilding
- Retaining

The first four of these strategies have increasing levels of effort and complexity, this however is rewarded with increasing value and ability to iterate going forwards. We will delve deeper into some of these approaches later in this chapter.

Rehosting

Rehosting is the process of moving an application from one host to another. It could be moving an application from running in a server room on a company's premises to a virtual machine in the cloud, or it could be moving from one cloud provider to another. As a strategy for migration, rehosting does not change (or even require access to) the source code. It is a process of moving assets in their final built or com-

1 Migrating Applications to the Cloud: Rehost, Refactor, Revise, Rebuild, or Replace? *https://www.gartner.com/en/documents/1485116/migrating-applications-to-the-cloud-rehost-refactor-revi*

piled state. In the .NET world this means `.dll` files, `.config` files, `.cshtml` views, static assets, and anything else required to serve your application. It is for this reason that rehosting is sometimes called the "lift and shift" approach to migration. Your entire application is lifted out as-is and shifted to a new host.

The advantages of rehosting your application include being able to take advantage of the cost savings and performance improvements possible on a cloud-hosted virtual machine. It can also make it easier to manage your infrastructure if you can rehost your lesser maintained or legacy applications alongside your more actively developed code on AWS.

For an overview of some of the tools and resources available, should you choose to follow this migration path, see "Rehosting On AWS" on page 12.

Replatforming

The replatforming approach goes one step further than simply rehosting and changes not just *where* but also *how* your application is hosted. Unlike rehosting, replatforming *could* involve changes to your code, although these changes should be kept to a minimum to keep the strategy viable.

There are many definitions of what constitutes a "platform", but one platform we as .NET developers are all aware of is Internet Information Services (IIS) running on Windows Server. Replatforming would be the process of migrating your application *away* from IIS and onto a more cloud native hosting environment such as Kubernetes. Later in this chapter we will explore one type of replatforming: "Replatforming via Containerization" on page 18.

Repurchasing

This strategy is relevant when your application depends on a licensed third party service or application that cannot run on a cloud infrastructure. Perhaps you use a self-hosted product for Customer Relationship Management (CRM) or Content Management System (CMS) functionality in your application that cannot be migrated to the cloud. Repurchasing is a migration strategy for applications that rely on these products and involves ending your existing, self-hosted license, and purchasing a new license for a cloud-based replacement. This can either be a cloud-based version of a similar product (for example Umbraco CMS to Umbraco Cloud), or a replacement product on the AWS Marketplace.

Rearchitecting

As the name implies, rearchitecting deals with the overall architecture of your application and asks you to think about how you can make changes to facilitate its move to

the cloud[2]. For a legacy .NET Framework application this will almost certainly mean moving to .NET. Microsoft in 2019 announced that version 4.8 will be the last major release of .NET Framework and, while it will continue to be supported and distributed with future releases of Windows, it will not be actively developed by Microsoft[3]

History has sculpted out a fairly linear journey for rearchitecting a monolithic web application as shown in Figure 1-1

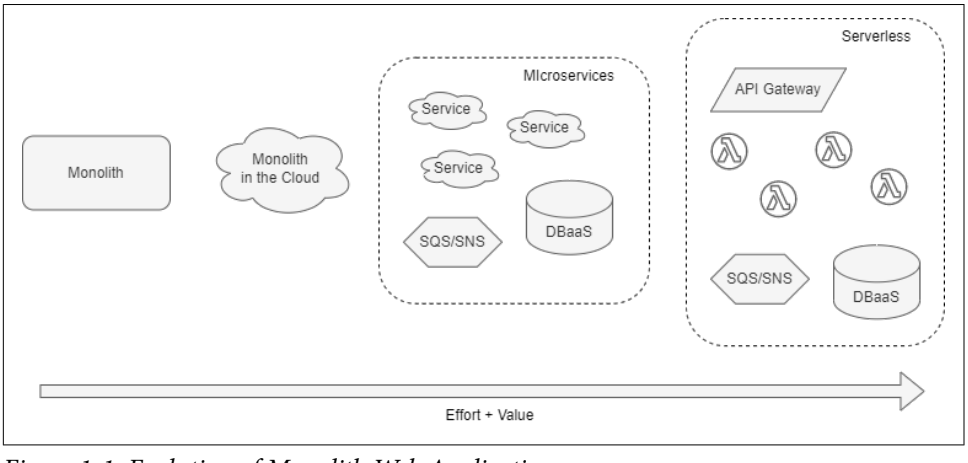

Figure 1-1. Evolution of Monolith Web Applications

We will take a deeper dive into porting .NET Framework to .NET in the section "Rearchitecting: Moving to .NET (Core)" on page 22

Rebuilding

Sometimes your legacy codebase fails to pass the effort vs value benchmark for migration and you have no choice but to rebuild from scratch. You are not, of course, starting entirely from scratch. You will be able to migrate some of your business logic; all those solved problems your legacy codebase has taken years to navigate through can be recreated on a new codebase. The code itself however, the architecture, the libraries, databases, API schemas and documentation[4] will not be coming with you.

2 This migration strategy is sometimes called "Refactoring" however, this can be a chameleon of a term so I'll be sticking to "Rearchitecting" for this book.

3 You can find the .NET Framework support policy at *https://dotnet.microsoft.com/en-us/platform/support/policy/dotnet-framework*

4 I will admit to looking up from my screen and taking a long sip of coffee before adding "documentation" to this list. My cat is giving me the same knowing stare that you are.

Retaining

The final migration strategy on this list is really just *none of the above*. Perhaps your legacy application has some special requirements, cannot be connected to the internet, will have to go through a prohibitively lengthy recertification process. There are many unique and often unforeseen reasons why some legacy codebases cannot be migrated to the cloud, or cannot be migrated at the present moment in time. You should only migrate applications for which a viable business case can be made and if that is not possible then selecting *none of the above* is sometimes your best option.

Choosing a Strategy

The migration strategy you choose will depend upon the current architecture of your application, where you want to get to, and how much you are willing to change in order to get there. The chart in Figure 1-2 summarizes the decisions you can make in order to choose the migration path most appropriate for your individual use case.

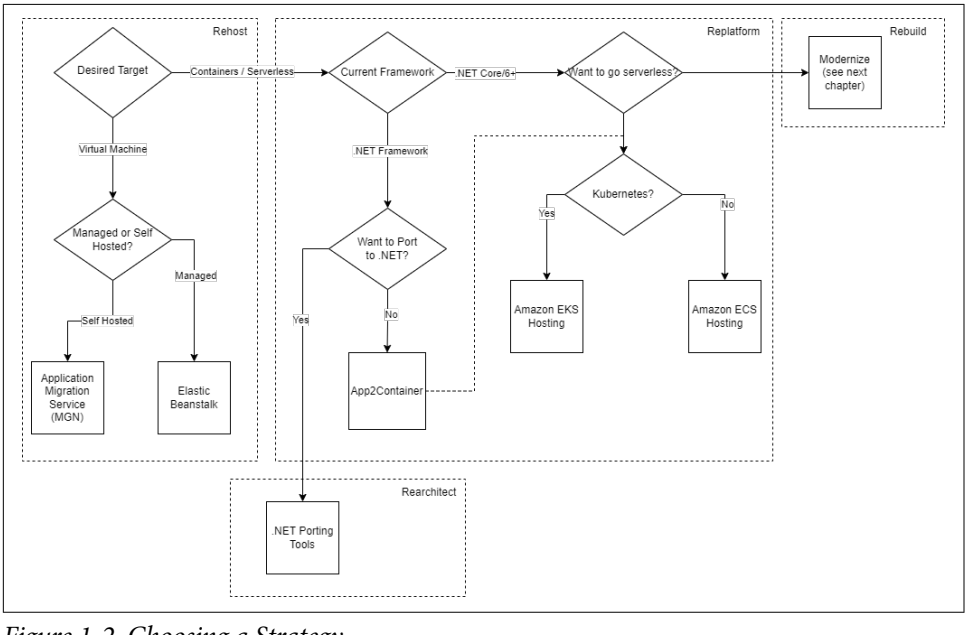

Figure 1-2. Choosing a Strategy

AWS Migration Hub Strategy Recommendations

For further assistance in choosing a migration strategy, AWS offers a tool called the Strategy Recommendations Service. This service gathers data from your existing servers, augments it with analysis of your source code and SQL schemas, then recommends a migration strategy from those we have covered previously.

The Strategy Recommendations Service is part of the AWS Migration Hub: a set of tools for analyzing an infrastructure planning for and then tracking a migration to AWS. The Migration Hub is available at no additional charge, you only pay the cost of any tools you use and any AWS resources consumed in the process

To get started with the Strategy Recommendation Service we first need to give it as much data about our existing infrastructure as possible. We can do this with the AWS Application Discovery Service, another service accessible from the AWS Migration Hub. To start application discovery, navigate to discovery tools in the AWS Management Console for your home region[5] and choose from one of three methods to collect the data shown in Figure 1-3.

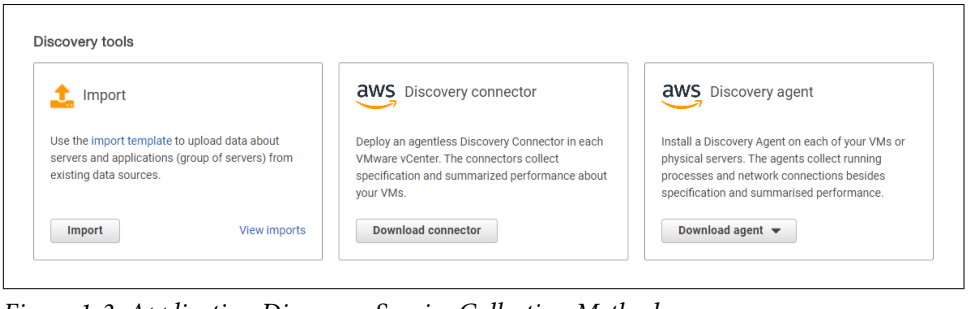

Figure 1-3. Application Discovery Service Collection Methods

Let's walk through a quick setup using the discovery agent. Before we start, ensure you have the AWS CLI installed and you have an IAM user access key (and secret). You can use the following commands to save these values in your AWS configuration files. These settings will then be used by all the tools in this chapter.

```
$ aws configure set aws_access_key_id <your-key-id>
$ aws configure set aws_secret_access_key <your-key-secret>
$ aws configure set default.region <home-region>
```

Next, open up a PowerShell terminal on one of the Windows servers you want to begin collecting data for, then download the Agent Installer.

```
PS C:\> mkdir ADSAgent
PS C:\> cd .\ADSAgent\
PS C:\ADSAgent> Invoke-WebRequest
https://s3.us-west-2.amazonaws.com/aws-discovery-agent.us-west-2/windows/latest/
ADSAgentInstaller.exe -OutFile ADSAgentInstaller.exe
```

5 Your home region in the AWS Migration Hub is the region in which migration data is stored for discovery, planning, and migration tracking. You can set a home region from the Migration Hub Settings page

Next set your home region for the AWS Migration Hub, your access key ID and secret.

And run the Discovery Agent Installer on this server

```
PS C:\ADSAgent> .\ADSAgentInstaller.exe REGION=$AWS_REGION KEY_ID=$KEY_ID
KEY_SECRET=$KEY_SECRET INSTALLLOCATION="C:\ADSAgent" /quiet
```

This will install the agent into the new folder you created `C:\ADSAgent`. Back in the Migration Hub section of the AWS Management Console you can navigate to *Discover* > *Data Collectors* > *Agents* and, if all has gone well, the agent you installed should appear in the list. Select the agent and click "Start Data Collection" to allow ADS to begin collecting data about your server

If your agent does not show up, ensure your server is allowing the agent process to send data over TCP port 443 to https://arsenal-discovery.<*your-home-region*>.amazonaws.com:443

The discovery agent will poll its host server approximately every 15 minutes and report data including CPU usage, free RAM, operating system properties and process IDs of running processes that were discovered. You will be able to see your servers in the Migration Hub by navigating to *Discover* > *Servers* on the dashboard. Once you have all your servers added to ADS you are ready to begin collating the data necessary for strategy recommendations.

The Strategy Recommendations Service has an automated agentless data collector you can use to analyze your .NET applications running on the servers you now have in ADS. To get started navigate to *Strategy* > *Get Started* in the migration hub console and follow the wizard to download the data collector as an Open Virtual Application (OVA). This can then be deployed to your VMware vCenter Server. Full instructions for setting up the data collector can be found at *https://docs.aws.amazon.com/migrationhub-strategy/latest/userguide/getting-started.html*.

Once your data collector is set up you can move to the next page of the wizard and select your priorities for migration. Figure 1-4 shows the priorities selection screen in the Strategy Recommendation Service. This will allow AWS to recommend a migration strategy that best fits your business needs and plans for the future. Select the options that most align with your reasons for migrating.

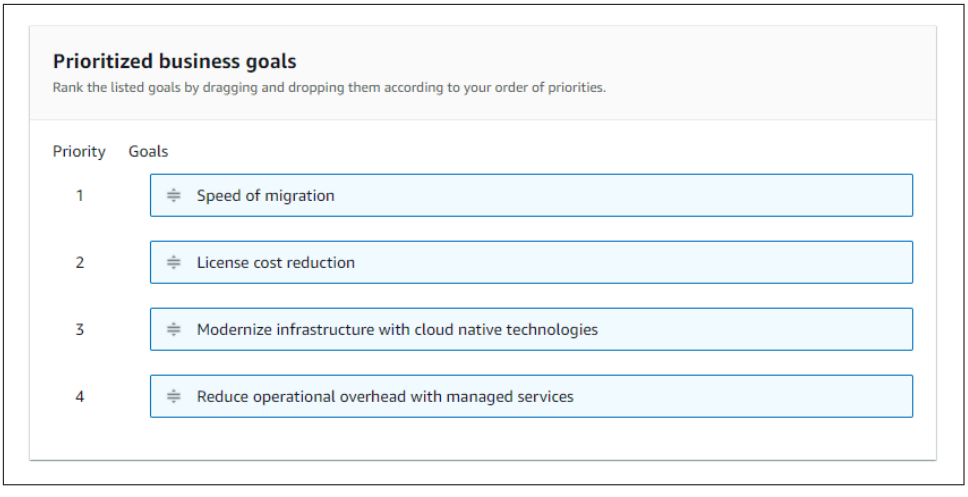

Figure 1-4. Strategy Recommendation Service Goals

After running data analysis on your servers the service will give you recommendations for each application, including a link to any relevant AWS tools to assist you with that type of migration. In Figure 1-5 you can see the tool is recommending we use *Rehosting* as a migration strategy onto Elastic Compute (EC2) using the *Application Migration Service*, which we will look at next.

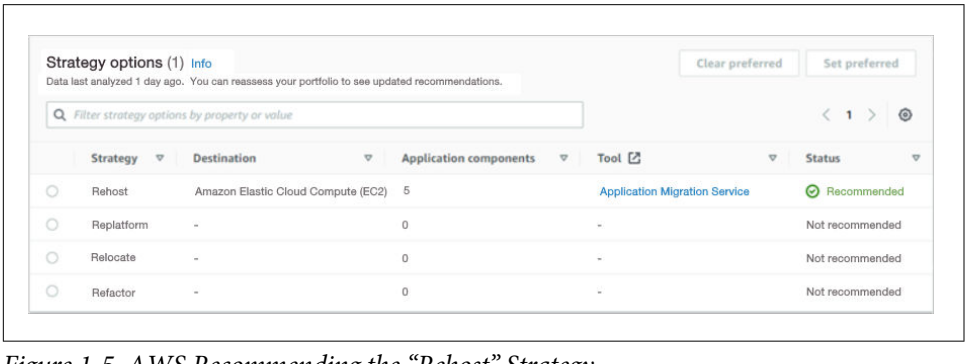

Figure 1-5. AWS Recommending the "Rehost" Strategy

Rehosting On AWS

The approach for rehosting your legacy .NET application onto AWS will vary depending on how your application is currently hosted. For web apps deployed to a single server running IIS you can easily replicate this environment on an Amazon Elastic Compute Cloud (EC2) instance on AWS. If your .NET application is currently deployed to a managed environment (an "App Service" as other cloud providers might call it) then the equivalent on AWS is *Elastic Beanstalk* and you should find the

experience of working with Elastic Beanstalk familiar. We will cover migrating managed hosting to Elastic Beanstalk later on in the section "Elastic Beanstalk" on page 16 but first, let's take a look at the case of rehosting a virtual machine running IIS over to EC2.

Application Migration Service (MGN)

The latest AWS offering for performing a lift-and-shift migration to EC2 is called the *Application Migration Service* or MGN[6]. This service evolved from a product called CloudEndure that AWS acquired in 2018. CloudEndure is a disaster recovery solution that works by creating and maintaining replicas of your production servers on AWS EC2 and Elastic Block Store (EBS). This replication concept can be repurposed for the sake of performing a lift and shift rehosting. You simply set up replication to AWS then, when you are ready, switch over to running your application exclusively from your AWS replicas, allowing you to decommission the original server. An overview of how the Application Migration Service replicates your servers is shown in Figure 1-6

6 Interestingly, the three letter abbreviation *MGN* for Application Migration Service is a contraction and not an initialism. Perhaps *AMS* was too similar to *AWS*.

