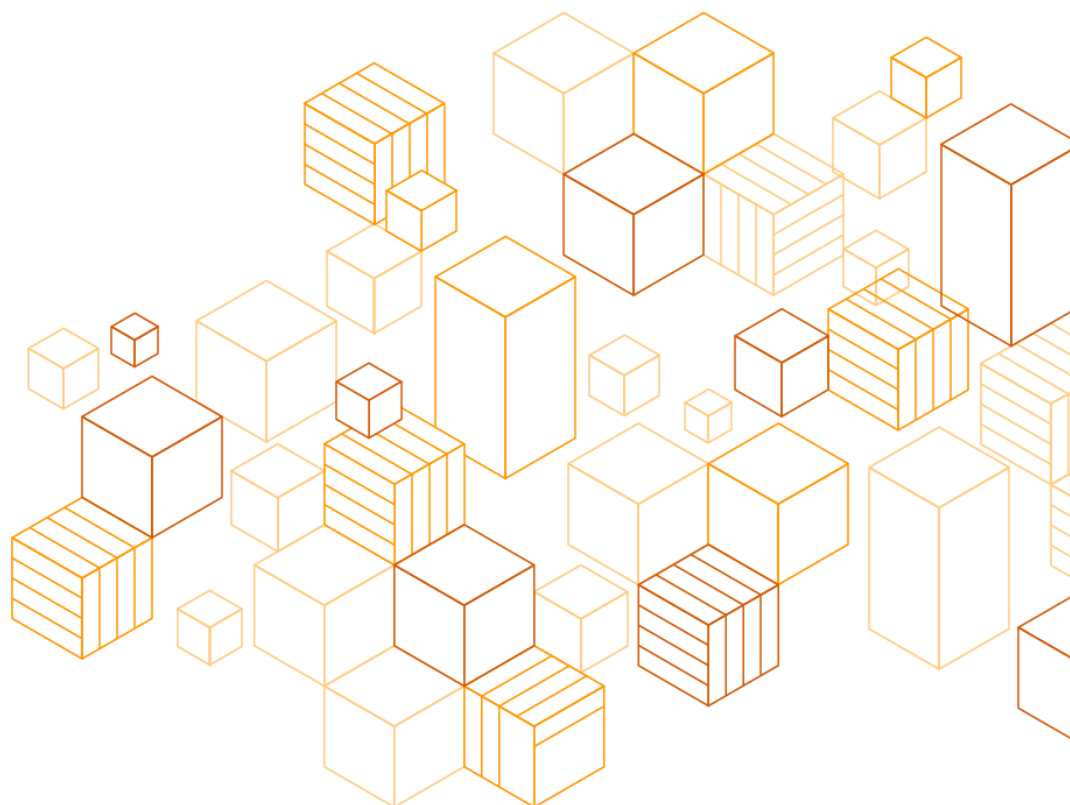


Modernize .NET Applications with Linux Containers

Technical Guide

August 5, 2021



Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided “as is” without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

© 2021 Amazon Web Services, Inc. or its affiliates. All rights reserved.

Contents

Overview	1
Before you begin	2
Understand your drivers.....	2
Build your action plan.....	4
Choosing container orchestration	6
Tools and libraries	7
Cost considerations	8
Cloud computing	8
AWS pricing model.....	8
AWS container services.....	9
Architecture overview	13
Walkthrough.....	16
Refactoring from .NET Framework to .NET 5	16
Replatforming from Windows VMs to Linux containers	50
Logging and monitoring.....	58
Security	60
User to application authentication and authorization	60
Application to database authentication and authorization	61
Identity and access management for Amazon ECS	63
Compliance validation for Amazon ECS	66
In-flight data protection using encryption.....	66
Source code.....	69
Conclusion	69
Contributors	69
Document revisions.....	70

About this guide

Many architects, developers, and IT practitioners want to modernize their existing .NET Framework applications by refactoring to the latest, cross-platform version of .NET (previously referred to as .NET Core) and replatforming from Windows virtual machines (VMs) to Linux containers. This guide outlines a methodology to assess applications that are suitable to move to Linux containers. It describes the business and technical benefits of this approach, and offers a prescriptive procedure using a sample application and reference architecture to guide organizations in the delivery of this process.

For comments, corrections, or questions, see [this form](#).

Overview

Organizations are modernizing their Windows workloads using a combination of re-hosting, replatforming, and refactoring approaches, to take full advantage of cloud economics, unlock innovation for their business, and deliver new functionality to their customers. Many organizations with .NET Framework applications, such as [DraftKings](#), [FileForce](#), and [AgriDigital](#), intend to refactor their applications to the latest, cross-platform version of .NET, and replatform to Linux containers in the cloud. However, the path to achieve this can be challenging. This guide aims to equip architects, developers, and IT professionals with the information they need to soundly assess and safely complete this approach.

Since the introduction of .NET Framework in 2002, more than six million developers have adopted the .NET programming ecosystem to build their applications.¹ In its initial form, .NET was built to run exclusively on the Windows operating system. This resulted in large portfolios of applications running on .NET and Windows, particularly in the enterprise.

However, over the course of the past 10+ years, a lot has changed in the technology industry and the way applications are built. These changes include the rise of the Linux operating system, the growing influence of open source, and widespread adoption of technologies such as the public cloud, containers, and DevOps practices. These trends, among other factors, drove Microsoft to focus their .NET investments on a new version of .NET that was initially called .NET Core and is now simply referred to as .NET for versions 5 and above. This new version of .NET is free, open source, and cross-platform, which brings new capabilities for .NET developers to run their applications anywhere powered by a modular, lightweight framework.

While this open and portable .NET future is welcomed by many in the .NET community, it brings complications for organizations that have invested in the .NET Framework and Windows to power their applications. On one hand, there are attractive benefits of refactoring .NET Framework applications to the latest, cross-platform version of .NET. These include removing Windows licensing costs by moving to Linux, and accessing the latest innovations from the .NET community. On the other hand, refactoring .NET Framework applications to the latest version of .NET is not a small effort, particularly for complex applications that have many dependencies on libraries that do not have cross-platform equivalents, such as ASP.NET Web Forms, Windows Communication Foundation (WCF), .NET Remoting, or Windows Workflow (WF).

In addition to porting to the latest version of .NET, many organizations simultaneously want to move their VM-based deployments to containers, to predictably deploy their applications across environments, maximize the efficiency of their resource consumption, and introduce DevOps practices to automate their development lifecycle.

Gartner predicts that by 2022, more than 75% of global organizations will be running containerized applications in production, up from less than 30% in 2020². An IDC survey found that 45% of respondents' application portfolio is running in containers today, and that is expected to increase to 60% in three years.³

It's clear that containers are becoming a primary mechanism for packaging applications. However, just like the pathway to the latest, cross-platform version of .NET, containers bring their own set of complexities and challenges, particularly for organizations that lack a depth of container expertise.

In the following sections, this paper walks through this use case of modernizing a .NET Framework application running on Windows VMs to .NET 5, and Linux containers running on [Amazon Elastic Container Service](#) (Amazon ECS) and [AWS Fargate](#).

Before you begin

Understand your drivers

Before you begin, take the time to understand your business and technical drivers, and work backwards from your desired results to form a plan of action. Common business and technical drivers that motivate the approach to modernize existing .NET Framework applications with the latest, cross-platform version of .NET and Linux containers are outlined in the following tables.