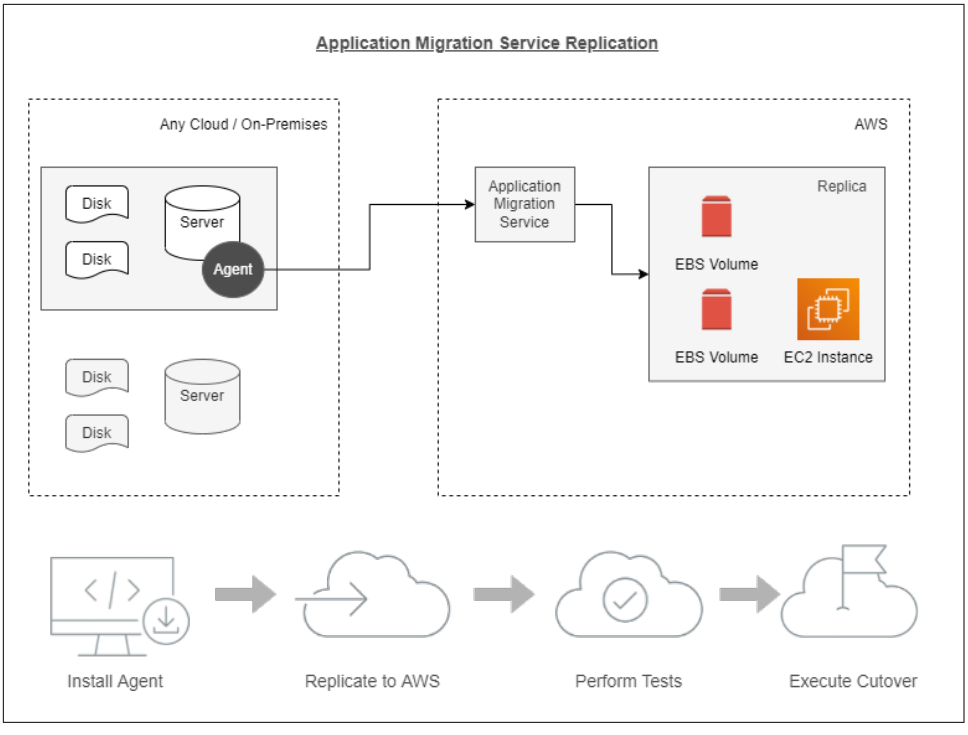

Figure 1-6. Overview of the Application Migration Service

The Application Migration Service is accessed via the AWS management console, type "MGN" into the search or find it in the menu of the Migration Hub as in Figure 1-7

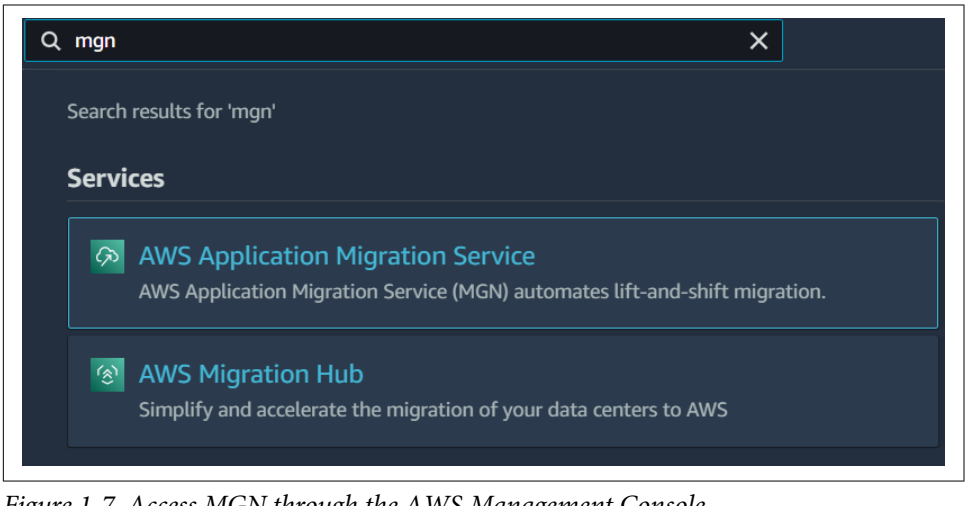

Figure 1-7. Access MGN through the AWS Management Console

Set up of the Application Migration Service begins by installing the replication agent onto your servers, similar to how we installed the discovery agent for the Application Discovery Service. First log onto your Windows server and open up a PowerShell window as Administrator to download the agent installer for Windows. Replace `<region>` with the AWS region you would like to migrate your servers into.

```
PS C:\> mkdir MGNAgent
PS C:\> cd .\MGNAgent\
PS C:\MGNAgent> Invoke-WebRequest
https://aws-application-migration-service-<region>.s3.<region>.amazonaws.com
/latest/windows/AwsReplicationWindowsInstaller.exe
 -OutFile C:\MGNAgent\AwsReplicationWindowsInstaller.exe
```

Next put your region, access key ID and secret into variables if you haven't already and execute the installer

```
PS C:\ADSAgent> $AWS_REGION="<region>"
PS C:\MGNAgent> $KEY_ID="<your-key-id>"
PS C:\MGNAgent> $KEY_SECRET="<your-key-secret>"

PS C:\MGNAgent> .\AwsReplicationWindowsInstaller.exe --region $AWS_REGION
 --aws-access-key-id $KEY_ID --aws-secret-access-key $KEY_SECRET
```

The agent installer will ask you which disks on this server you want to replicate and then will get to work syncing your disks to AWS. You can see the status of the agent installer operations in the console window as shown in the following Figure 1-8.

Figure 1-8. MGN Replication Agent Console Window

If you head back over to the MGN Management Console you will be able to see your server under *Source Servers* in the menu. Click on your server name and you can see the status of the replication for this server as shown in Figure 1-9. It can take a while to get through all the stages but once complete the status in the console will change to "Ready For Testing". One nice feature of the Application Migration Service is the ability to spin up an instance of a server from a replica and test that everything is as you expect it to be, without interrupting or otherwise interfering with the replication itself.

Figure 1-9. MGN Replication Status in the Management Console

To test a server select *Launch test instances* from the *Test and Cutover* action menu of your source server. This will launch a new EC2 instance that should mirror the original Windows server you replicated, with transfer of licenses handled automatically by the Application Migration Service. You can connect to the EC2 instance with Remote Desktop (RDP) by selecting it in the list of EC2 instances from the management console once it becomes ready. When you are happy that the test instance is working the way you expect it to you can execute the final stage of the migration: Cutover.

If you refer back to the stages of application migration via replication in Figure 1-6 you can see the final stage being *Execute Cutover*. This is where we create the resources for all our source servers (that is: spin up a new EC2 instance for each server), stop replication of our old servers, and allow us to decommission the original servers we installed the replication agents onto.

So now we have all our servers running like-for-like on EC2 we have performed the lift and shift rehosting, what's next? Staying inside the realm of *rehosting*, we can go one step further and take advantage of AWS's managed environment for running a web application: Elastic Beanstalk.

Elastic Beanstalk

Elastic Beanstalk is a *managed* hosting environment for your web application. It supports a variety of backend stacks such as Java, Ruby or PHP but it is the .NET Framework support that we are most interested in here. The difference between hosting our app on a Windows server on EC2, as we did in the previous section, and using Elastic Beanstalk can be distilled down to one key discrepancy:

With an unmanaged server you upload your compiled and packaged website files to a server and then tweak the settings on that server in order to handle web traffic. With

a managed service you upload your package to the cloud and the service will take care of the rest for you, setting up load balancers and dynamically scaling virtual machines horizontally. Managed services are the real draw for deploying your applications to the cloud and the more you lean into having AWS manage your infrastructure for you, the more you can concentrate on just writing code and solving problems for your business. Later in this book we will cover serverless programming and how you can architect your .NET applications to be more serverless, a concept rooted in managed services as much as possible. You can think of Elastic Beanstalk as an equivalent to "App Service" which you may be familiar with from a slightly *bluer* cloud provider.

So let's get started with our first managed service on Elastic Beanstalk. The simplest way to deploy code is by using the AWS Toolkit for Visual Studio. With the AWS Toolkit installed, deploying to Elastic Beanstalk really is as simple as right clicking your solution and selecting "Publish to Elastic Beanstalk". Figure 1-10 Shows the toolkit in use in the Solution Explorer of Visual Studio.

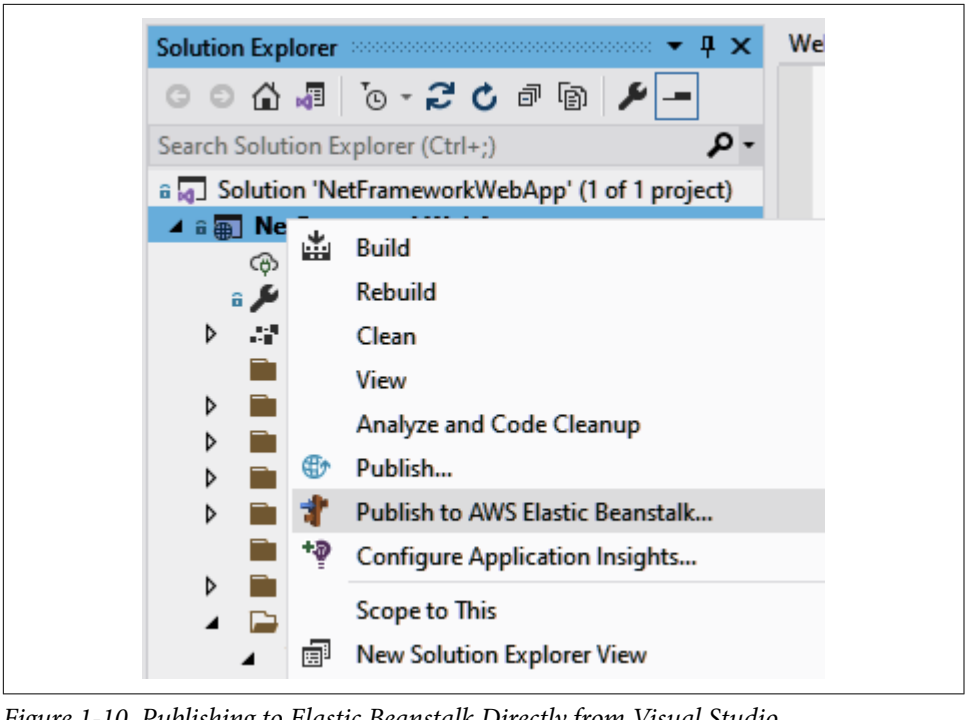

Figure 1-10. Publishing to Elastic Beanstalk Directly from Visual Studio

There is of course also a CLI tool you can use in your CI pipeline. To get started with the Elastic Beanstalk CLI see the EB CLI Installer on GitHub. The CLI tool allows you to publish your code directly to Elastic Beanstalk from AWS CodeBuild or GitHub.

Lastly, there is one final tool worth mentioning if you are thinking of trying out Elastic Beanstalk and that is the Windows Web App Migration Assistant (WWMA). This is a PowerShell script that you can run on any Windows Server with an IIS hosted web application to automatically migrate the application onto Elastic Beanstalk *without needing the source code at all*. This is useful if you have a legacy .NET application that is no longer maintained and you want to take advantage of the benefits of Elastic Beanstalk but you no longer perform releases for this app. It is a true *rehosting* tool that simply moves the compiled website assets from your `C:\inetpub` folder on the server into EC2 instance(s) managed and scaled by Elastic Beanstalk.

Replatforming via Containerization

As a migration strategy, replatforming is concerned with looking at the *platform* on which our .NET application is running and exploring moving somewhere else. In the case of a .NET Framework web application that *platform* will be some version of Windows Server running IIS. While previously we looked at Elastic Beanstalk as a way of getting more out of this infrastructure, Elastic Beanstalk still hosts your application on IIS on Windows Server, albeit in a much more scalable and efficient way. If we want to really push the envelope for scalability and performance, while still keeping away from the original source code (these are "legacy" applications after all) then we need to move away from IIS and onto something else. This is where containerization comes in.

I'm going to skip over exactly what containerization is and why it matters as we have Chapter 5 later in this book dedicated to Containerization of .NET, suffice to say moving your legacy .NET application from Windows Server web hosting to containers unlocks both performance and cost benefits to your organization, all without having to touch any of that legacy code.

App2Container

App2Container is a command line tool from AWS that runs against your .NET application running on IIS on a Windows Server. It analyzes your application and dependencies, then creates Docker container images that can be deployed to an orchestration service in the cloud such as Elastic Container Service (ECS) or Amazon Elastic Kubernetes Services (Amazon EKS). Because App2Container runs on an application that is already deployed to a server, it doesn't need access to your source code and sits right at the end of a deployment pipeline. For this reason App2container is perfect for quickly replatforming an old application that is not being actively developed and you don't want to be rebuilding, simply skip right to the last two steps of the pipeline shown in Figure 1-11 and containerize the production files.

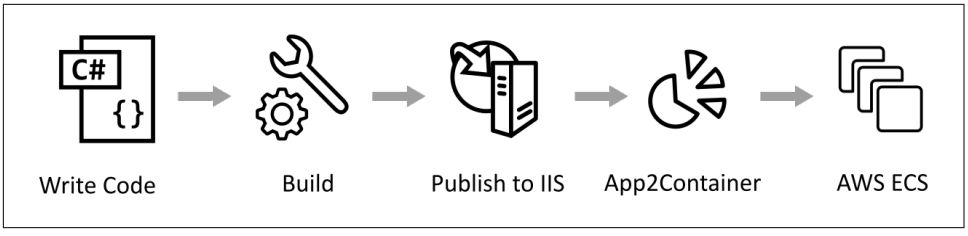

Figure 1-11. Deployment Pipeline of .NET Framework Application using App2Container

To containerize a .NET application, you first need to download and install App2Container onto the server running your application. You can find the installation package on AWS's website at *https://docs.aws.amazon.com/app2container/latest/UserGuide*. Download, unzip and run `.\install.ps1` from an Administrator Powershell terminal on the application server. This will install the `app2container` command line utility. If you haven't already, make sure your application server has the *AWS Tools for Windows PowerShell* installed and you have a default profile configured that allows you access to manage AWS resources from your application server. If your server is running on an EC2 instance (for example if you rehosted it using the Application Migration Service, see "Rehosting On AWS" on page 12) then these tools will already be installed as they are included on the Windows-based machine images used in EC2. Once you have confirmed you have an AWS profile with IAM permissions to manage AWS resources you can initialize App2Container by running

```
PS C:\> app2container init
```

The tool will ask about collecting usage metrics and ask you for an S3 bucket to upload artifacts, but this is entirely optional and you can skip through these options. Once initialized you are ready to begin analyzing your application server for running .NET applications that can be containerized. Run the inventory command to get a list of running applications in JSON format, and then pass in the *JSON key* as the `--applcation-id` of the app you want to containerize as shown in Figure 1-12

```
PS C:\> app2container inventory
PS C:\> app2container analyze --application-id iis-example-d87652a0
```

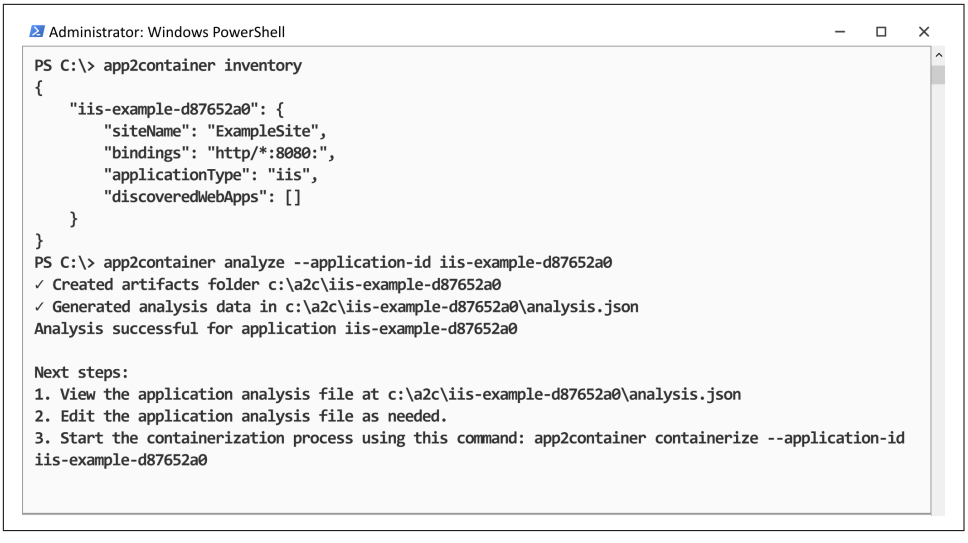

Figure 1-12. Listing the IIS Sites capable of being containerized

We are encouraged to take a look at the `analysis.json` file that is generated for us, and I echo that sentiment. A full list of the fields that appear in `analysis.json` can be found in the App2Container user guide[7] but it is worth spending the time exploring the analysis output as these settings will be used to configure our container. You can edit the `containerParameters` section of `analysis.json` before containerizing if required. It is also worth opening up `report.txt` in the same folder as this is where any connection strings will be added by the analyze command. When you are ready, run the `containerize` and `generate` commands to build a docker image then generate all the artifacts you need to deploy it to either ECS or EKS[8].

```
# Create a Dockerfile
PS> app2container containerize --application-id iis-example-d87652a0

# Generate a deployment
PS> app2container generate app-deployment --application-id iis-example-d87652a0
```

The second command here (`generate app-deployment`) will upload your containers to Elastic Container Registry (ECR) and create a CloudFormation template you can use to deploy your app to (in this case) Elastic Container Service (ECS). The tool will

7 *https://docs.aws.amazon.com/app2container/latest/UserGuide/config-containers.html*

8 We will revisit these two container orchestration services later in this book, but in a nutshell ECS is a simpler and more "managed" service for running containers without the added complexity of dealing with Kubernetes.

show you the output destination of this CloudFormation template (see Figure 1-13) and give you the command you need to deploy it to AWS.