Table 1 - Business drivers

Driver	Description	Solution
Accelerate innovation	Development and IT teams spend most of their time maintaining existing applications rather than innovating.	Adopt containers to facilitate DevOps practices and automation.
Lower total cost of ownership (TCO)	Licensing costs inflate overall spend and manual processes slow teams down.	Move from Windows to Linux to reduce licensing costs and to containers to optimize resource utilization.

Driver	Description	Solution
Address skills gap	Internal expertise in cloud-native technologies creates barriers to cloud adoption and modernization.	Use AWS modernization programs and experts to upskill your staff.
Improve security and compliance	Unmaintained applications and IT practices pose a risk to the organization's security posture.	Upgrade to the latest version of .NET Core and containers for new security features.

Table 2 - Technical drivers

Driver	Description	Solution
Access the latest enhancements	Teams work around issues that have been solved in newer releases of the software.	Upgrade to .NET Core for new features and access innovations on Linux such as AWS Graviton Processor .
Increase productivity	Manual processes slow teams down and cause delays in release cycles.	Adopt DevOps practices and automation.
Scale with traffic demands	Applications are provisioned for peak load and underutilize infrastructure resources.	Use container automatic scaling and stateless applications to scale up and down as traffic demands.
Predictable performance across environments	Application behavior is not consistent across environments leading to unforeseen problems.	Use containers for isolation and consistent behavior across environments.

If these drivers resonate with you and your teams, then modernizing your .NET Framework applications with the latest, cross-platform version of .NET and Linux containers is likely a good approach to start your modernization journey.

Remember, there are many options for modernizing your existing .NET Framework applications that also include moving to Windows containers, breaking down monoliths to microservices, and rewriting the application to be event-driven with serverless functions and [AWS Lambda](#).

While these other approaches have their own unique set of benefits, refactoring .NET Framework applications to the latest, cross-platform version of .NET and Linux containers brings significant cost savings, as outlined in the next section. This approach

can be a good fit for teams with tight timelines, as the amount of architectural change is generally less than decomposing a monolith or moving to AWS Lambda.

Build your action plan

With any engineering project, it's important to establish a clear plan and scope up front. For example, [DraftKings found the following key principles](#) for success during their .NET refactoring project:

Table 3 – DraftKings key principles for success

Key Principle	Description
Get buy-in from leadership	Seek leadership support by showing a strong business case
Update with a purpose	Limit the scope of changes to the refactoring effort
Modernize incrementally	Start small and invest in automation once the process is proven
Trust but verify	Invest in functional, integration, and performance tests to ensure consistent behavior
Communicate often but deliberately	Communicate regular updates to resolve issues quickly and bring visibility to the progress
Expect unforeseen problems	Expect problems to occur and document them to remove possibilities of repeating mistakes
Plan for support	Plan to support other teams as they embark on following the process to refactor their applications

Once your plan is established, consider the following recommendations for your project. Keep in mind that each project is unique, and you will have to assess and experiment to find the practices that work best for your teams.

Table 4 – Refactoring recommendations

Recommendation	Description
Start with an assessment	Understand the scope of the refactoring effort and determine which libraries will be problematic. The Porting Assistant for .NET can help with this assessment by providing a compatibility report for your existing solutions. Additionally, assess your code for pieces that can be refactored in isolation so progress can be made in smaller, safer steps.
Consider starting with applications that have limited downstream impact	If your organization has several applications that require refactoring, consider starting with functionality that will not break many downstream dependencies. This allows for learning and tuning of the process without as much risk as starting with highly coupled, business critical functionality.
Upgrade tests first	Upgrading tests first helps to ensure that everything works at a functional level, so that changes are verified incrementally as you port to the latest, cross-platform version of .NET.
Verify dependencies work for target OS	Evaluate your third-party dependencies to make sure that they are supported on your target platform (Windows, Linux, macOS) before making large code changes.
Maintain backwards compatibility	When you refactor from .NET Framework to the latest, cross-platform version of .NET, reduce the risk of changing behavior of the application by keeping the changes specific to the refactoring effort, rather than adding to or altering existing functionality.

When building a plan for refactoring, it can be difficult to predict the time and resources that it will take to complete. Every project is different, and the complexity of the applications in scope dictates the level of effort for refactoring.

Many organizations choose to start with a single application, to get a clearer picture of the time and resource requirements, before addressing the rest of the application estate. It's common for organizations to invest in internal best practices and runbooks to expedite the process for future refactoring projects.

The next section reviews the options for running containers on AWS. Choosing a container orchestrator is the first decision point when creating a replatforming plan from VMs to containers.

Choosing container orchestration

As you re-platform your application, you can select a container orchestrator that is most suitable for your requirements. 80% of all containers in the cloud run on AWS,⁴ and you have a broad set of options to run and manage your containers on AWS. When choosing your container orchestration option, you should start with the question, “How much of the container infrastructure do I want to manage?” The following options are available to you:

- **Self-Managed Containers on [Amazon Elastic Compute Cloud \(Amazon EC2\)](#)** — EC2 virtual machines give you full control of your server clusters and provide a broad range of customization options. You can choose Amazon EC2 to run your container orchestration if you need server-level control over the installation, configuration, and management of your container orchestration environment.
- **[Amazon Elastic Kubernetes Service \(Amazon EKS\)](#)** — Amazon EKS is a managed service that makes it easy for you to run Kubernetes on AWS without needing to install and operate your own Kubernetes control plane or worker nodes. Amazon EKS provisions and scales the Kubernetes control plane, including the API servers and backend persistence layer, across multiple AWS [Availability Zones](#) for high availability and fault tolerance. Amazon EKS is integrated with various AWS services such as [Elastic Load Balancing](#), [AWS Identity and Access Management \(AWS IAM\)](#), [Amazon VPC](#), and [AWS CloudTrail](#) to provide scalability and security for your applications.
- **[Amazon Elastic Container Service \(Amazon ECS\)](#)** — Amazon ECS is a highly scalable, high-performance container management service that supports Docker containers and enables you to run applications on a managed cluster of Amazon EC2 instances. With simple API calls, you can launch and stop container-enabled applications, query the complete state of your cluster, and access many familiar features like security groups, Elastic Load Balancing, EBS volumes, and IAM roles. You can use Amazon ECS to schedule the placement of containers across your cluster based on your resource needs and availability requirements.

- **[AWS Fargate](#)** — AWS Fargate is a serverless compute engine for containers that works with both Amazon ECS and Amazon EKS. AWS Fargate removes the need to provision and manage servers and enables you to specify and pay for resources per application. AWS Fargate is the easiest way to get started with containers on AWS. With AWS Fargate, developers don't have to manage the underlying infrastructure and they can launch and scale their containers easily and manage everything at the container level. This guide uses AWS Fargate for ECS to deploy containerized .NET Framework applications on AWS.

The next section covers some of the tools and libraries that are available to you as you build your .NET applications and DevOps automation on AWS.

Tools and libraries

AWS has many tools available for developers and IT practitioners to build and run container applications and infrastructure. The following table covers some of the primary tools that are useful for the refactoring and replatforming process. You'll use many of these tools later on in the Walkthrough section of the guide. For the latest news and releases, visit the [.NET on AWS landing page](#).