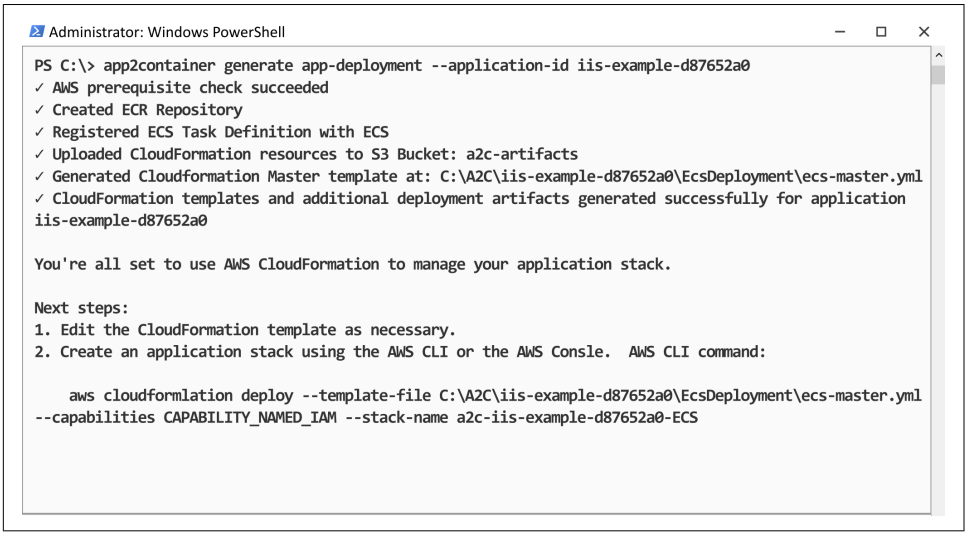

Figure 1-13. Results of App2Container deployment generation

This has been a brief overview of App2Container from AWS but much more is possible with this tool than we have covered here. Instead of performing the containerization on the application server itself, App2Container also lets you deploy a worker machine either to EC2 or your local virtualization environment. This would be useful if you wanted to protect the application server, which could be serving a web application in production, from having to spend resources executing a containerization process. Since App2Container is a CLI tool it would also be simple to integrate into a full deployment pipeline for code you are still actively working on and releasing changes to. If you refer back to Figure 1-11 you can see how App2Container can be used to *extend* an existing .NET deployment pipeline into containerization without touching anything further upstream, including your code.

One final note on App2Container is framework version support which has been expanding with new releases of the tool. You can use App2Container on both .NET Framework and .NET Core/5/6+ applications running on both Windows and, more recently, Linux. For .NET Framework the minimum supported version is .NET 3.5 running in IIS 7.5. Java applications can also be containerized with App2Container in as similar way to that we have explored here, so it's not just us C# developers that can benefit.

Rearchitecting: Moving to .NET (Core)

So far we have looked at migration approaches for our .NET applications that do not involve making changes to the code, but what can we do if changing the code is acceptable? In the next chapter we will be looking at Modernizing .NET applications to Serverless however, if your application is still built on .NET Framework the first step down every modernization path will almost certainly be a migration to .NET 6+. The road ahead for .NET Framework applications is not a long one and, aside from the rehosting and replatforming approaches we have covered in this book, you will eventually be approaching the topic of migrating framework versions. .NET is, after all, a complete rewrite of Microsoft's framework and feature parity was not an aim. APIs have changed, namespaces like `System.Web.Services` are no longer present and some third party libraries that you rely on may not have been migrated, forcing you to replace them with an alternative. For these reasons it is vital to do as much investigation as possible in order to assess the lift required in migrating your legacy .NET Framework application to modern .NET.

While there is no such thing as a tool that will automatically refactor your entire solution and convert your .NET Framework monolith to .NET 6+, what does exist are a handful of extremely useful tools to analyze your project, perform small refactoring tasks, and give you an insight into where you will find compatibility problems. I'm going to give you a brief insight into two of these tools: the .NET Upgrade Assistant from Microsoft, and the Porting Assistant from AWS.

Before you start however, it is worth becoming familiar with which .NET Framework technologies are unavailable on .NET 6+[9]. These include almost everything that used the Component Object Model (COM, COM+, DCOM) such as .NET Remoting and Windows Workflow Foundation. For applications that rely heavily on these Windows-only frameworks one of the migration strategies we discussed earlier in this chapter may be more appropriate. Applications that use Windows Communication Foundation (WCF) can take advantage of the CoreWCF[*https://github.com/CoreWCF/CoreWCF*] project in order to continue using WCF features on modern .NET.

Microsoft .NET Upgrade Assistant

With the releases of .NET 5 and 6 Microsoft cemented its vision for a unified single framework going forwards, with .NET 6 being the long term support (LTS) release of the platform. In order to assist migration of .NET Framework applications to this new, unified version of the framework Microsoft has been developing a command line tool called the *.NET Upgrade Assistant*. The Upgrade Assistant is intended to be a

9 *https://docs.microsoft.com/en-us/dotnet/core/porting/net-framework-tech-unavailable*

single entry point to guide you through the migration journey and wraps within it the more longstanding .NET Framework conversion tool *try-convert*. It is a good idea to use *try-convert* from within the context of the Upgrade Assistant as you will get more analysis and guidance towards the strategies most applicable to your project.

The types of .NET Framework applications that this tool can be used with at time of writing are:

- .NET Class libraries
- Console Apps
- Windows Forms
- Windows Presentation Foundation (WPF)
- ASP.NET MVC Web Applications

The .NET Upgrade Assistant has an extensible architecture that encourages the community to contribute extensions and analyzers / code fixers. You can even write your own analyzers to perform automatic code refactoring based on rules you define. The Upgrade Assistant comes with a set of default analyzers that look for common incompatibilities with your code and offer a solution. For example, the `HttpContextCurrentAnalyzer` looks for calls to the static `System.Web.HttpContext.Current`, a pattern often employed in controller actions of .NET Framework applications that will need to be refactored since `HttpContext.Current` was removed in .NET Core. In Figure 1-14 you can see an example of the message this analyzer emits when `HttpContext.Current` is found in your code.

So let's get going with an upgrade. For this example I have created a very simple ASP.NET MVC Web Application in Visual Studio 2019 using .NET Framework version 4.7.2. There is a controller action that takes the query string from the web request, adds it to the `ViewBag`, then displays it on the *Index.cshtml* Razor view. The output of this website can be seen in Figure 1-14

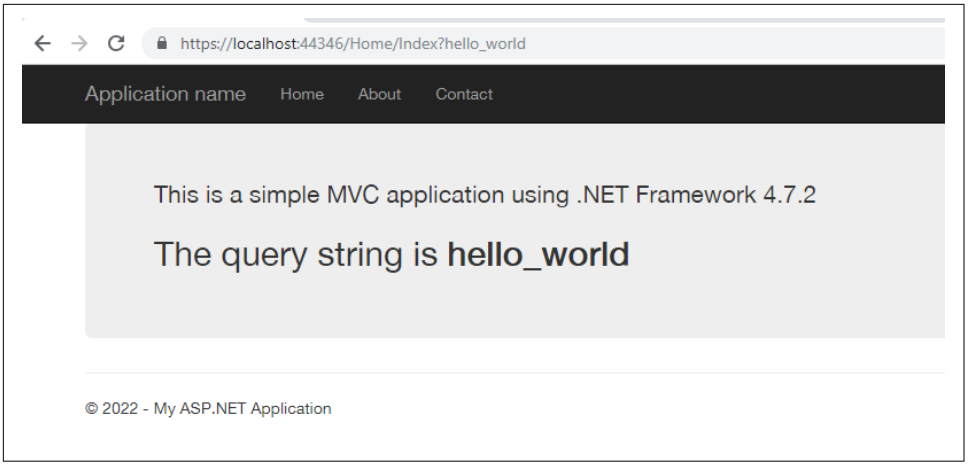

Figure 1-14. Example ASP.NET MVC Website Running in a Browser

HomeController.cs

```
using System.Web.Mvc;

namespace NetFrameworkMvcWebsite.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.QueryString = System.Web.HttpContext.Current.Request.QueryString;

            return View();
        }

    }
}
```

Index.cshtml

```
@{
    ViewBag.Title = "Home Page";
}

<div class="jumbotron">
    <p class="lead">This is a simple MVC application using .NET Framework 4.7.2</p>

    <h2>The query string is <strong>@ViewBag.QueryString</strong></h2>
</div>

<hr />

@WriteIndexFooter("Footer written using a Razor helper")
```

```
@helper WriteIndexFooter(string content)
{
    <footer>
        <p>@content</p>
    </footer>
}
```

I have purposefully added a couple of things to this example that I *know* are not compatible with .NET Core and later versions of the framework. Firstly, as introduced earlier we have a call to `HttpContext.Current` on line 9 of *HomeController.cs*. This will need to be replaced with a call to an equivalent HTTP context property in .NET 6. We also have a Razor helper in *Index.cshtml*, the `@helper` syntax for which is not present in later versions of .NET. I have this code checked into a Git repository with a clean working tree, this will help view the changes to the code that the .NET Upgrade Assistant will make by using a Git diff tool.

To get started with the .NET Upgrade Assistant, first install it as a .NET CLI tool:

```
dotnet tool install -g upgrade-assistant
```

Next, in the directory that contains your solution file run the upgrade assistant and follow the instructions to select the project to use as your entry point.

```
upgrade-assistant upgrade NetFrameworkMvcWebsite.sln
```

Depending on the type of .NET Framework project you have (WebForms, WPF etc) the upgrade assistant will give you a different set of steps. I have an ASP.NET MVC Web Application so we are offered a 10 step process for the upgrade as seen in Figure 1-15. Most of the steps are self explanatory, step 2 in my example is "Convert project to SDK style". This means reformatting the `.csproj` file to the newer .NET format that starts with `<Project Sdk="Microsoft.NET.Sdk">`. As you go through these steps, use a source control diff tool such (eg KDiff, VSCode or the "Git Changes" window in Visual Studio) to see the changes being made to your code and project files.

```
[12:29:48 INF] Initializing upgrade step Move to next project

Upgrade Steps

Entrypoint: D:\Code\CSharpBookExamples\NetFrameworkMvcWebsite\NetFrameworkMvcWebsite.csproj
Current Project: D:\Code\CSharpBookExamples\NetFrameworkMvcWebsite\NetFrameworkMvcWebsite.csproj

1. [Complete] Back up project
2. [Complete] Convert project file to SDK style
3. [Complete] Clean up NuGet package references
4. [Complete] Update TFM
5. [Complete] Update NuGet Packages
6. [Complete] Add template files
7. [Complete] Upgrade app config files
    a. [Complete] Convert Application Settings
    b. [Complete] Convert Connection Strings
    c. [Complete] Disable unsupported configuration sections
    d. [Complete] Convert system.web.webPages.razor/pages/namespaces
8. [Complete] Update Razor files
    a. [Complete] Apply code fixes to Razor documents
    b. [Complete] Replace @helper syntax in Razor files
9. [Complete] Update source code
    a. [Complete] Apply fix for UA0001: ASP.NET Core projects should not reference ASP.NET namespaces
    b. [Complete] Apply fix for UA0002: Types should be upgraded
    c. [Complete] Apply fix for UA0005: Do not use HttpContext.Current
    d. [Complete] Apply fix for UA0006: HttpContext.DebuggerEnabled should be replaced with System.Diagnostics.Debug
ger.IsAttached
    e. [Complete] Apply fix for UA0007: HtmlHelper should be replaced with IHtmlHelper
    f. [Complete] Apply fix for UA0008: UrlHelper should be replaced with IUrlHelper
    g. [Complete] Apply fix for UA0010: Attributes should be upgraded
    h. [Complete] Apply fix for UA0012: 'UnsafeDeserialize()' does not exist
10. [Next step] Move to next project

Choose a command:
   1. Apply next step (Move to next project)
   2. Skip next step (Move to next project)
   3. See more step details
   4. Configure logging
   5. Exit
>
```

Figure 1-15. Microsoft .NET Upgrade Assistant Complete

The logs from the upgrade assistant are stored in Compact Log Event Format (CLEF) inside the directory in which you ran the tool. It will also create a backup of your project however this is not particularly useful if you have everything checked into source control (you *do* have everything checked into source control right?).

You can see from this screenshot of the completed upgrade that step **9c.** was *Do not use HttpContext.Current*. This is coming from the HttpContextCurrentAnalyzer we introduced earlier and the fix from this analyzer will change all usage of `HttpContext.Current` in your code to `HttpContextHelper.Current`. We do still get a warning about `HttpContextHelper.Current` being obsolete and to use dependency injection instead however, this doesn't prevent my upgraded code from compiling. The upgrade assistant also refactored my `@helper` syntax in the Razor view (step **8b.** in Figure 1-15) and replaced it with a .NET 6 compatible helper method. The code after running the upgrade assistant looks like this:

HomeController.cs

```
namespace NetFrameworkMvcWebsite.Controllers
{
    public class HomeController : Microsoft.AspNetCore.Mvc.Controller
```

```
    {
        public Microsoft.AspNetCore.Mvc.ActionResult Index()
        {
            ViewBag.QueryString = HttpContextHelper.Current.Request.QueryString;

            return View();
        }

    }
}
```

Index.cshtml

```
@using Microsoft.AspNetCore.Mvc.Razor
@{
    ViewBag.Title = "Home Page";
}

<div class="jumbotron">
    <p class="lead">This is a simple MVC application using .NET Framework 4.7.2</p>

    <h2>The query string is <strong>@ViewBag.QueryString</strong></h2>
</div>

<hr />

@WriteIndexFooter("Footer written using a Razor helper")

@{ HelperResult WriteIndexFooter(string content)
        {
            <footer>
                <p>@content</p>
            </footer>
                return new HelperResult(w => Task.CompletedTask);
    }
}
```

AWS Porting Assistant

Another tool you can use to aid your migration from .NET Framework is the Porting Assistant provided by AWS themselves. This is a Windows application that you download to the machine on which you have your .NET Framework solution. Although the porting assistant runs locally on your code it will need to connect to AWS in order to retrieve NuGet package upgrade information from an S3 bucket, it is for this reason that you need to set it up with a local AWS profile as shown in Figure 1-16. No resources will be created on your AWS profile. The porting assistant can be downloaded from The porting assistant can be downloaded from the AWS Porting Assistant page[*https://aws.amazon.com/porting-assistant-dotnet*].