Table 5 – Tools and libraries

Tool/Library	Description
AWS SDK for .NET	Use AWS services with purpose-built .NET libraries and APIs
AWS Cloud Development Kit	Define AWS infrastructure directly in code with the Cloud Development Kit for .NET
AWS Toolkit for Visual Studio	Extension for Visual Studio to create, debug, and deploy applications on AWS
Porting Assistant for .NET	Code analysis tool that scans .NET Framework applications and generates a .NET Core/.NET 5+ compatibility assessment
AWS Extensions for .NET CLI	Build .NET applications with the .NET CLI

Tool/Library	Description
AWS Tools for PowerShell	Manage AWS services and resources with PowerShell
AWS Toolkit for Azure DevOps	Extension for Azure DevOps to deploy applications on AWS
AWS Command Line Interface (AWS CLI)	Unified tool to manage your AWS services

Cost considerations

Cloud computing

With cloud computing, companies have access to a scalable platform, low-cost storage, database technologies, and tools on which to build enterprise-grade solutions. Cloud computing helps businesses reduce costs and complexity, adjust capacity on-demand, accelerate time-to-market, increase opportunities for innovation, and enhance security.

Weighing the financial considerations of operating an on-premises data center versus using cloud infrastructure is not as simple as comparing hardware, storage, and compute costs. Whether you own your own data center or rent space at a colocation facility, you have to manage investments that include capital expenditures, operational expenditures, staffing, opportunity costs, licensing, and facilities overhead.

AWS pricing model

AWS offers a simple, consistent, pay-as-you-go pricing model, so you are charged only for the resources you consume. With this model, there are no upfront fees, no minimum commitments, and no long-term contracts required. There is also flexibility to choose the pricing model that best fits your needs if the pay-as-you-go model is not optimal for your use case. Short descriptions of all of these pricing models are found below.

- **On-Demand Instance** — With On-Demand Instances, you pay for compute capacity by the hour, with no minimum commitments required.
- **Reserved Instance** — For longer-term savings, you can purchase in advance. In addition to providing a significant discount (up to 60 percent) compared to On-Demand Instance pricing, Reserved Instances allow you to reserve capacity.

- **Spot Instance** — You can request unused Amazon Elastic Compute Cloud (Amazon EC2) capacity. Instances are charged the Spot Price, which is set by Amazon EC2 and fluctuates, depending on supply and demand. For more information, see [Amazon EC2 Spot Instances Pricing](#).

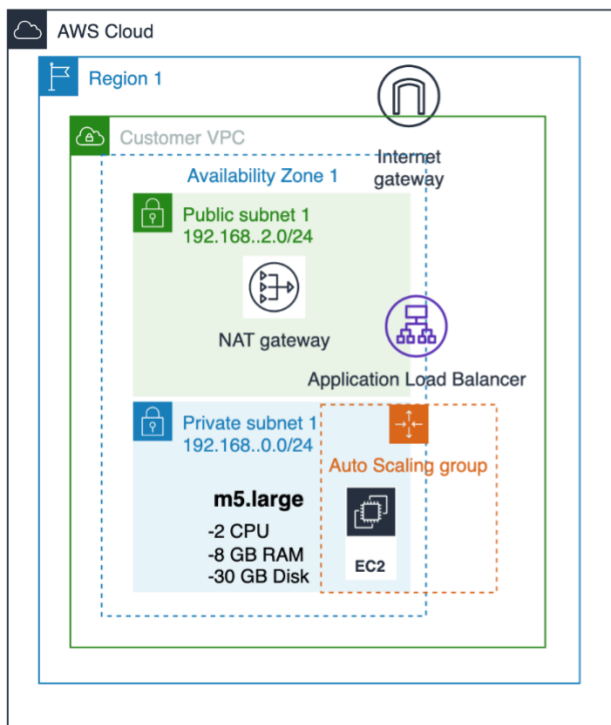
For more information, see the [AWS Cloud Economics Center](#) and [Savings Plans](#).

AWS container services

There are several cost aspects to consider when running applications on AWS. These include, but are not limited to, storage, data transfer, service usage, compute, and operations. This guide focuses on compute (where the containers run) and operations cost (staffing) of managing the compute resources. To simplify the analysis, it does not include the cost of running a database in the example.

As mentioned, there are four different services that AWS provides to run container-based applications: Amazon EC2, Amazon EKS, Amazon ECS, and AWS Fargate. To understand the cost implications of running containers on each of these services, this paper reviews an example of a simple application architecture, and compares the cost of running it on each service with an On-Demand pricing model in the us-east-1 Region. As detailed previously, you could choose a different pricing structure such as Spot Instances or Saving Plans, which are supported for the services covered in this guide. The examples in the following sections were generated by the [AWS Pricing Calculator](#).

As the baseline for the comparison, this paper uses an application running on Windows in Amazon EC2, as shown in the following figure:



Total monthly

\$192.57 USD

- 1 x EC2 (m5.large)
- 1 x NAT Gateway (50 GB)
- 1 x Application Load Balancer (100 GB)

Application architecture in Amazon EC2

Summary by service and operating system

The following table represents the summary of running the preceding application on each service, and compares the monthly cost of running on Windows versus Linux. These figures include the cost of compute, the [Amazon Virtual Private Cloud](#) (Amazon VPC), and [Elastic Load Balancing](#) (ELB). Running on Linux costs less than running on Windows for each service. This is influenced by the license-included cost of Windows. Additionally, the cost for Amazon EKS compared to the other options is higher due to the per-cluster cost of cluster management.

*Table 6 – Sample monthly cost summary by service and OS**

Service	Windows (monthly)	Linux (monthly)
Amazon EC2	\$192.57 (details)	\$125.41 (details)
Amazon EKS	\$265.57 (details)	\$198.41 (details)
Amazon ECS	\$192.57 (details)	\$125.41 (details)
AWS Fargate	N/A	\$137.38**

**This table is an example only. For the most recent pricing information, see the [AWS Pricing Calculator](#).*

***AWS Fargate is not included in the AWS Pricing Calculator. This paper breaks down the costs for AWS Fargate in the following section.*

Self-managed containers on Amazon EC2

Running containers on Amazon EC2 gives you the highest level of control over the underlying compute, but it comes with the highest TCO, because you must manage the entirety of the container's lifecycle. Additionally, you are responsible for optimally utilizing the underlying compute, rather than leaving this to the managed container orchestrator. For more information, see the **details** links in the [preceding table](#).

Amazon EKS

With Amazon EKS, you pay \$0.10 per hour for each cluster that you create. You can use a single Amazon EKS cluster to run multiple applications by taking advantage of Kubernetes namespaces and IAM security policies. You can run Amazon EKS on AWS using either Amazon EC2 or AWS Fargate, and on-premises using [AWS Outposts](#) or [Amazon EKS Anywhere](#).

If you are using Amazon EC2 (including with Amazon EKS-managed node groups), you pay for AWS resources (such as EC2 instances or EBS volumes) you create to run your Kubernetes worker nodes. You pay only for what you use, as you use it; there are no minimum fees and no upfront commitments.

The calculations in this guide use Amazon EC2 for compute. For more details on Amazon EKS pricing, see the [Amazon EKS pricing page](#). For more information, see the **details** links in the [preceding table](#).