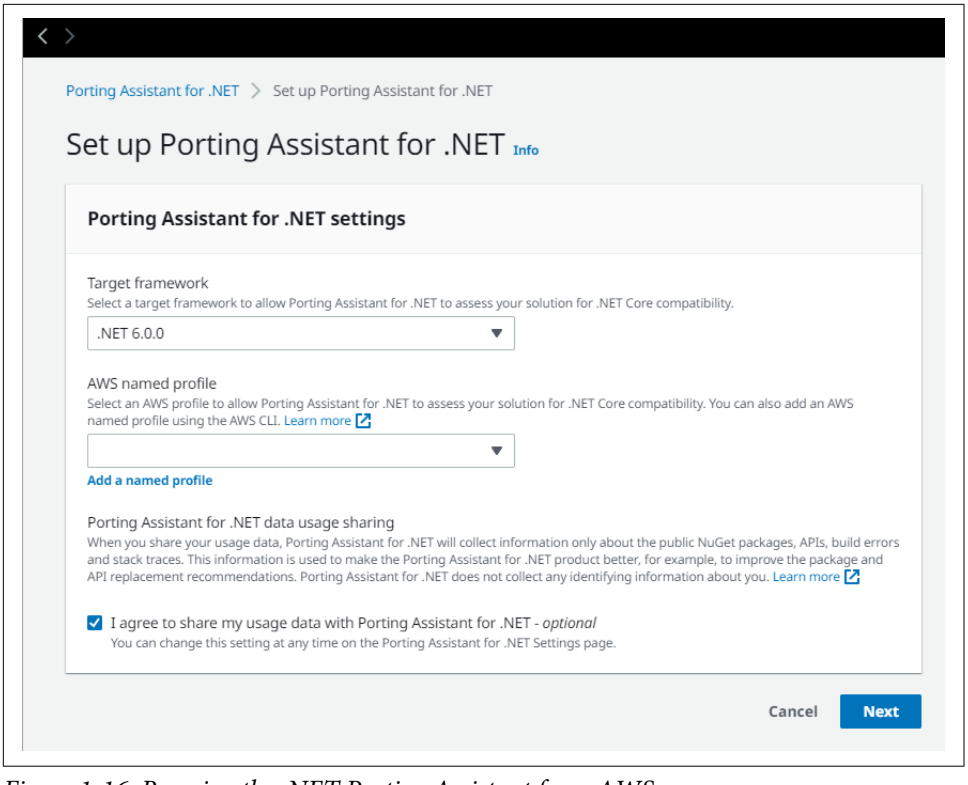

Figure 1-16. Running the .NET Porting Assistant from AWS

If we step through the wizard and use the same ASP.NET MVC Web Application we used for the Microsoft Upgrade Assistant we can see that it has correctly identified the solution is targeting .NET Framework 4.7.5, we have seven incompatible NuGet packages and fifteen incompatible APIs. This is the *15 of 17* values highlighted in Figure 1-17 You can see more information about these in the relevant tabs of the Porting Assistant including links to the code that has been identified as incompatible.

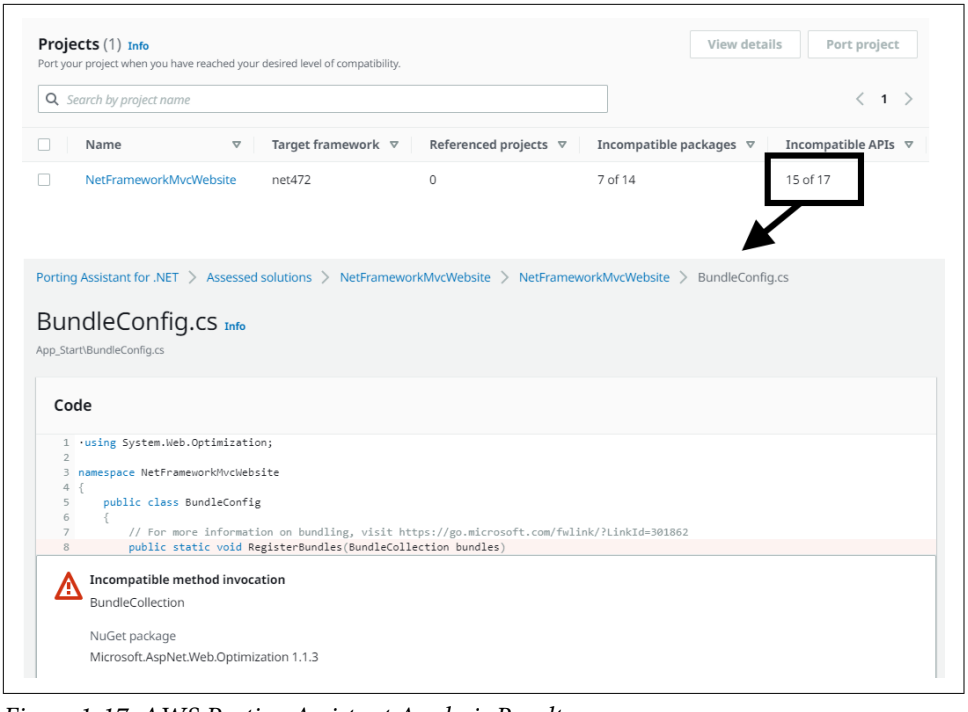

Figure 1-17. AWS Porting Assistant Analysis Results

When you are ready, click *Port Solution* to begin making the changes to your project files. The application will ask you where you would like to save the ported solution to, having your code in source control means you can choose "Modify source in place". Unlike Microsoft's tool, the AWS Porting does actually let you fine tune which versions you would like to upgrade your NuGet packages to in the UI. For this example I have simply left the defaults in place for all packages and stepped through the assistant. You can see now from Figure 1-18 that the project files have been upgraded and the project is now showing in the porting assistant as .NET 6.

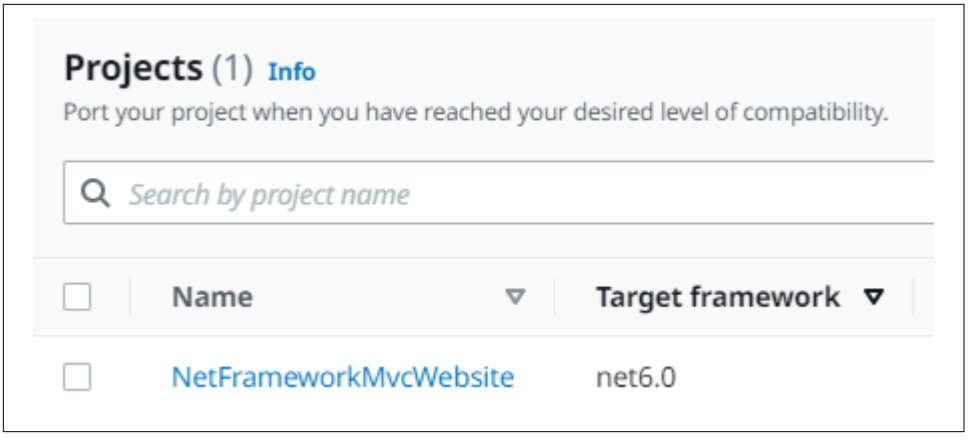

Figure 1-18. AWS Porting Assistant Complete

If you click the link to open in Visual Studio you will see the solution loads and your NuGet packages will have been upgraded to versions compatible with .NET Core/6+. Where the Porting Assistant's help ends however, is by refactoring all the unsupported code. When I try to build this I will still get errors that `System.Web.HttpContext` does not exist and "the helper directive is not supported" so it may be worth trying out all tools available and comparing your results. Overall the .NET Porting Assistant from AWS does provide a very quick and accessible UI for visualizing and assessing the effort involved in refactoring you .NET Framework code to work with modern .NET.

Conclusion

The strategies and tools we have covered in this chapter will help you move an existing application running on IIS to AWS. From the basic "Rehosting On AWS" on page 12 of an application without touching the original source, to "Rearchitecting: Moving to .NET (Core)" on page 22 and making inroads into modernization of your codebase. Whichever strategy you choose you will benefit from at least some of the advantages of running .NET in the AWS cloud, however, it is worth considering the next steps for your application. Some of the approaches we have covered here will leave your code in the exact same state as it was before you migrated. If you were not actively developing your codebase before the migration then you will not be in a much better position to do so after choosing one of these paths. For a codebase you intend to continue to develop, to iterate on and to add functionality to, then you should plan for a modernization of your legacy application. Modernization will involve not just moving to .NET 6+ (although that will be a prerequisite) it will also involve replacing third party dependencies, replacing external services, and refactoring the architecture of your application to use patterns that better exploit the cloud

environment you are now running in. All these will be covered in the next chapter in which I will sell you on the term "serverless" and how it can apply to you as a .NET developer.

CHAPTER 2

Modernizing .NET Applications to Serverless

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *mpotter@oreilly.com*.

The term "serverless" can be a source of confusion in software engineering circles. How can you run a web application without servers? Is it really just a marketing phrase? Well in a way, yes it is. Serverless computing is the broad term for backend systems that are architected to use managed services instead of manually configured servers. Serverless applications are "serverless" in the same way that a wireless charger for your smartphone is "wireless". A wireless charger still has a wire; it comes out of the charger and plugs into the mains outlet, transferring energy from your wall to your desk. The *wireless* part is only the interface between the charger and your phone. The word "serverless" in Serverless computing is used in the same way. Yes, there are servers, but they are all managed behind the scenes by AWS. The servers are the mains cable plugging your otherwise wireless charger into the wall outlet. The other side, the part visible to us as developers, the part that really matters when deploying

and running code, that part does not involve any configuration or management of servers and is therefore *serverless*.

You are most likely already using serverless computing for some tasks. Consider three popular types of server:

- Web Servers
- Email Servers
- File Servers

Of these three, the second two are servers that we have been replacing with managed serverless solutions for a long time now. If you have ever sent an email via an API call to Mailchimp, Mailgun, SendGrid, SparkPost, or indeed Amazon's Simple Email Service (SES) then you have used a serverless emailing solution. The days of running an SMTP server either in-house or in the cloud are, for a lot of organizations, already firmly in the past. File servers too are steadily going out of fashion and being replaced by serverless file storage solutions. Many modern web applications rely entirely on cloud native services such as Amazon Simple Storage Service (S3) as their primary storage mechanism for files. The last stone to fall from that list above is the web server. In this chapter we will show you how to replace your .NET web server with a serverless, cloud native implementation and what that means for the way you write your code.

A Serverless Web Server

A web server is a computer, either physical or virtual, that is permanently connected to the internet and is ready to respond to HTTP(S) requests 24 hours a day. The lifecycle of an HTTP request inside a web server involves a lot of steps and executes code written by many different parties in order to read, transform and execute each request. Right in the middle of that journey is the code written by you, the application developer.

In order to replicate the functionality of a web server, we need a way to run that custom logic on a managed and on-demand basis. AWS offers us a few ways to do this, we will look at a service that allows us to deploy a fully-managed containerized application without having to worry about servers. AWS Fargate is another serverless solution for running web servers on a pay-as-you-go pricing model. These services take your web services and deploy them to the cloud in a serverless way. We can, however, go one step further and break apart the service itself into individual functions that are deployed independently to the cloud. For this we will need functions as a service (FaaS).

FaaS solutions, such as AWS Lambda, are stateless compute containers that are triggered by events. You upload the code you want to run when an event occurs and the

cloud provider will handle provisioning, executing and then deprovisioning resources for you. This unlocks two main advantages:

1. FaaS allows your code to scale down to zero. You only pay for the resources you use *while your function is executing* and nothing in between. This means you can write applications that follow a true pay-as-you-go pricing model paying only for each function execution and not for the unused time in between.
2. FaaS forces you to think about separation of concerns, and places restrictions on the way you write your application that will generally lead to better code.

> Simplicity is hard work. But, there's a huge payoff. The person who has a genuinely simpler system - a system made out of genuinely simple parts, is going to be able to affect the greatest change with the least work. He's going to kick your ass. He's gonna spend more time simplifying things up front and in the long haul he's gonna wipe the plate with you because he'll have that ability to change things when you're struggling to push elephants around.
>
> —Rich Hickey, creator of the Clojure programming language

Simplicity as it relates to writing your code for FaaS means thinking in terms of **pure functions** and following the Single Responsibility Principle. A function should take a value object, perform an action, and then return a value object. A good example of a pure function that follows this pattern is a `Reverse()` function for a string.

```
public static string Reverse(string str)
{
    char[] charArray = str.ToCharArray();
    Array.Reverse( charArray );
    return new string( charArray );
}
```

This function fits the two requirements to be called a "pure function".

1. It will always return the same output for a given input.
2. It has no side effects, it does not mutate any static variables, mutable reference arguments, or otherwise require persistence of state outside the scope of the function.

These are both characteristics that are required for writing functions for FaaS. Since FaaS functions run in response to events, you want to keep point number 1 above true so that the sentence "when X happens perform Y" remains true for every time event X occurs. You also need to keep your function stateless (point 2 above). This is both a limitation and a feature of running your function on managed FaaS. Because AWS can and will deprovision resources in between executions of your function it is not possible to share static variables. If you want to persist data in an FaaS function

you must intentionally persist it to a shared storage area, for example by saving to a database or a Redis cache.

Architecting your code into stateless functions with no side effects like this can be a challenge that requires discipline. It forces you to think about how to write your logic and how much scope each area of your code is really allowed to control. It also forces you to think about separation of concerns.

When multiple FaaS functions are connected together using events and augmented with other serverless services (such as message queues and API gateways), it becomes possible to build a back end application that will be able to perform any function a traditional web API running on a server can. This allows you to write a completely serverless back end application hosted on AWS. And the best bit is, you can do all of this in C#! AWS offers many services you can use as the building blocks for such a serverless application, let's take a look at some of the components they have to offer.

Choosing Serverless Components for .NET on AWS

We are going to introduce you to some of the most useful serverless services from AWS and how you can use the various packages of the AWS SDK for .NET to interact with these services in your code. Let's progressively build a serverless web application for the rest of this chapter, adding functionality with each new concept introduced.

Imagine that we run a software development consultancy and are recruiting for the best and brightest C# developers around! In order for them to apply, we want them to visit our website, where they will find an HTML form that allows them to upload their resume as a PDF file. Submitting this form will send their PDF resume to our API, where it will be saved to a file server and an email will be sent to our recruitment team, asking them to review it. We'll call this the Serverless C# Resume Uploader. Figure 2-1 shows a high level overview of the architecture.

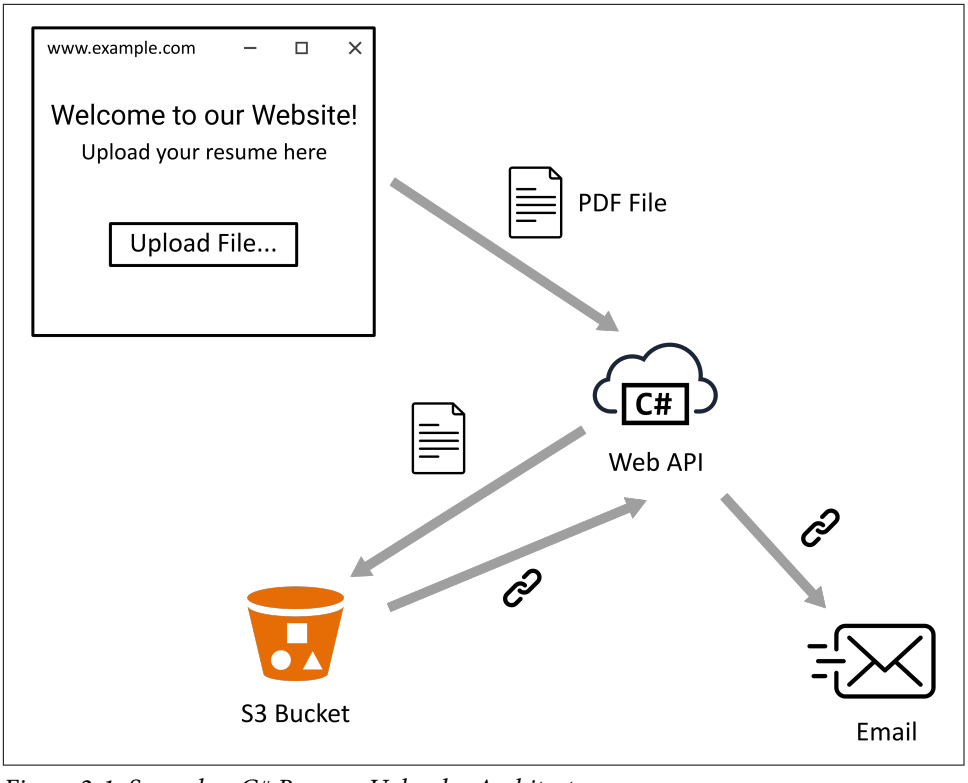

Figure 2-1. Serverless C# Resume Uploader Architecture

The current implementation of our backend uses a web API controller action to accept the PDF file upload, save it to cloud storage and email our recruitment team. The code looks like this:

```
[ApiController]
[Route("[controller]")]
public class ApplicationController : ControllerBase
{
    private readonly IStorageService _storageService;
    private readonly IEmailService _emailService;

    public ApplicationController(IStorageService storageService,
                                 IEmailService emailService)
    {
        _storageService = storageService;
        _emailService = emailService;
    }

    [HttpPost]
    public async Task<IActionResult> SaveUploadedResume()
    {
```

```
            Request.EnableBuffering();
            using var fileStream = new MemoryStream();
            using var reader = new StreamReader(fileStream);
            await Request.Body.CopyToAsync(fileStream); ❶

            var storedFileUrl = await _storageService.Upload(fileStream); ❷

            await _emailService.Send("recruitment@example.com",
                $"Somebody has uploaded a resume! Read it here: {storedFileUrl}"); ❸

            return Ok();
        }
    }
```

❶ Read the uploaded file from the request.

❷ Save it to cloud storage .

❸ Send an email to our recruitment team with a link to the file.