Amazon ECS

With Amazon ECS, there is no additional charge for the cluster management. You can run Amazon ECS on AWS using either Amazon EC2 or AWS Fargate, and on-premises using AWS Outposts or Amazon ECS Anywhere. Again, you only pay for what you use, as you use it.

For the calculations in this guide, we used EC2 for compute, and for more details on ECS pricing reference the [Amazon ECS pricing page](#). For more information, see the **details** links in the [preceding table](#).

AWS Fargate

AWS Fargate can be used to run containers on Amazon ECS and Amazon EKS, removing the operational cost of managing the underlying infrastructure yourself. Pricing is calculated based on the vCPU and memory resources used from the time you start to download your container image until the Amazon ECS Task or Amazon EKS Pod ends, rounded up to the nearest second.

For simplicity, this guide assumes that AWS Fargate will run all the time (730 hours per month), but ideally you benefit more from AWS Fargate when cluster utilization falls under certain thresholds. Windows containers are not supported to run on AWS Fargate at this time. Therefore this paper estimates only the cost of running Linux containers. See the [AWS Fargate pricing page](#) for more information.

For details of the Fargate calculations in the [preceding table](#), see the following table.

Table 7 – AWS Fargate cost breakdown (example)

Component	Cost breakdown	Cost (monthly)
Total	\$85.05 + \$35.10 + \$17.23	\$137.38
Compute	vCPU per hour: \$0.04048 RAM GB per hour: \$0.004445 (\$0.04048 x 2 vCPU) + (\$0.004445 x 8GB) = \$0.11652 \$0.11652 x 730 hours	\$85.05
NAT Gateway	Usage monthly cost: \$0.045 * 730 hours = \$32.85 Data Processing Cost: \$0.045 * 50 GB per month = \$2.25	\$35.10
Application Load Balancer	Fixed hourly charges (monthly): \$16.43 LCU charges x 100 GB (monthly): \$0.80	\$17.23

Operational costs (staffing)

One of the benefits of using managed services is that you can save time by not having to perform operations that are considered undifferentiated heavy lifting. Managed services in AWS such as AWS Fargate remove the burden of managing your

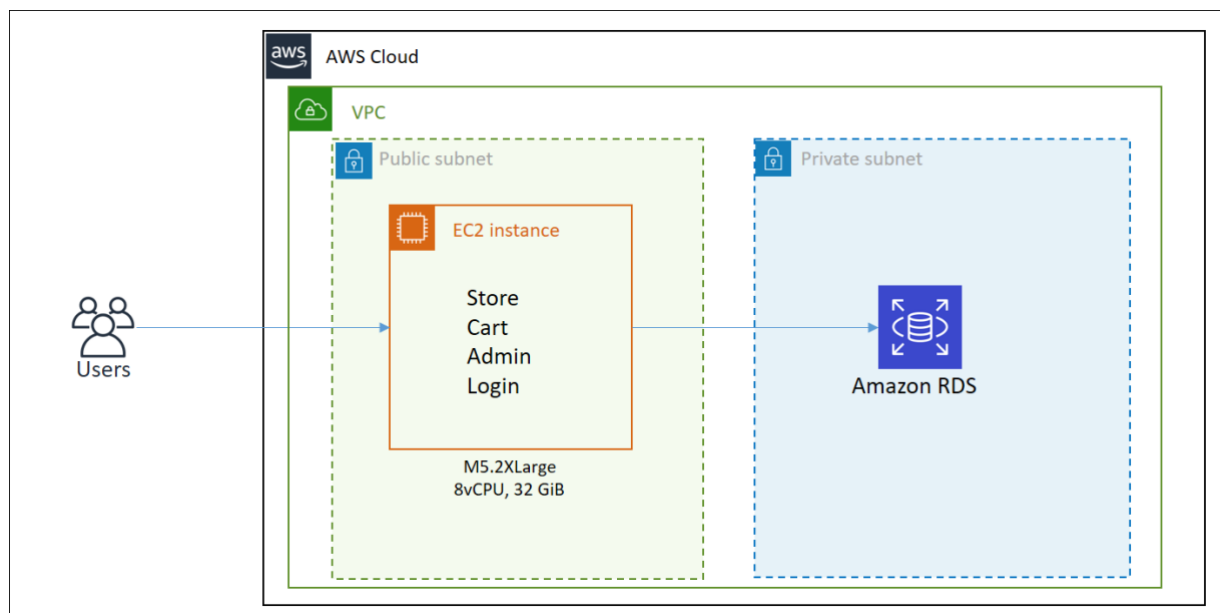
infrastructure, freeing your resources to focus more on building your applications to drive business outcomes. For more information on the operational savings from using managed services such as AWS Fargate, see [Saving money a pod at a time with EKS, Fargate, and AWS Compute Savings Plans](#).

Architecture overview

It's common for enterprise applications built in the last decade to follow a layered [N-Tier](#) architectural approach. The functionality for different aspects of the application are logically separated, yet bundled together as interdependent code modules.

There are multiple advantages of an N-Tier architecture. They're easy to develop, more feasible to test if the application size is small, and can scale vertically. However, as more functionality is added and the code base grows, the applications become cumbersome to manage, change, and scale. State-dependent applications are particularly difficult to scale horizontally, and the compute capacity must be provisioned to consider the peak load.

This guide uses the familiar [MvcMusicStore](#) reference application, which is built on an N-Tier approach using ASP.NET MVC 5 and .NET Framework 4.8. It maintains a session state in memory, and it can scale vertically by default. The data persistence layer uses Microsoft SQL Server. The high-level architecture for this application follows.



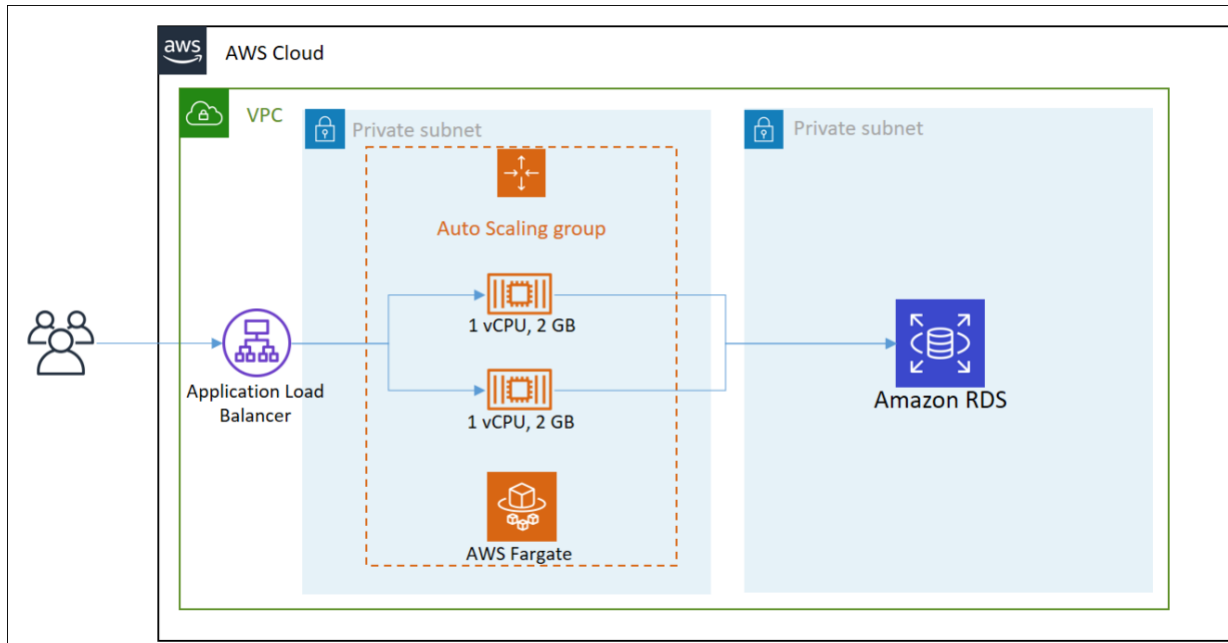
MvcMusicStore application in Amazon EC2 with Amazon RDS database

Many customers start their cloud journey with a lift-and-shift approach, running their N-Tier .NET Framework applications on EC2 without any code changes. It's common for these deployments to have more than one EC2 Windows instance with an [Application Load Balancer](#) (ALB), routing the user requests to one of the EC2 instances. A stateful application can have session affinity (sticky sessions) enabled at the ALB level to create an affinity between a client and a specific EC2 instance.

Along with the ALB, developers can use [AWS Auto Scaling](#) to monitor an application and automatically adjust capacity to maintain steady, predictable performance at the lowest possible cost. [Amazon RDS for SQL Server](#) is a managed database service that frees you up to focus on application development by managing time-consuming database administration tasks, including provisioning, backups, software patching, monitoring, and hardware scaling.

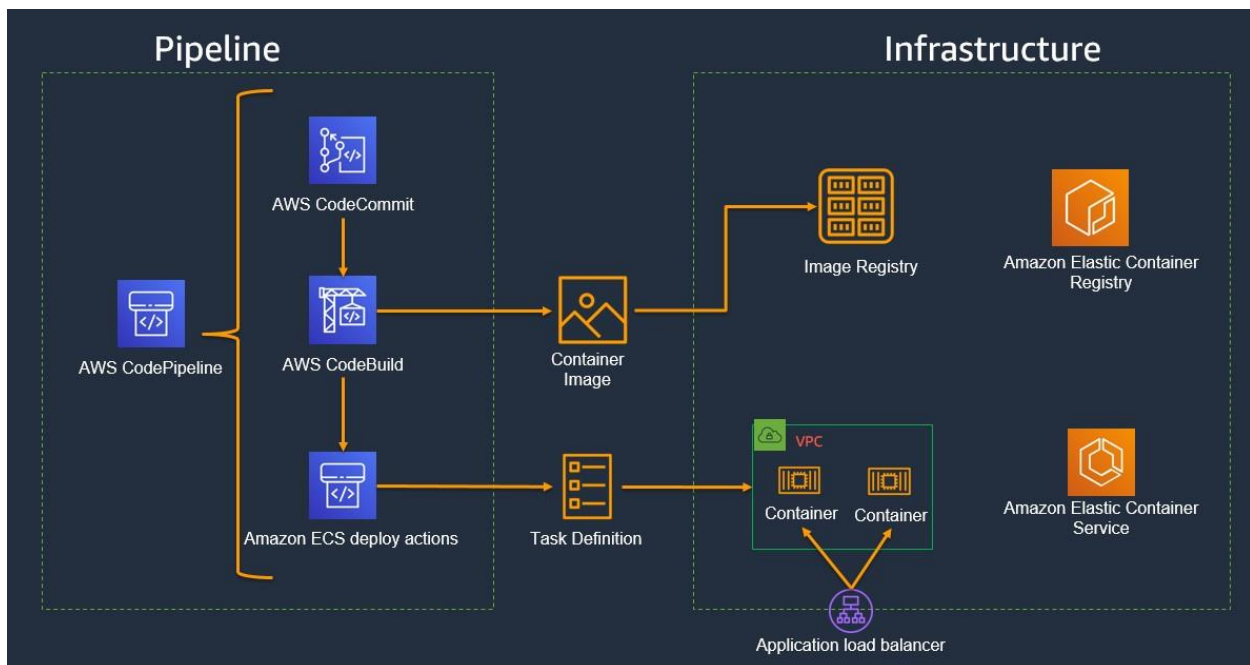
This guide refactors this traditional N-Tier .NET Framework application to run on Amazon ECS with AWS Fargate. As mentioned in the previous sections, running containers on Amazon EC2 and Amazon EKS is also an option, but using Amazon ECS and AWS Fargate is a simple yet powerful place to start if you and your teams are new to containers.

Fargate integrates natively with AWS services such as Application Load Balancer and [AWS Auto Scaling](#), enabling developers to start with the minimum amount of compute to meet user requirements and scale dynamically as the incoming traffic increases. The high-level architecture for the containerized version of this application follows, and is the target of the transformation detailed in this guide. In the following figure, the one vCPU, two GB blocks represent the AWS Fargate tasks where the application containers are running.



MvcMusicStore application in AWS Fargate with Amazon RDS database

Additional benefits of moving to containers are realized when teams also implement automation and DevOps. A cloud-optimized, containerized application allows you to quickly and frequently deliver consistent applications to your users. A common development pipeline for continuous deployment with AWS follows.



DevOps architecture with CI/CD pipeline

Following is a walkthrough of the procedural guidance to accomplish this transformation quickly and safely.

Walkthrough

Refactoring from .NET Framework to .NET 5

The following walkthrough uses the [MvcMusicStore](#) application to demonstrate how to port an ASP.NET MVC and Entity Framework-based application to .NET 5. There are two primary methodologies for refactoring; modify the code in place, or incrementally move code from the original project to a new project.

The first methodology (modify in place) is used in this guide for simplicity, though the second has value for large applications to ensure that the code is buildable throughout the process. Additionally, in this guide, we use the GUI version of the Porting Assistant for .NET, but there is also a [Visual Studio extension](#) available if you prefer to stay in your integrated development environment (IDE).

The majority of the code changes for this walkthrough are driven by the conversion of ASP.NET to ASP.NET Core, which includes modifying the configuration, static content, session management, and identity handling. To minimize changes, Entity Framework (EF) remains during the porting effort, because EF6 is compatible with .NET Core and .NET 5.

For this section of the walkthrough, there are three stages:

- [Prerequisites](#) – Prepare the local development environment for refactoring.
- [Assessment](#) – Use the Porting Assistant to assess compatibility with .NET 5.
- [Refactoring](#) – Port each component of the application to .NET 5.

Prerequisites

1. Install the Porting Assistant for .NET and its prerequisites. See [Getting started with Porting Assistant for .NET.](#)
2. Install Visual Studio 2019 16.4 or later (any version, community, professional or enterprise) with the ASP.NET and web development workload. [Download here.](#)
3. Clone the MvcMusicStore repository ([GitHub repository](#)) and switch to the `net48-upgrade-completed` branch.