The code for `IStorageService` looks like this:

```
public class AwsS3StorageService : IStorageService
{
    const string BucketName = "csharp-examples-bucket";

    public async Task<string> Upload(Stream stream)
    {
        var fileName = Guid.NewGuid().ToString() + ".pdf"; ❶

        using var s3Client = new AmazonS3Client(RegionEndpoint.EUWest2);

        await s3Client.PutObjectAsync(new PutObjectRequest()
        {
            InputStream = stream,
            BucketName = BucketName,
            Key = fileName,
        }); ❷

        var url = s3Client.GetPreSignedURL(new GetPreSignedUrlRequest()
        {
            BucketName = BucketName,
            Key = fileName,
            Expires = DateTime.UtcNow.AddMinutes(10)
        }); ❸

        return url;
    }
}
```

❶ Create a unique S3 key name.

❷ Upload the file to S3.

❸ Generate a pre-signed URL pointing to our new file.[1]

This uses the `AWSSDK.S3` NuGet package for saving a file to an AWS S3 bucket. In this example the bucket is called "csharp-examples-bucket"[2] and the file will be uploaded and given a unique key using `Guid.NewGuid()`.

The second service in our example is the `IEmailService` which uses AWS Simple Email Service (SES) to send an email to our recruitment team. This implementation uses another NuGet package from the AWS SDK called `Amazon.SimpleEmail`.

```
public class AwsSesEmailService : IEmailService
{
    public async Task Send(string emailAddress, string body)
    {
        using var emailClient = new AmazonSimpleEmailServiceClient(
                                                RegionEndpoint.EUWest1);

        await emailClient.SendEmailAsync(new SendEmailRequest
        {
            Source = "from@example.com",
            Destination = new Destination
            {
                ToAddresses = new List<string> { emailAddress }
            },
            Message = new Message
            {
                Subject = new Content("Email Subject"),
                Body = new Body { Text = new Content(body) }
            }
        });
    }
}
```

We will be using these two implementations for the rest of the examples in this chapter. As mentioned earlier, by saving the PDF file with S3 and sending our email via SES we are already using serverless solutions for file storage and email services. So let's get rid of this web server too and move our controller action to managed cloud functions.

1 A pre-signed url is a link to an S3 resource that can be used from anywhere to download the contents anonymously, ie. by entering it into a browser window. The url is pre-signed using the authentication permissions of the IAM role that made this request. This effectively allows this you to share access to a secured S3 resource with anyone with whom this pre-signed url is shared.

2 S3 bucket names are unique across *all* AWS accounts in a region. So you may find the bucket name you want is unavailable as it is in use by another AWS account. You could avoid this by prefixing the bucket name with the name of your product or organization.

Developing with AWS Lambda and C#

AWS's FaaS product is called Lambda. AWS Lambda was introduced in 2014 by Werner Vogels, CTO of Amazon, with the following summary that we feel nicely sums up the motivation and value behind AWS Lambda.

> The focus here is on the events. Events may be driven by Web services that would trigger these events. You'll write some code, say, in JavaScript, and this will run without any hardware that you have to provision for it.
>
> —Werner Vogels

Since 2014, AWS Lambda has grown enormously in both adoption and features. They've added more and more event options to trigger your functions from, cutting edge abilities like Lambda@Edge that runs your functions on the CDN server closest to your users (aka "edge computing"), custom runtimes, shared memory layers for libraries called "Lambda layers", and crucially for us, built-in support for .NET on both x86_64 and ARM64 CPU architectures.

There are a lot of examples provided by AWS for writing Lambda functions in C# and for refactoring existing code so it can be deployed to AWS Lambda. Here is a quick sample to create and deploy a simple Lambda function in C# by taking advantage of the templates found in the `Amazon.Lambda.Templates` nuget package. This package includes project templates that can be used with the `dotnet new` command on the .NET Core CLI.

```
dotnet new -i Amazon.Lambda.Templates

dotnet new lambda.EmptyFunction --name SingleCSharpLambda
```

This will create a folder called "SingleCSharpLambda" containing a source and a test project for your function. The sample function is in a file called `Function.cs`.

```
[assembly: LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.
           DefaultLambdaJsonSerializer))]

namespace SingleCSharpLambda
{
    public class Function
    {
        public string FunctionHandler(string input, ILambdaContext context)
        {
            LambdaLogger.Log("Hello from" + context.FunctionName);

            return input?.ToUpper();
        }
    }
}
```

The template creates a class with a public function `FunctionHandler(input, context)`. This is the method signature for the entry point to every C# Lambda function in AWS Lambda. The two parameters are the input, the shape of which will be determined to whatever event we hook our Lambda function up to, and the `ILambdaContext context` which is generated by AWS on execution and contains information about the currently executing Lambda function.

We've also added a line to print a log message to the template function above so we can check it executes when we run it on AWS. The static class `LambdaLogger` here is part of the `Amazon.Lambda.Core` package which was added in with our template. You can also use `Console.WriteLine()` here, AWS will send any call that writes to `stdout` or `stderr` to the CloudWatch[3] log stream attached to your function.

Now we can get on and deploy our function to AWS. If you don't already have them installed, now is a good time to get the AWS Lambda Global Tool for .NET. A Global Tool is a special kind of NuGet package that you can execute from the `dotnet` command line.

```
dotnet tool install -g Amazon.Lambda.Tools
```

Then deploy your function.

```
dotnet lambda deploy-function SingleCSharpLambda
```

If not already set in the `aws-lambda-tools-defaults.json` configuration file, you will be asked for AWS region and the ARN for an IAM role you would like the function to use when it executes. The IAM role can be created on the fly by the `Amazon.Lambda.Tools` tool including all the permissions needed. You can view and edit the permissions for this (or any other) IAM role by visiting the IAM Dashboard.

Figure 2-2 shows our example function in the AWS Management Console for the region we deployed to. You can test the function directly in the console from the "Test" tab on the next screen.

3 CloudWatch is a service from AWS that provides logging and monitoring for your entire cloud infrastructure. AWS Lambda connects to CloudWatch and posts log messages that you can view from the CloudWatch console, or integrate into another AWS service further downstream.

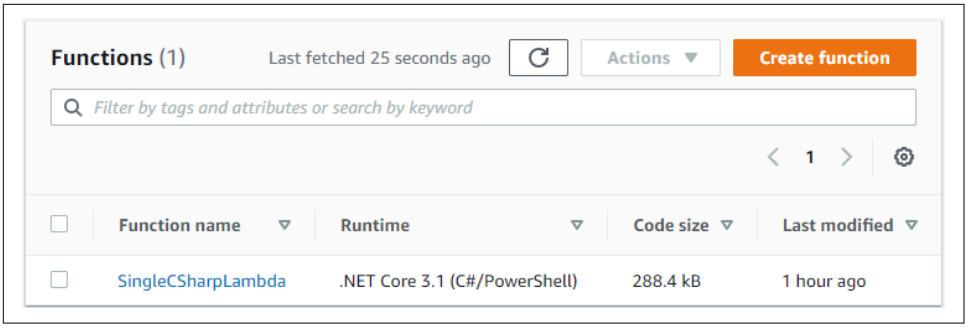

Figure 2-2. Function list in the AWS Lambda Console

This test window (shown in Figure 2-3) allows you to enter some JSON to pass into your function in the `input` parameter we defined earlier. Since we declared our input value as being of type **string** in `FunctionHandler(string input, ILambdaContext context)` we should change this to be any string and we can test the function directly in the management console.

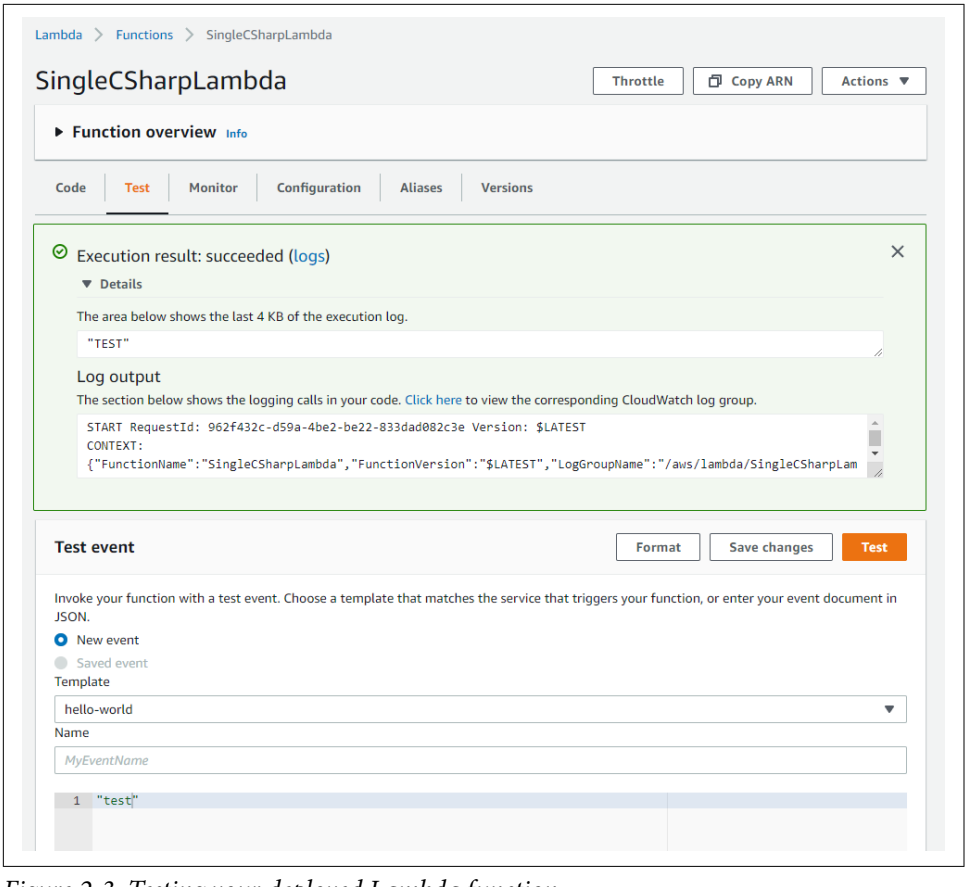

Figure 2-3. Testing your deployed Lambda function

The results of your execution are inserted into the page, including an inline "Log output" window which is also shown in Figure 2-3. This allows you to rapidly find and debug errors in your Lambda function and perform manual testing.

C# Resume Example: AWS Lambda

Now that we are familiar with AWS Lambda, let's apply this to our serverless example. We are going to take our web API controller from the C# Resume Uploader example and turn it into an AWS Lambda function. Any web API controller can be deployed to AWS Lambda with the help of a package from AWS called `Amazon.Lambda.AspNetCoreServer.Hosting`. This package uses makes it easy to wrap a .NET controller in a Lambda function called by API Gateway

API Gateway is a service from AWS that makes building an API composed of Lambda functions possible. It provides the plumbing between HTTP requests and Lambda functions allowing you to configure a set of individual Lambda functions to run on GET, PUT, POST, PATCH and DELETE requests to API routes you configure in API Gateway. We are using API Gateway in this example to map the route POST:https://our-api/Application to a Lambda function.

To deploy an ASP .NET application to AWS Lambda using this method, install the `Amazon.Lambda.AspNetCoreServer` package then add a call to `AddAWSLambdaHosting()` in the services collection of the application[4]. This allows API Gateway and AWS Lambda to act as the web server when running on Lambda.

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

// Add AWS Lambda support.
builder.Services.AddAWSLambdaHosting(LambdaEventSource.HttpApi); ❶

app.MapRazorPages();

var app = builder.Build();
```

❶ This is the only line we need to add.

Just as we did in "Developing with AWS Lambda and C#" on page 40, deploy this Lambda function to AWS using our new class as the entry point. We will have to create a JSON file for settings. If you have Visual Studio and the AWS Toolkit for Visual Studio installed, there are template projects you can try out that demonstrate how this JSON file is configured, so I'd urge you to give those a go. For our example here however, we are just going to deploy it straight to AWS Lambda.

```
dotnet lambda deploy-function UploadNewResume
```

Figure 2-4 shows the steps `dotnet lambda deploy-function` will take you through if you do not have all these settings configured in your JSON file. For the IAM role, we need to create a role with the policies in place to do all the things our resume uploader function will need to do (saving to S3, sending email via SES) along with invoking a Lambda function and creating CloudWatch log groups.

4 This method uses the Minimal APIs style of configuring and ASP .NET application that was introduced in .NET 6. Amazon.Lambda.AspNetCoreServer does also support .NET applications built on earlier versions of .NET Core however the configuration is slightly different and involved implementing Amazon.Lambda.AspNetCoreServer.APIGatewayProxyFunction. More information can be found at Amazon.Lambda.AspNetCoreServer

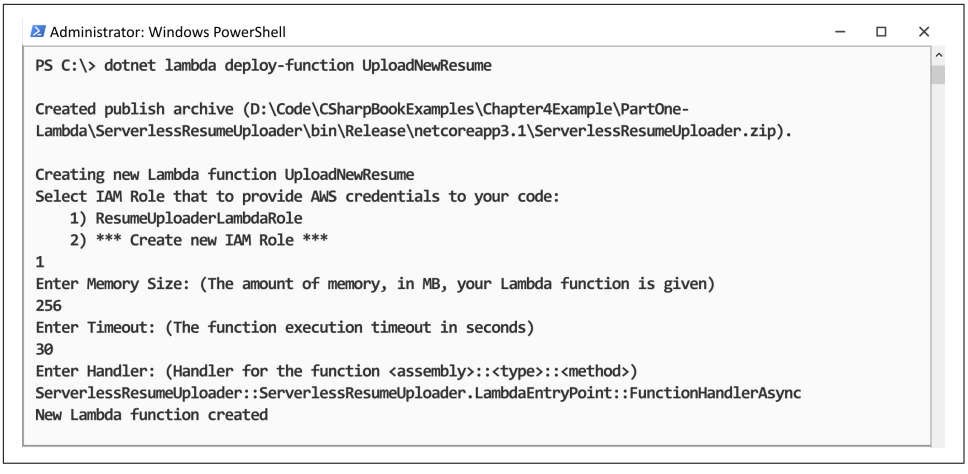

Figure 2-4. Execution of dotnet lambda deploy-function in the console

Next we have to create an API Gateway service to attach our Lambda to the outside world. Later on in this chapter we will explore the AWS Serverless Application Model (SAM), which is a great way of creating and managing serverless resources such as API Gateway, using template files that can be checked in to source control. For now, however, I would recommend creating the API Gateway service using the AWS Management Console as shown in Figure 2-5. The AWS Management Console is a great way to explore and familiarize yourself with all the different settings and options of these managed AWS services so when you do come to keep these settings in template files you have an anchor point in your mind to how everything fits together.

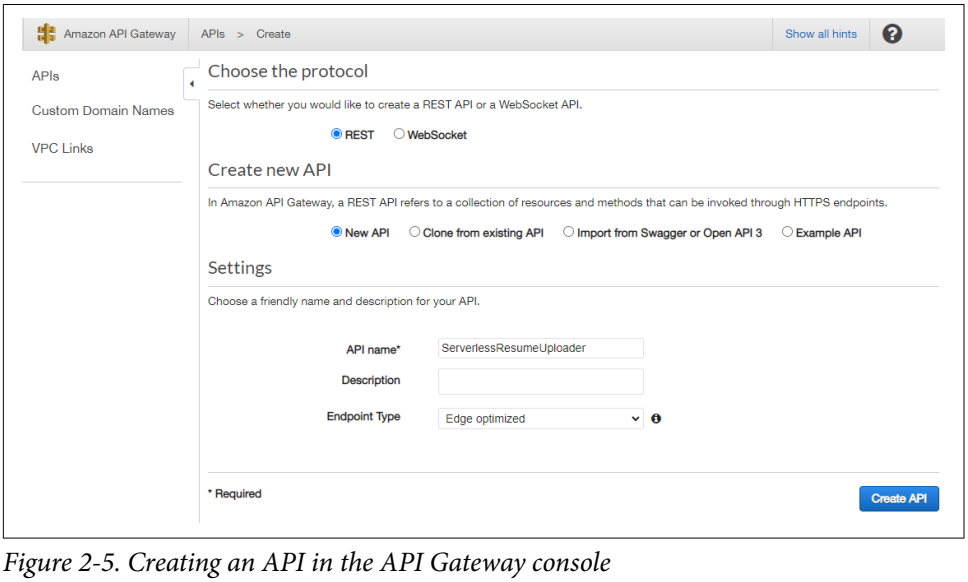

Figure 2-5. Creating an API in the API Gateway console

Figure 2-6 shows how we have set up an API Gateway service with a Lambda function proxy that forwards all routes to the AWS Lambda function we just deployed, `UploadNewResume`. You can test this out either by using the `TEST` tool in the console (shown on the left in Figure 2-6) or, better yet, by making an actual HTTP POST request with the PDF file in the body to the public URL of your API Gateway service. You can find the URL of API Gateway in the "Stages" setting in the management console. For example, my API Gateway instance has the id `xxxlrx74l3` and is in the `eu-west-2` region. To upload a resume to our API we can execute a POST request to this using `curl`.