4. Install full .NET Framework 4.8 and .NET 5.0.
5. Install and start SQL Server on your local machine. See the [installation instructions](#).
6. Install SQL Server Management Studio. [Download here](#).
7. Set up Membership Schema and database to manage users by opening PowerShell and running the command that matches your local SQL Server installation:

- For SQL Server Express:

```
&
$env:WINDIR\Microsoft.Net\Framework\v4.0.30319\aspnet_regsql.exe
-S .\SQLEXPRESS -d Identity -A all -E
```

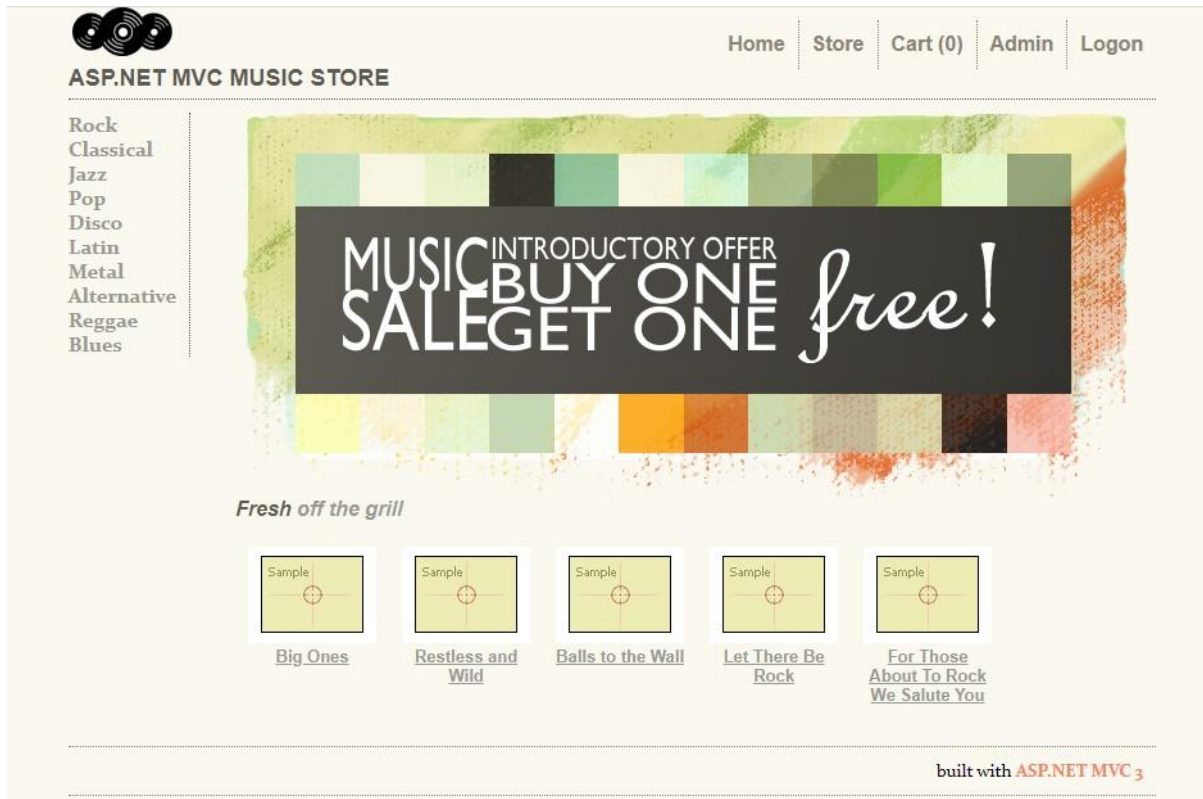
- For other SQL Server versions:

```
&
$env:WINDIR\Microsoft.Net\Framework\v4.0.30319\aspnet_regsql.exe
-S . -d Identity -A all -E
```

8. Open the MvcMusicStore solution in Visual Studio. If using SQL Server Express, in Web.config, change the connectionStrings to the following:

```
<connectionStrings>
  <add name="MusicStoreEntities" connectionString="Data
Source=.\SQLEXPRESS;Initial Catalog=MusicStore;Integrated
Security=SSPI" providerName="System.Data.SqlClient"/>
  <add name="IdentityConnection" connectionString="Data
Source=.\SQLEXPRESS;Initial Catalog=Identity;Integrated
Security=SSPI" providerName="System.Data.SqlClient"/>
</connectionStrings>
```

9. Run the application in Visual Studio by choosing **IIS Express** for hosting and start the application in debug mode. A browser window should start, and you should see the home page for the application, as shown in the following figure.

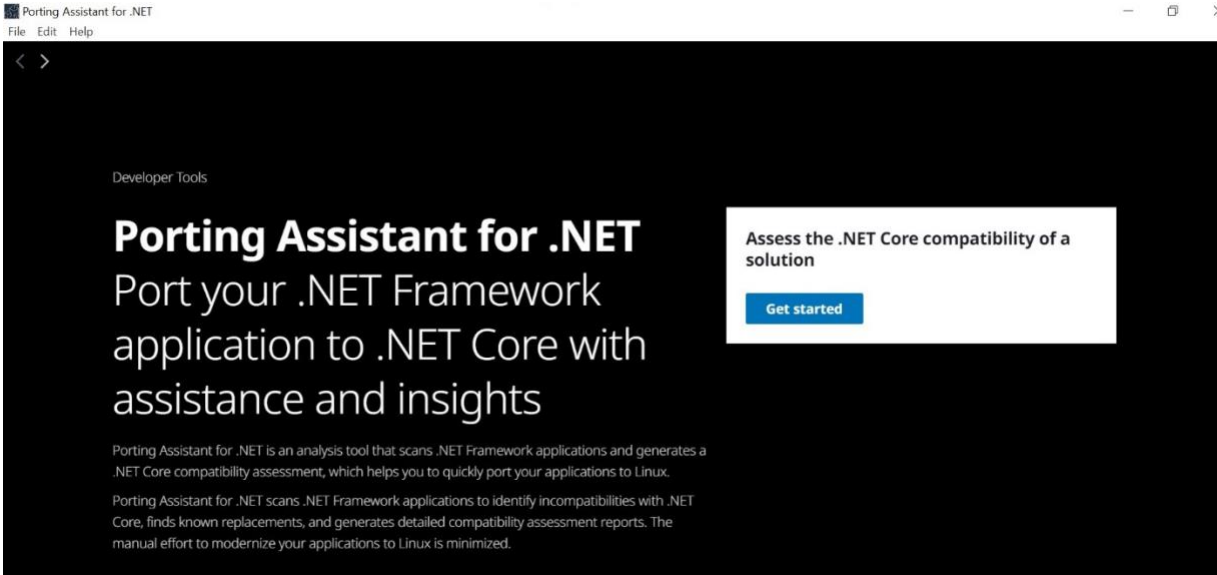


MvcMusicStore homepage user interface (UI)

Assessment

This section assesses the compatibility of the MvcMusicStore application using the Porting Assistant for .NET to identify the source files and libraries that are not compatible with .NET Core/.NET 5. This exercise gives you an idea of the level of effort that refactoring will require. This walkthrough uses Porting Assistant v1.4.3.

1. Start the Porting Assistant application and choose **Get Started**.



Porting Assistant home page

2. On the **Porting Assistant for .NET settings** page, for **Target framework**, choose **.NET 5.0.0**.
3. Specify an **AWS named profile** (if you need to create a profile, see [Create the IAM user](#)).
4. Choose **Next**.

Set up Porting Assistant for .NET [Info](#)

Porting Assistant for .NET settings

Target framework
Select a target framework to allow Porting Assistant for .NET to assess your solution for .NET Core compatibility.

.NET 5.0.0 ▼

AWS named profile
Select an AWS profile to allow Porting Assistant for .NET to assess your solution for .NET Core compatibility. You can also add an AWS named profile using the AWS CLI. [Learn more](#) [↗](#)

▼

[Add a named profile](#)

Porting Assistant for .NET data usage sharing
When you share your usage data, Porting Assistant for .NET will collect information only about the public NuGet packages, APIs, build errors and stack traces. This information is used to make the Porting Assistant for .NET product better, for example, to improve the package and API replacement recommendations. Porting Assistant for .NET does not collect any identifying information about you. [Learn more](#) [↗](#)

I agree to share my usage data with Porting Assistant for .NET - *optional*
You can change this setting at any time on the Porting Assistant for .NET Settings page.

Cancel [Next](#)

Porting Assistant setup page

5. On the **Specify a solution file path** page, for **Solution file**, choose the **MvcMusicStore.sln** file for the project.
6. Choose **Assess**.

Porting Assistant for .NET > Assessed solutions > Assess a new solution

Assess a new solution [Info](#)

Specify solution file path [Info](#)

Solution file

Specify the source file for your solution.

 Choose file

Filetype must be .sln

MvcMusicStore.sln

C:\Users\██████████\dotnet-modernization-music-store\MvcMusicStore.sln

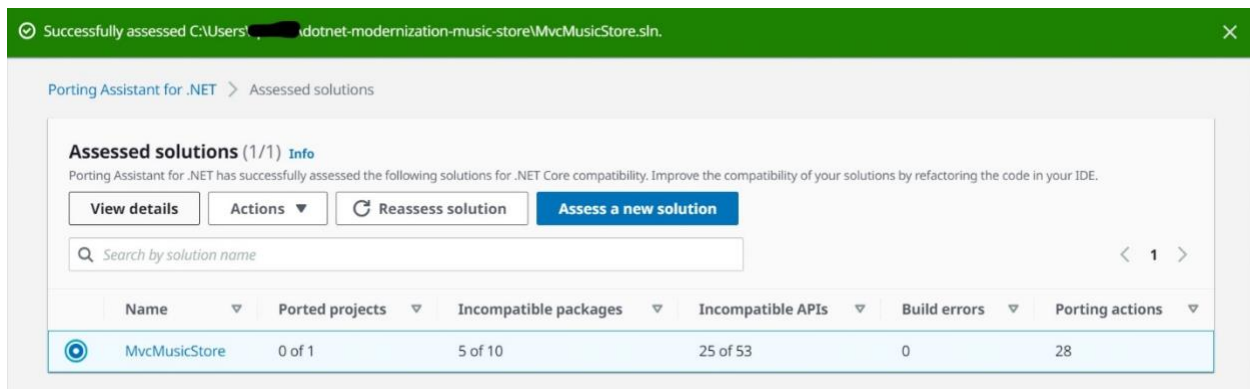
Cancel

Assess

Porting Assistant assess a new solution UI

A **Successfully assessed** notification banner appears with the MvcMusicStore application listed as an assessed solution.

- In the list of **Assessed solutions**, choose the **MvcMusicStore** application, and choose **View details**.



Successfully assessed C:\Users\██████████\dotnet-modernization-music-store\MvcMusicStore.sln

Porting Assistant for .NET > Assessed solutions

Assessed solutions (1/1) [Info](#)

Porting Assistant for .NET has successfully assessed the following solutions for .NET Core compatibility. Improve the compatibility of your solutions by refactoring the code in your IDE.

[View details](#)
[Actions](#)
[Reassess solution](#)
[Assess a new solution](#)

Search by solution name

Name	Ported projects	Incompatible packages	Incompatible APIs	Build errors	Porting actions
MvcMusicStore	0 of 1	5 of 10	25 of 53	0	28

Porting Assistant assessed solutions UI

This action takes you to the **Assessment overview** page, which displays a summary of the number of incompatible packages, incompatible APIs, build errors, and number of suggested porting actions. To share the results of the assessment, choose **Export assessment report**.

Porting Assistant for .NET > Assessed solutions > MvcMusicStore

MvcMusicStore [Info](#)

Improve the compatibility of your solution by refactoring the source code for each project and reassessing it.

[Export assessment report](#) [Reassess solution](#) [Port solution](#)

Assessment overview

The level of compatibility will affect the effort required to port your solution to .NET Core.

Category	Incompatible	Compatible
Incompatible NuGet packages Info	5 of 10	5
Incompatible APIs Info	25 of 53	28
Build errors Info	0	0
Porting actions Info	28	0

Filepath
C:\Users\...\.dotnet-modernization-music-store\MvcMusicStore.sln

Porting Assistant assessment overview UI

Further down on the page, there are additional details of the assessment that include the following:

- **Projects tab** — Lists the projects in your solution, the target framework for the solution, the number of referenced projects in the solution, the number of incompatible packages in the solution, the number of incompatible APIs in the solution, and the port status (ported or not ported) of the solution.
- **Project references tab** — Displays the project references graph, which is a graphical representation of your projects and references. Select a node to see its project dependencies.
 - To port your projects, start by selecting the base of your library, and then move outward to test each layer. You should first port base libraries with the most dependencies from other projects (libraries that show more inward arrows than outward arrows).
 - To view details about a node, select the node and choose **View details**.
- **NuGet packages tab** — Lists the number of NuGet packages in your solutions file, the name of the NuGet packages, the version number of the packages, the source files in the package, the number of compatible APIs in the package, and the compatibility status and suggested replacement of each package.

- **APIs tab** — Lists the name of each API call in the solution, the name of the package within which the API call appears, the number of source files that include each API call, the suggested replacement for the API call, and the compatibility status of the API call.
- **Source files tab** — Lists the source file name in the solution and the number of incompatible API calls over the total number of API calls for each source file. You can select a source file to view the incompatible API calls and replacement suggestions in the source code. In each source file, any section of code that is detected as incompatible will be highlighted as follows:
 - **Porting action** — Code that initiates a porting action in the project
 - **Incompatible method invocation** — An API that is incompatible with .NET 5

Browse each of the tabs and familiarize yourself with the proposed changes by reviewing each source file. Once you have completed your review, you will have an understanding of the changes required to complete the refactor to .NET 5. Proceed to the next section to begin the refactoring effort.

Refactoring

Now that you've assessed the solution, you're ready to port the project to .NET 5. For simplicity of this guide, you will port the solution in place and make the necessary modifications to each component of the application's functionality. If you want to jump directly to the .NET 5 compatible version, you can check out the refactored `framework-to-core-completed` branch.

Use Porting Assistant to initiate the refactor

1. On the **Assessment overview** page of the **Porting Assistant**, choose **Port solution**.

The screenshot shows the 'Assessment overview' page in the Porting Assistant for .NET. At the top, there are three buttons: 'Export assessment report', 'Reassess solution', and 'Port solution' (highlighted with a red border). Below the buttons, the 'Assessment overview' section provides a summary of compatibility issues:

- Incompatible NuGet packages:** 5 of 10. A progress bar shows 5 incompatible (dark grey) and 5 compatible (light grey) packages.
- Incompatible APIs:** 25 of 53. A progress bar shows 25 incompatible (dark grey) and 28 compatible (light grey) APIs.
- Build errors:** 0.
- Porting actions:** 28.
- Filepath:** C:\Users\...\.dotnet-modernization-music-store\MvcMusicStore.sln

Porting Assistant assessment overview UI

2. On the **How would you like to save ported projects** dialog box, for **Write method**, choose **Modify source in place**, and choose **Save**.