```
curl https://xxxlrx74l3.execute-api.eu-west-2.amazonaws.com/Prod/Application
 --data-binary @MyResumeFile.pdf
```

This will upload our PDF file to API Gateway, which will pass it on to our Lambda function where our code will save it to S3 and send out an email.

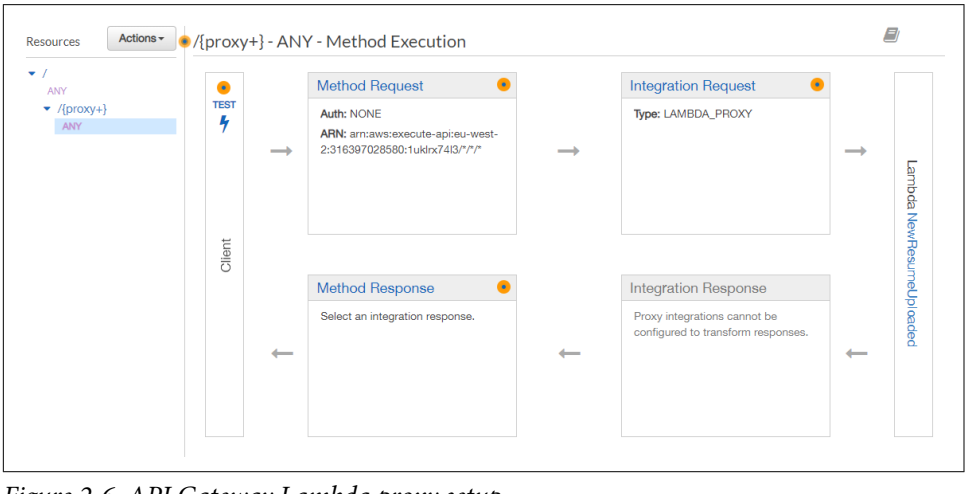

Figure 2-6. API Gateway Lambda proxy setup

All this has been possible by wrapping our API controller in the `APIGatewayProxyFunction` class and deploying to AWS Lambda as a cloud function, next we are going to explore what is possible when we add multiple Lambda functions to perform more complex and granular tasks in our applications.

Developing with AWS Step Functions

By this point we have deployed a web API controller to a Lambda function and hooked it up to API Gateway so it can be called via HTTP requests. Next let's think about what we can do to really take advantage of our new function as a service execution model. Remember how we spoke about FaaS forcing you to be more intentional about separation of concerns? The function in our example is doing quite a lot. This

is often the case with API controller actions in any language, not just C# - they do tend to assume multiple responsibilities at once. Our `UploadNewResume()` is doing a lot more than simply uploading a resume, it is emailing the recruitment team and it is creating a new file name. If we want to further unlock some of the flexibility of serverless and FaaS we need to split these operations out into their own functions. AWS has a very neat tool for managing workflows involving multiple Lambda functions (and more) and it is called AWS Step Functions.

AWS Step Functions abstracts away the plumbing between steps in your workflow. We can take a look at the controller action from our previous example and see how much of this code is just plumbing.

```
[HttpPost]
public async Task<IActionResult> SaveUploadedResume()
{
    Request.EnableBuffering();
    using var fileStream = new MemoryStream();
    using var reader = new StreamReader(fileStream);
    await Request.Body.CopyToAsync(fileStream); ❶

    var storedFileUrl = await _storageService.Upload(fileStream); ❷

    await _emailService.Send("recruitment@example.com",
        $"Somebody has uploaded a resume! Read it here: {storedFileUrl}"); ❸

    return Ok(); ❹
}
```

❶ Moving bits around between memory streams.

❷ Executing a function on the storage service.

❸ Executing a function on the email service.

❹ Creating an HTTP response with code 200.

This function is doing four things, none of which are particularly complex, however *all* of which make up the business logic (or "workflow") of our back end API. It also does not really respect the single responsibility principle. The function is called `SaveUploadedResume()` so you could argue that its single responsibility should be inserting the resume file into S3. Why then, is it sending emails and constructing HTTP responses?

If you were to write unit tests for this function it would be nice to simply cover all the cases you would expect to encounter when "saving an uploaded resume". Instead, you have to consider mocking out email sending and the boilerplate needed to execute an API controller action in a unit test. This increases the scope of the function and

increases the number of things you need to test, ultimately increasing the friction involved in making changes to this piece of code.

Wouldn't it be nice if we could reduce the scope of this function down to truly having one responsibility (saving the resume) and split out email and HTTP concerns into something else?

With AWS Step Functions you can move some or all of that workflow out of your code and into a configuration file. Step Functions uses a bespoke JSON object format called Amazon States Language to build up what it calls the *State Machine*. These state machines in AWS Step Functions are triggered by an event and flow through the steps configured in the definition file, executing tasks and passing data along the workflow until an end state is reached.

We are going to refactor our Serverless C# Resume Uploader to use Step Functions in the next section but suffice to say, one of the major advantages of using step functions is the ability to develop the *plumbing* of your application separately to the individual functions. Figure 2-7 is a screenshot from the AWS Step Functions sections in the AWS management console with the section "Execution event history" showing the operations this execution of the state machine took to arrive at the end state.

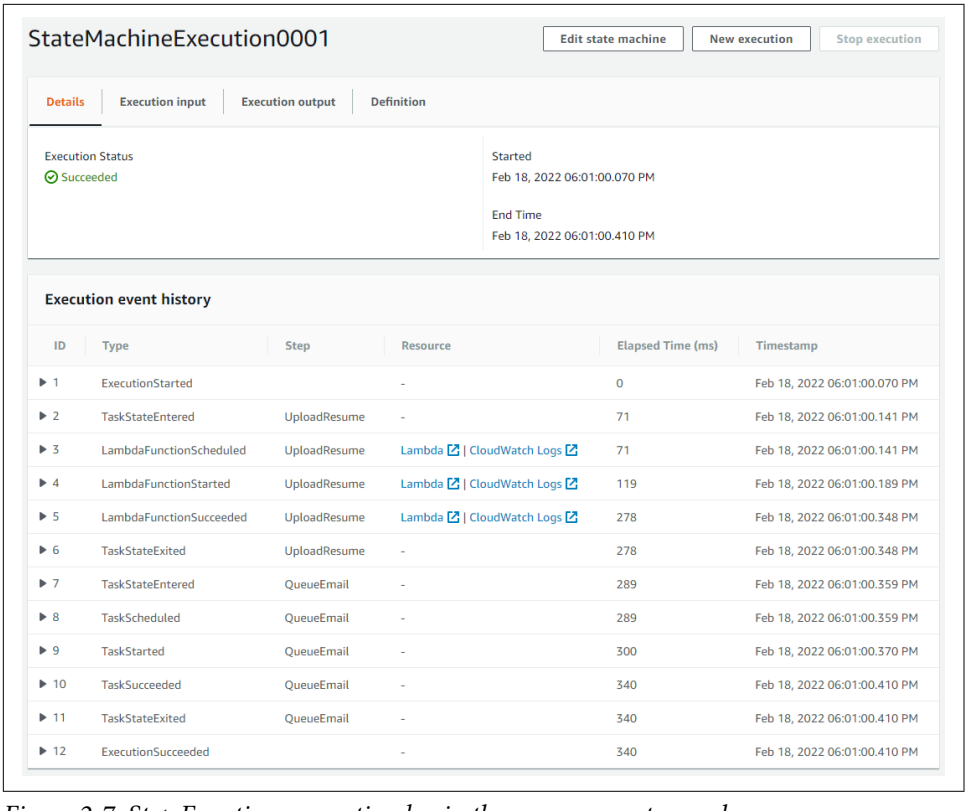

Figure 2-7. Step Functions execution log in the management console

C# Resume Uploader Example: Step Functions

To demonstrate what we can do with AWS Step Functions, let's take the AWS Lambda function we created in the previous part of our Resume Uploader example and split it out from one, to multiple Lambda functions. The `UploadNewResume` Lambda function can have the single responsibility of uploading the file to S3, then we have a second Lambda to send the email (`EmailRecruitment`). The HTTP part can also be abstracted away entirely and handled by the API Gateway. This gives us much more flexibility to change or optimize our workflow on the back end even after these two functions have been developed, tested and deployed

```
[assembly: LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.
           DefaultLambdaJsonSerializer))]

namespace ServerlessResumeUploader
{
    public class LambdaFunctions
    {
        private readonly IStorageService _storageService = new AwsS3StorageService();
```

```
        private readonly IEmailService _emailService = new AwsSesEmailService();

        ❶
        public async Task<StepFunctionsState> UploadNewResume(StepFunctionsState state,
                                                              ILambdaContext context)
        {
            byte[] bytes = Convert.FromBase64String(state.FileBase64);
            using var memoryStream = new MemoryStream(bytes);

            state.StoredFileUrl = await _storageService.Upload(memoryStream);

            state.FileBase64 = null;

            return state;
        } ❷
        public async Task<StepFunctionsState> EmailRecruitment(StepFunctionsState state,
                                                               ILambdaContext context)
        {
            await _emailService.Send("recruitment@example.com",
            $"Somebody uploaded a resume! Read it here: {state.StoredFileUrl}\n\n" +
            $"...and check out their Github profile: {state.GithubProfileUrl}");

            return state;
        }
    }
}
```

❶ The first Lambda function that takes a file and saves to S3.

❷ The second Lambda that emails our recruitment team a link to the file.

You can see the code for this is very similar to the `ApplicationController` code we had earlier, in this chapter except it has been refactored into two methods on a `LambdaFunctions` class. This will be deployed as two individual AWS Lambda functions that use the same binaries, a fairly common way to deploy multiple Lambda functions that share code. The actual uploading and email sending is again performed by the `AwsS3StorageService` and `AwsSesEmailService`, however we are not using dependency injection any more[5]. Also note that for these two functions the first parameter is now `state` and they are both of the type `StepFunctionsState`. AWS Step Functions executes your workflow by passing a state object between each task (in this case a task is a function). The functions add to or remove properties from the state. The state for our functions above looks like this:

5 It is possible to do dependency injection with AWS Lambda functions in C#, either by hand coding the setup or using third party libraries. You may often find, however, that the functions are so simple they really do not need to be decorated with it.

```
public class StepFunctionsState
{
    public string FileBase64 { get; set; }

    public string StoredFileUrl { get; set; }
}
```

When this `state` input is passed to the first function, `UploadNewResume()`, it will have the `state.FileBase64` set by API Gateway from the HTTP POST request from our front end. The function will save this file to S3 then set `state.StoredFileUrl` before passing the state object to the next function. It will also clear `state.FileBase64` from the state object. The state object in AWS Step Functions is passed around between each step and since this base 64 string will be quite large we can set it to null after reading it to reduce the size of the state object that is passed on.

We can deploy these two functions to AWS Lambda as we did before, however this time we need to specify the `--function-handler` parameter for each. The *function handler* is the C# function in our code that acts as the entry point to our Lambda, as we saw previously in "Developing with AWS Lambda and C#" on page 40.

```
dotnet lambda deploy-function UploadNewResume
    --function-handler ServerlessResumeUploader::ServerlessResumeUploader.
    LambdaFunctions::UploadNewResume


dotnet lambda deploy-function EmailRecruitment
    --function-handler ServerlessResumeUploader::ServerlessResumeUploader.
    LambdaFunctions::EmailRecruitment
```

Now if we go into AWS Step Functions we can create a new state machine, connect these two Lambdas together and to API Gateway. The AWS Management Console does give us a graphical user interface to build up these workflows, however, we are engineers so we are going to write it in Amazon States Language in a JSON file. This also has the crucial benefit of utilizing a plaintext JSON file that we can check into source control, giving us a great *Infrastructure as Code (IaC)* deployment model. The JSON configuration for our simple workflow here looks like this:

```
{
  "Comment": "Resume Uploader State Machine",
  "StartAt": "SaveUploadedResume",
  "States": {
    "SaveUploadedResume": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:eu-west-2:00000000:function:UploadNewResume",
      "Next": "EmailRecruitment"
    },
    "EmailRecruitment": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:eu-west-2:00000000:function:EmailRecruitment",
      "End": true
```

```
        }
      }
    }
```

The ARN (Amazon Resource Name) of the Lambda function can be found in the AWS Lambda section of the Management Console and it allows us to reference these Lambda functions from anywhere in AWS. If we copy the above JSON into the definition of our AWS Step Functions state machine you can see in the console that it creates a graphical representation of the workflow for us. Figure 2-11 later in this chapter shows what that looks like.

The last step in this example is to connect the start of our state machine to API Gateway. Whereas previously API Gateway was configured to proxy all requests to our AWS Lambda function, we now want to specify the exact route in API Gateway and forward it to our step functions workflow. Figure 2-8 shows the setup of a POST endpoint called `/applications` that has "Step Functions" set as the AWS Service. You can also see in Figure 2-8 that we have set Content Handling to "Convert to text (if needed)". This is an option specific for our resume uploader example here. Since we will be POSTing the PDF file to our API as raw binary, this option tells API Gateway to convert that to Base64 text for us. This can then be easily added onto our state object (represented by `StepFunctionsState.cs` shown earlier) and passed around between Lambda functions on our AWS Step Functions workflow.

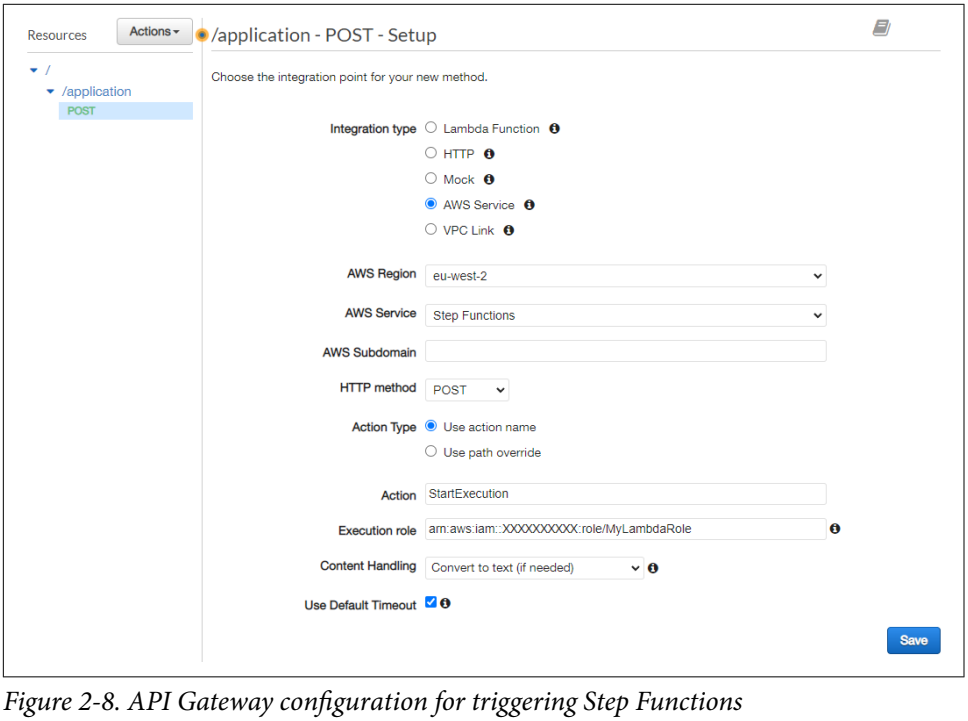

Figure 2-8. API Gateway configuration for triggering Step Functions

Lastly, we can set up some request mapping in API Gateway to transform the request into a JSON object that tells AWS Step Functions which state machine to execute. That is shown in Figure 2-9 as a mapping template linked to the `application/pdf` content type header.

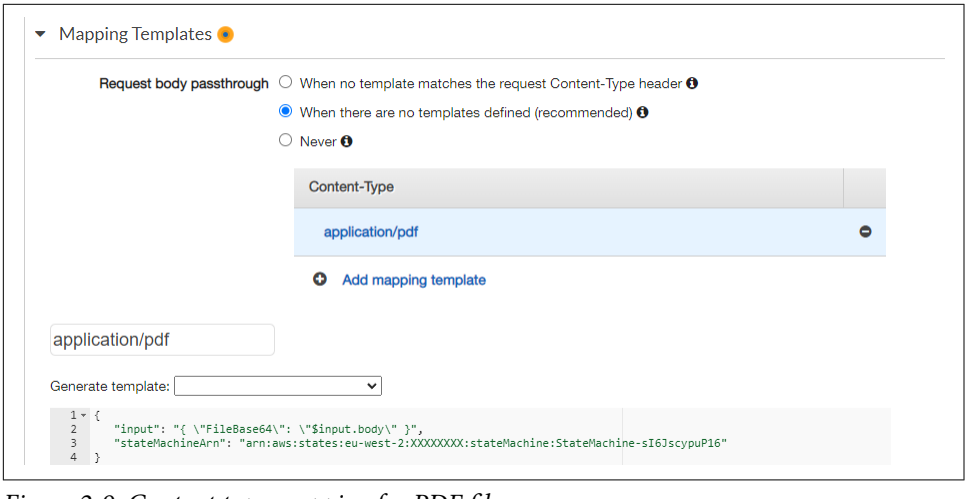

Figure 2-9. Content type mapping for PDF file

We now have everything in place to send a file to our API and watch the step functions workflow execute.

```
curl https://xxxlrx74l3.execute-api.eu-west-2.amazonaws.com/Prod/Application
 --data-binary @MyResumeFile.pdf
```

If we navigate to the *Executions* tab of the AWS Step Functions state machine in the management console we will be able to see this upload trigger an execution of our state machine. Figure 2-10 shows a visual representation of the execution as it goes through our two steps, you can see this graph by clicking on the latest execution.

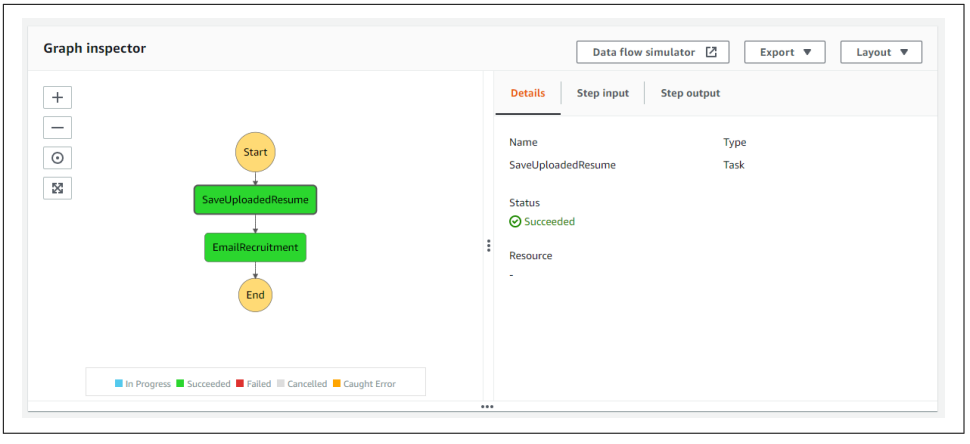

Figure 2-10. Successful execution of our state machine

We will admit, this has seemed like a lot of work just to save a file to S3 and send an email. By this point you might be thinking what is even the point? So we can abstract our workflow into a JSON file? The point of refactoring all of this and deploying to AWS Step Functions is that now we have the starting point for something much more flexible and extensible. We can do some extremely cool things now without having to go anywhere near our two deployed Lambda functions. For example, why not take advantage of AWS Textract to read the PDF file and extract the candidate's GitHub profile? This is what we'll be doing in the next example.

C# Resume Uploader Example: AWS Textract

This is the kind of functionality that not too long ago would have been a very involved task to code, but with step functions and AWS Textract we can add this in very easily without even having to pull extra third party libraries into our existing code, or even recompile it.

```
public async Task<StepFunctionsState> LookForGithubProfile(StepFunctionsState state,
ILambdaContext context)
{
    using var textractClient = new AmazonTextractClient(RegionEndpoint.EUWest2);❶

    var s3ObjectKey = Regex.Match(state.StoredFileUrl,
                "amazonaws\\.com\\/(.+?(?=\\.pdf))").Groups[1].Value + ".pdf";

    var detectResponse = await textractClient.DetectDocumentTextAsync(
        new DetectDocumentTextRequest
    {
        Document = new Document
        {
            S3Object = new S3Object
            {
```

```
                Bucket = AwsS3StorageService.BucketName,
                Name = s3ObjectKey,
            }
        }
    });

    state.GithubProfileUrl = detectResponse.Blocks
            .FirstOrDefault(x => x.BlockType == BlockType.WORD &&
                                 x.Text.Contains("github.com"))
            ?.Text;

    return state;
}
```

❶ The AmazonTextractClient is found in the `AWSSDK.Textract` nuget package and allows us to easily call AWS Textract.

Textract is one of many machine learning services offered by AWS. They also provide AWS Comprehend for performing natural language processing on text and AWS Cognition for tagging images. All of these services are priced on an accessible pay-as-you-go model and can be called from a Lambda function like we are doing here with AWS Textract.

Here we have the code for another C# Lambda function that will take in the same state object we have been using for all our functions and send the PDF file to Textract. The response from this will be an array of text blocks that were extracted from the PDF file. If any of these text blocks contains the string "github.com" we add it into our state object for use by a later Lambda function. This allows us to include it in the email that is sent out, for example.

```
public async Task<StepFunctionsState> EmailRecruitment(StepFunctionsState state,
                                                       ILambdaContext context)
{
    await _emailService.Send("recruitment@example.com",
    $"Somebody uploaded a resume! Read it here: {state.StoredFileUrl}\n\n" + ❶
    $"...check out their Github profile: {state.GithubProfileUrl}"); ❷

    return state;
}
```

❶ The stored file URL has still been left in the state object from the first function that saved it to S3

❷ This new field was added by `LookForGithubProfile()` when Textract found a GitHub URL in the PDF

We can deploy our new `LookForGithubProfile()` function as a third AWS Lambda and add it into our state machine JSON. We have also added an error handler in here that calls off to a function to notify us that there was an error uploading the resume. There are many different steps and ways to create complex paths in your workflow using Amazon States Language JSON definitions. Figure 2-11 shows all of this together in the AWS Management Console with our definition JSON on the left and a visual representation of our workflow on the right.

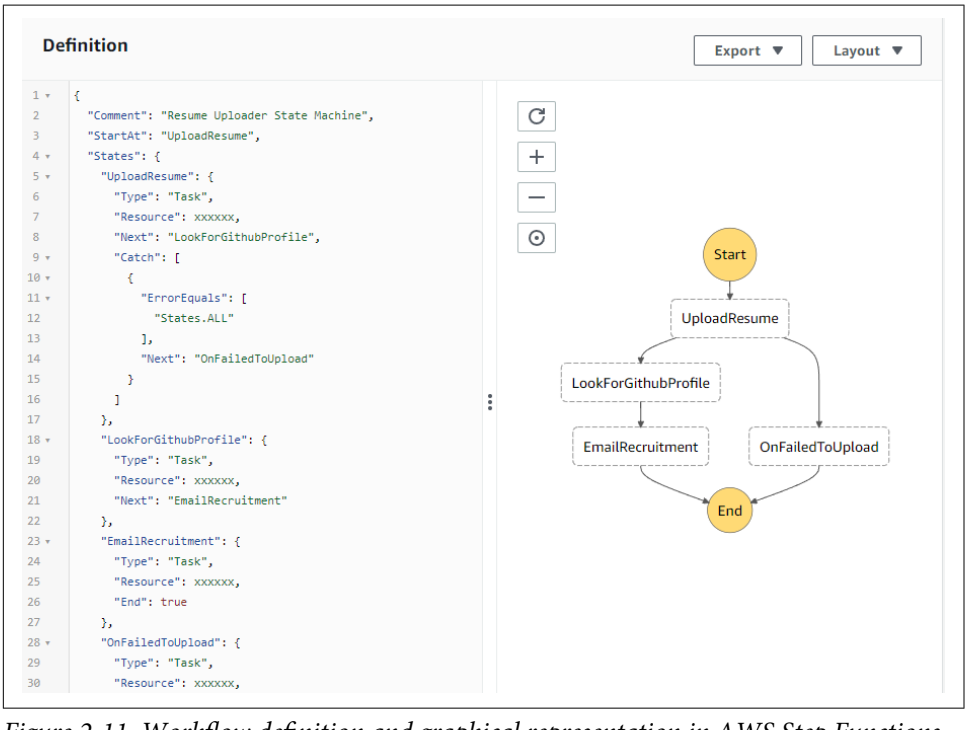

Figure 2-11. Workflow definition and graphical representation in AWS Step Functions

We can also update our architecture diagram to remove the web server shown in Figure 2-1 and make the application completely serverless as illustrated in Figure 2-12

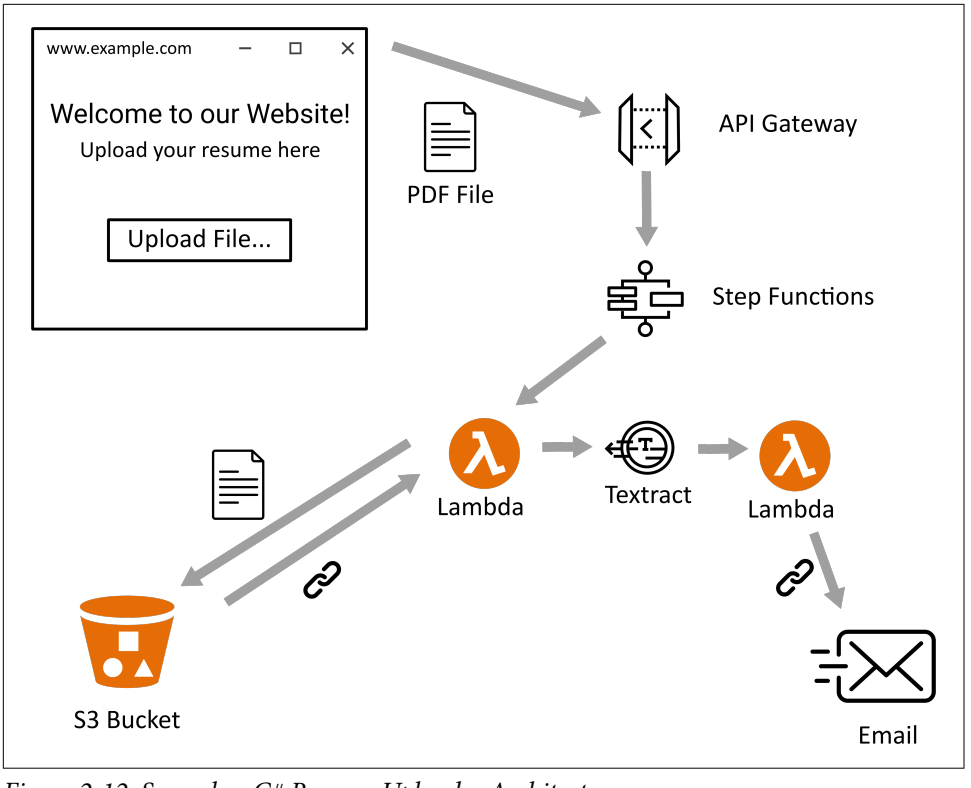

Figure 2-12. Serverless C# Resume Uploader Architecture

Now we have a truly serverless application in which the individual components can be developed, tested and deployed independently. But what if we don't want to rely on AWS Step Functions to bind all this logic together?

Developing with SQS and SNS

AWS Step Functions are not the only way to communicate between AWS Lambda invocations and other serverless services. Long before Step Functions state engines existed we had message queues and the publisher/subscriber pattern. These are the two services offered by AWS, which are spelled out in detail below.

Amazon Simple Notification Service (SNS)
: SNS is a distributed implementation of the pubsub pattern. Subscribers attach themselves to an SNS channel (called a "topic") and when a message is published all subscribers will instantly be notified. You can have multiple subscribers listening to published messages on any given topic in SNS and it supports several endpoints such as email and SMS, as well as triggering Lambda functions or making HTTP requests.

Amazon Simple Queue Service (SQS)
: SQS is AWS's message queue. Instead of being pushed to subscribers, messages sent to SQS are added to the queue and stored there for a duration of time (up to 14 days). Message receivers poll the queue at a rate suitable for them, reading and then deleting the messages from the queue as necessary. SQS queues make it possible to delay actions or batch up messages until the subscriber is ready to handle them.

These two services allow us to think in terms of *messages* and *events*, which are important concepts for building serverless systems. They will allow us to implement a publisher/subscriber pattern in our example system.

C# Resume Uploader Example: SQS

Think about what we can do with our Serverless C# Resume Uploader application to take advantage of SQS or SNS. We know users can upload their resume to our API (via API Gateway) which will store it in S3 and then use Textract to read the candidate's GitHub profile. How about instead of immediately then emailing our recruitment team we add a message to a queue? That way we can run a job once every morning and send *one* email for *all* the messages that have built up on the queue during the past 24 hours. This might make it easier for our recruitment team to sit down and read all of the day's resumes at once instead of doing it piecemeal throughout the day. Can we do all of this without having to touch the rest of our code? Without having to rebuild, re-test or redeploy the entire application? Yes, and here's how.

Since we are using AWS Step Functions we can simply create a task in our definition JSON file that posts a message to an SQS queue. We don't need another Lambda function to do this, instead we can specify the message body directly in the JSON and then update our state engine definition. Here we have the task called `"QueueEmail"` that posts a message to an SQS queue we have set up called `UploadedResumeFiles`, referenced here by its ARN.

```
{
  "Comment": "Resume Uploader State Machine",
  "StartAt": "SaveUploadedResume",
  "States": {
    "SaveUploadedResume": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:eu-west-2:00000000:function:UploadNewResume",
      "Next": "QueueEmail"
    },
    "QueueEmail": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sqs:sendMessage",
      "Parameters": {
        "QueueUrl": "https://sqs.eu-west-2.amazonaws.com/00000000/UploadedResumeFiles",
        "MessageBody": {
```

```
            "StoredFileUrl.$": "$.StoredFileUrl"
          }
        },
        "End": true
      }
    }
}
```

Now when we trigger the state machine from a file being uploaded the `EmailRecruitment` Lambda will not be executed, we will just get a message posted to the queue. All we need to do now is write a new AWS Lambda function to run once a day that will read all the messages from the queue and send an email containing all the file URLs. The code for such a Lambda might look like this:

```
public async Task<string> BatchEmailRecruitment(object input, ILambdaContext context)
{
    using var sqsClient = new AmazonSQSClient(RegionEndpoint.EUWest2);
    var messageResponse = await sqsClient.ReceiveMessageAsync(
        new ReceiveMessageRequest() ❶
        {
            QueueUrl = queueUrl,
            MaxNumberOfMessages = 10
        });

    var stateObjects = messageResponse.Messages.Select(msg => Deserialize(msg.Body));

    var listOfFiles = string.Join("\n\n", stateObjects.Select(x => x.StoredFileUrl)); ❷

    await _emailService.Send("recruitment@example.com", ❸
        $"You have {messageResponse.Messages.Count} new resumes to review!\n\n"
         + listOfFiles);

    await sqsClient.DeleteMessageBatchAsync(new DeleteMessageBatchRequest() ❹
    {
        QueueUrl = queueUrl,
        Entries = messageResponse.Messages.Select(x => new DeleteMessageBatchRequestEntry()
        {
            Id = x.MessageId,
            ReceiptHandle = x.ReceiptHandle
        }).ToList()
    });

    return "ok";
}
```

❶ Connect to the SQS queue and read a batch of messages.

❷ Concatenate the file URLs from the state objects we posted inside each message.

❸ Send one email with all links.

❹ Delete the messages from the queue - this does not happen automatically.

Triggering this Lambda once per day can be done using AWS EventBridge as shown in Figure 2-13. This *Add Trigger* form is accessed from the *Function Overview* window of our new Lambda function in the AWS Management Console.

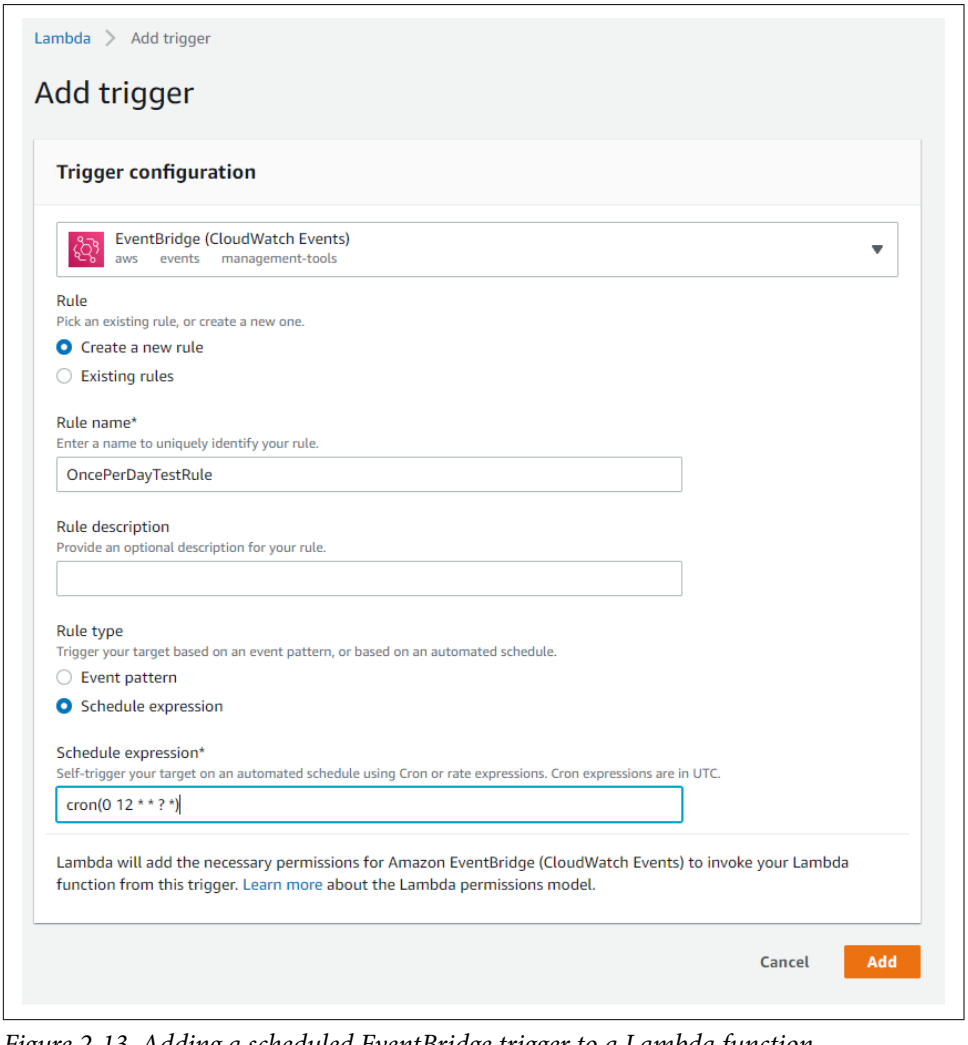

Figure 2-13. Adding a scheduled EventBridge trigger to a Lambda function

Now our system will send out an email once per day, including links to all the resumes on the queue, just as in Figure 2-14.

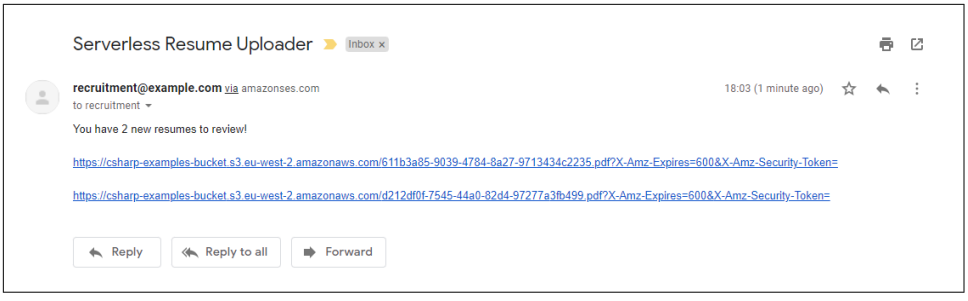

Figure 2-14. Email sent from our scheduled Lambda function

By adding an SQS queue we have modified the behavior of our system at runtime and improved the service it provides by collating all the links into one daily email. All of this however is still kicked off by an HTTP request to our API. Next we are going to explore some other types of events that can trigger logic and functions to run in our state engine.

Developing Event-Driven Systems with AWS Triggers

Imagine you speak to someone who has a brilliant idea for an app and wants you to build it for them[6]. You ask them to describe the functionality and they say something like this

"When the user uploads a photo I want to notify their followers, and when they send an email to my support mailbox I want to open a support ticket"

These "when X then do Y" statements are describing an *event-driven system*. All systems are event-driven under the hood, the trouble is that for a lot of back end applications those events are just HTTP requests. Instead of, "When the user uploads a photo I want to notify their followers". What we usually end up implementing is, "When our server receives an HTTP message indicating a user has uploaded a photo, notify their followers".

That's understandable, up until serverless computing came along the only way you could really implement this was to listen to an HTTP event on your API. But wouldn't it be great if we could remove that extra step and implement our architecture to respond to the *real events as they happen* and not some intermediary implementation detail, such as an HTTP request?

This is the central tenet behind event-driven serverless systems. You execute functions and trigger workflows on the changes that really make sense to the problems you are trying to solve. Instead of writing an application that listens to an action from

6 One of the drawbacks of being a software developer is this tends to happen a lot, whether you ask for it or not.

a human and then sends an HTTP request to an API to trigger an effect, you let AWS do all that and simply attach the event to the action directly. In the example above you could listen to a `s3:ObjectCreated:Post` in S3 and run your code whenever that event occurs, bypassing the API step completely.

C# Resume Uploader Example: Event-Driven

Let's visit our serverless resume uploader for one final time. If you look back to Figure 2-12 you can see the PDF file goes from our website to the Step Functions tasks via API Gateway. There is no reason for API Gateway to be there, it is simply an implementation detail to allow our front end to make an easy HTTP call when it uploads the file. Wouldn't it be nice if we could rid ourselves of this API altogether and have the website upload the file directly to an S3 bucket? This would also allow us to throw away our `UploadNewResume` Lambda function too, and there is no more cathartic feeling in software development than deleting code we no longer need while retaining all the functionality of our system.

By removing the API step and having the front end upload the file directly into S3 we also open up new possibilities for our system. For example, what if in addition to uploading a resume on our *website*, we want to accept resumes emailed to `application@ouremail.com`? Using SES it is relatively simple to hook up an email forwarder that will extract an attached PDF file and save it to S3. This would then trigger the same workflow on the back end, because we have hooked our logic up to the S3 event and not some intermediary HTTP API call. The statement "when we see a new PDF file in S3, kick off our workflow" becomes much more natural to the business problem we are solving. It doesn't actually matter any more how the PDF file gets in there our system reacts in the same way.

As far as uploading the PDF file directly to S3 goes we have options. If we already happen to be using AWS Amplify[7] on the front end then we can use the Storage module and the `"protected"` upload method, restricting access to a path in our S3 bucket based on the Cognito Identity of the authenticated user:

```
amplify add storage
```

The frontend JavaScript used to upload a file to S3 would look like this:

```
import Amplify, { Auth, Storage } from 'aws-amplify';

Amplify.configure({ Storage: { AWSS3: {
            bucket: '<bucket-name>'
        } } });
```

7 AWS Amplify is a front end framework for mobile and web apps that allows us to quickly build up a UI around serverless AWS services, such as S3 Simple Storage Service.

```
async function uploadFile(fileName, file) {
    await Storage.put(fileName, file, { level: 'protected',
            contentType: file.type });
}
```

Even without AWS Amplify and an authenticated user on the front end, there is still a way we can upload directly to the S3 bucket using a Pre-Signed URL. The process for this is to create a Lambda function behind an API gateway that, when executed, will call `AmazonS3Client.GetPreSignedURL()` and return that to the front end to use to upload the file. Sure, you still have an API in this scenario, but the *function* of this API is much more in line with performing one generic task. You could, after all, use the presigned URL to upload other types of file in the front end and hook multiple step function workflows to each.

Once you have the front end uploading files directly into S3 instead of sending them via a Web API, adding a trigger into the S3 bucket can be done directly in the management console on the *Properties* tab of the S3 bucket configuration, as shown in Figure 2-15.

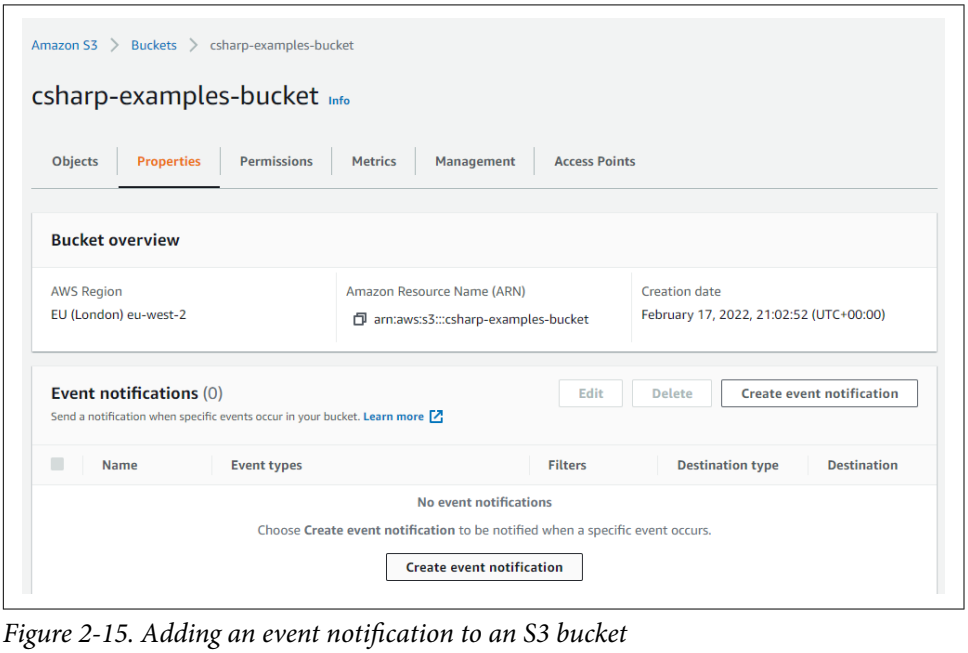

Figure 2-15. Adding an event notification to an S3 bucket

Add the step functions state engine as the destination for this event and we arrive at the final form of our event-driven, serverless, C# Resume Uploader Service! The final architecture is shown in Figure 2-16.

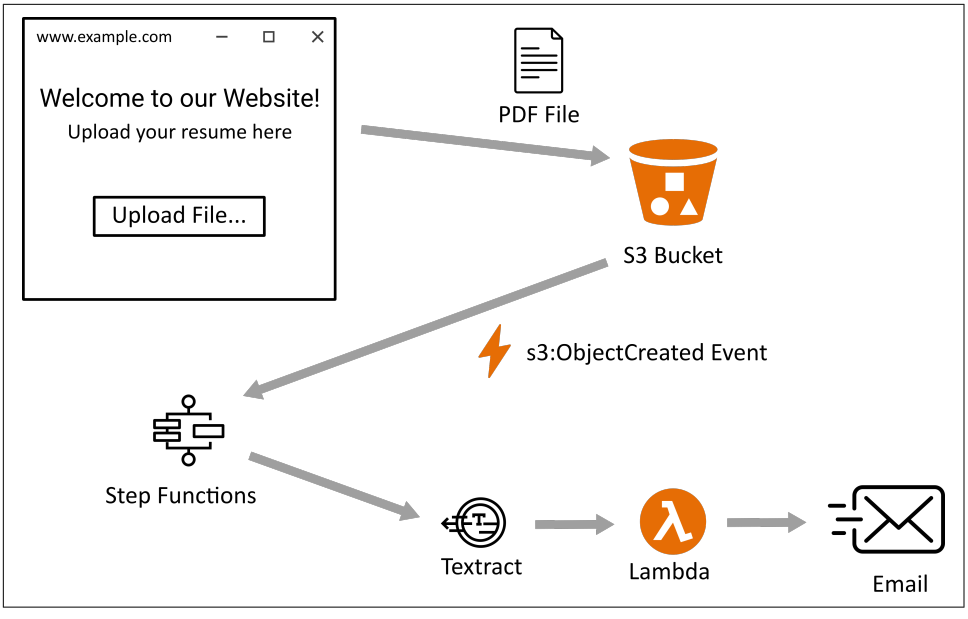

Figure 2-16. Final event-driven architecture of our resume uploader example

As you can see we now have a concise, descriptive and *event-driven* system that is dynamic enough to allow us to extend or modify parts of the workflow without risking introducing bugs or issues elsewhere. The last thing to think about is how we manage all these moving parts and easily configure them in one place.

Serverless Application Model (SAM)

So far we have been making all our configuration changes by logging into the AWS Management Console and clicking around the UI. This is fine for experimentation, but is not a method particularly well suited to running a production system. One wrong click and you could cause an outage to part of your application. This is where Infrastructure as Code (IaC) can help.

Infrastructure as code is the process of configuring your serverless infrastructure through machine readable definition files. CloudFormation is an IaC tool used across AWS that allows you to keep your entire cloud configuration in either YAML or JSON files. Virtually everything in your AWS can be modeled in CloudFormation templates, from DNS settings to S3 bucket properties to IAM roles and permissions. When a setting needs changing, you change the value in a CloudFormation template and tell AWS to apply the change to your resources. The most obvious advantage of this is that you can check your template JSON/YAML file into version control and have each change code reviewed, audited, and tested in a sandbox/staging environment, just like with any other code change.

One drawback of CloudFormation, however, is it can be quite complicated and verbose. Due to the sheer number of settings that are available for you to modify in your AWS resources, CloudFormation templates can become unwieldy when trying to configure a serverless system composed of multiple Lambda functions, message queues, and IAM roles. There are various tools that add an abstraction layer around CloudFormation and aid in configuring serverless systems. Tools such as Serverless Framework and Serverless Application Model (SAM) have been created to solve this problem.

You can think of SAM as a layer over the top of CloudFormation that brings the most pertinent settings for serverless applications to the front. You can find the full specification for SAM online at AWS SAM Documentation, but to give you an overview, here is part of the SAM YAML file for our C# Resume Uploader system.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: Resume Uploader Serverless C# Application.
Resources:
  SaveUploadedResumeLambda:
    Type: AWS::Lambda::Function
    Properties:
      Handler: ServerlessResumeUploader::ServerlessResumeUploader.
      LambdaFunctions::SaveUploadedResume
      Role: arn:aws:iam::0000000000:role/ResumeUploaderLambdaRole
      Runtime: dotnetcore3.1
      MemorySize: 256
      Timeout: 30
  LookForGithubProfileLambda:
    Type: AWS::Lambda::Function
    Properties:
      Handler: ServerlessResumeUploader::ServerlessResumeUploader.
      LambdaFunctions::LookForGithubProfile
      Role: arn:aws:iam::0000000000:role/ResumeUploaderLambdaRole
      Runtime: dotnetcore3.1
      MemorySize: 256
      Timeout: 30
```

You can see how we have defined the `SaveUploadedResumeLambda` and `LookForGithubProfileLambda` Lambda functions, indicated where in our C# code the entry point is, and configured them with memory, timeout, and permissions settings for execution.

With your infrastructure configured in SAM files like this you can easily deploy new environments for testing or staging. You benefit from code reviews and you get the ability to create an automated deployment pipeline for your *resources* just like with your application code.

Conclusion

Serverless computing allows you to build intricate, yet flexible and scalable solutions that operate on a pay-as-you-go pricing model and can scale down to zero. Personally, we use a serverless execution model whenever we need to build something quickly to validate an idea, solve a business problem, or where budget is a concern. Because you only pay for what you use, a serverless architecture centered on AWS Lambda can be an extremely low cost way to build a Minimum Viable Product (MVP) or deploy the back end for a mobile app. AWS offers a whopping 1 million Lambda executions per month for free, which can easily be enough to get a startup out of the idea phase or beta test a product. The other services we have introduced in this chapter all have extremely generous free tiers allowing you to experiment with ideas and architectures without breaking the bank.

That being said, going serverless will not automatically cause your system to be cheaper to run. You will need to be considerate of designing applications that make unnecessarily large numbers of AWS Lambda calls. Just because two functions *can* be two separate Lambdas doesn't always mean they *should* be from a cost and performance point of view. Experiment and measure. Functions as a service can also be expensive at higher volumes if your application is not architected to make efficient use of each invocation. Because you pay per execution, at very high volumes you may find the costs soaring well past what it would have been to simply run a `dotnet` process on EC2 or ElasticBeanstalk. The costs of serverless should always be weighed up against the other advantages such as scalability.

Flexibility is another enormous advantage of serverless architectures, as the example in this chapter has shown. In each step we have radically changed the architecture of a part of our Serverless C# Resume Uploader by making very small changes — most of the time without even having to redeploy the rest of the application. This separation between the moving parts of your system makes growth much easier and unlocks diversification of talent within your development team. There is nothing in a serverless architecture that specifies what version, technology, or even programming language the individual components are written in. Have you ever wanted to try your hand at F#? With a system built on AWS Lambda there is nothing stopping you from writing the latest feature in an F# Lambda and slotting it into your serverless architecture. Need to route some HTTP calls to a third party API? You can proxy that directly from API Gateway if you need to, without having to create the interface and the "plumbing" in your code. By adopting Infrastructure as Code tools such as SAM (or third party frameworks such as Serverless Framework and Terraform) you can automate changes to your infrastructure and run code reviews, pull requests, and automated testing pipelines on the infrastructure configuration itself. A pull request for a serverless system will often be composed of two changes: a simple, easy to review AWS Lambda and an entry into the SAM/Terraform/Cloudformation template showing how that Lambda integrates with the rest of the system.

About the Authors

Noah Gift is the founder of Pragmatic A.I. Labs. He lectures at many top Universities including Duke Data Science and AI programs. He teaches and designs graduate machine learning, MLOps, AI, and data science courses, and consults on machine learning and cloud architecture for students and faculty. He is also an AWS ML Hero and Python Software Foundation Fellow. As a former CTO, individual contributor, and consultant he has over 20 years' experience shipping revenue-generating products in many industries including film, games, and SaaS. He also the author hundreds of technical publications, courses and books.

James Charlesworth is a software engineer, manager, blogger, public speaker and developer advocate. Starting his career in industrial control systems James rode the storm surge of cloud computing into web-based applications around 2011. He is a deeply passionate advocate of cloud native architecture, ditching servers and leaning on managed services to provide scalability, reliability and security.