The dialog box titled 'How would you like to save ported projects?' has a close button (X) in the top right corner. Under the 'Write method' section, there is a dropdown menu with the text 'Specify how you want Porting Assistant for .NET to store your solution and its ported projects.' The dropdown menu is currently set to 'Modify source in place'. At the bottom right of the dialog, there are two buttons: 'Cancel' and 'Save' (highlighted with a red border).

Porting Assistant write method UI

3. On the next page review the Port destination, new target framework version, and the NuGet package upgrades. Then, choose **Port**.
4. After the process completes, choose **View log** to view the log of the changes made by the Porting Assistant.
5. After you have reviewed the changes, open Visual Studio. The next step is to make manual changes that are required to complete the port to .NET 5.

Add dependencies required for the refactor

Before changing the application source code, install the dependencies you need for the .NET 5 version of your application. To follow the best practice of minimizing the changes to our application during the refactor, continue using `EntityFramework` because it is compatible with .NET Core, and will remove the change that was automatically made by the Porting Assistant to switch to `EntityFrameworkCore`. Additionally, the .NET Framework Membership APIs and Forms Authentication are not compatible with .NET 5, so you must switch to ASP.NET Core Identity. To support these changes, take the following actions:

1. Remove the `Microsoft.EntityFrameworkCore` Package Reference from the `.csproj` file:

```
<ItemGroup>
  <PackageReference Include="Microsoft.EntityFrameworkCore"
Version="*" />
  ...
</ItemGroup>
```

2. Navigate to **Tools > NuGet Package Manager > NuGet Package Console** and install the following dependencies:

```
Install-Package EntityFramework -Version 6.4.4
Install-Package Microsoft.AspNetCore.Identity.EntityFrameworkCore
Install-Package Microsoft.AspNetCore.Authentication.Cookies
Install-Package Microsoft.EntityFrameworkCore.SqlServer
```

Set up configuration

In ASP.NET Core, the `Startup.cs` class replaces the .NET Framework `Global.asax` class. It acts as an entry point to the application, setting up configuration and wiring up services (dependency injection). You can also configure the request pipeline in the `Startup.cs` class to handle all requests made to the application.

Note the following method calls in `Startup.cs`:

- The `AddControllersWithViews` extension method registers MVC services for controllers and views.
- The `UseStaticFiles` extension method adds the static file handler to support images, CSS files, and so on. Order is important here and the `UseStaticFiles` extension method must be called before `UseRouting`.

- The `UseRouting` extension method adds MVC Controller routing.
1. The .NET Framework version of the `MvcMusicStore` application uses `PascalCase` for JSON serialization. To maintain same behavior, set up `JsonSerializerOptions` in the `ConfigureServices` method of `Startup.cs`:

```
public class Startup
{
    [...]

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllersWithViews()
            .AddJsonOptions(jsonOptions =>
                jsonOptions.JsonSerializerOptions.PropertyNamingPolicy = null);
    }

    [...]
}
```

2. Navigate to the `appsettings.json` file in the project root, where you will see the settings that were ported from `Web.config`. Note the `ConnectionStrings` section and ensure it matches your SQL Server installation. If you are not using SQL Server Express, remove `\\SQLEXPRESS` from each string.

```
"ConnectionStrings": {
    "LocalSqlServer": "data source=.\SQLEXPRESS;Integrated
Security=SSPI;AttachDBFilename=|DataDirectory|aspnetdb.mdf;User
Instance=true",
    "MusicStoreEntities": "Data Source=.\SQLEXPRESS;Initial
Catalog=MusicStore;Integrated Security=SSPI",
    "IdentityConnection": "Data Source=.\SQLEXPRESS;Initial
Catalog=Identity;Integrated Security=SSPI"
}
```

Migrate static content and layout

ASP.NET MVC applications depend upon a well-known folder structure. The scaffolding depends on “View source code” files being in the `Views` folder, “Controller source code” files being in the `Controllers` folder, and so on. Some of the non-.NET specific folders are also at the same level, such as `CSS` and `images`. In ASP.NET Core MVC, most of the structure is the same, with the exception of static content, which should be in the `wwwroot` folder. This includes `Javascript`, `CSS`, and `image` files.

1. To adhere to the structure of ASP.NET Core MVC applications, move the `Content` and `Scripts` folders to the `wwwroot` folder. If you cannot see this directory, in the Solution Explorer, choose **Show All Files**.
2. Since the target is Linux, ensure that the case is correct for the static images by copying the source code from [this file](#) to `wwwroot/Content/Site.css`.

Set up EntityFramework for MusicStoreEntities

You saw earlier that the Porting Assistant automatically converted the `EntityFramework` usage to `EntityFrameworkCore`. In this section you'll roll back those changes and configure the `MusicStoreEntities` model to use `EntityFramework`, because `EntityFramework` is supported in .NET Core, and you want to minimize the changes made to the application.

1. In `Models/MusicStoreEntities.cs`, replace `using Microsoft.EntityFrameworkCore` with `using System.Data.Entity`.
2. Add a constructor which takes the connection string:

```
public class MusicStoreEntities : DbContext
{
    public MusicStoreEntities(string nameOrConnectionString)
        : base(nameOrConnectionString)
    {
    }

    [...]
}
```

3. Remove the `OnConfiguring` method.
4. In the ASP.NET version of the application, you specified configuration via XML, but in ASP.NET Core all configuration is code-based. You implement this by creating a subclass of `System.Data.Entity.Config.DbConfiguration` and applying `System.Data.Entity.DbConfigurationTypeAttribute` to the `MusicStoreEntities DbContext` subclass:

```
namespace MvcMusicStore.Models
{
    [DbConfigurationType(typeof(CodeConfig))]
    public class MusicStoreEntities : DbContext
    {
        [...]
    }
}
```

```
public class CodeConfig : DbConfiguration
{
    public CodeConfig()
    {
        SetProviderServices("System.Data.SqlClient",
System.Data.Entity.SqlServer.SqlProviderServices.Instance);
    }
}
}
```

5. In the `Startup.cs` class, within `ConfigureServices`, add factory method for the `MusicStoreEntities` context with its connection string. The `DbContext` should be resolved once per request to ensure optimal performance and reliable operation of `EntityFramework`.

```
using MvcMusicStore.Models;

[...]

public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews()
        .AddJsonOptions(jsonOptions =>
jsonOptions.JsonSerializerOptions.PropertyNamingPolicy = null);
    services.AddScoped(_ => new
MusicStoreEntities(Configuration.GetConnectionString("MusicStoreEnt
ities")));
}
```

Remove EntityFramework functionality that is not supported in .NET Core

1. In `Models/AccountModels.cs`, remove the following line:

```
using Compare =
System.ComponentModel.DataAnnotations.CompareAttribute;
```

2. In `Models/Album.cs`, add using `Microsoft.AspNetCore.Mvc.ModelBinding`, and change the class attribute `[Bind(Exclude="AlbumId")]` to `[Bind]`, because the property in the `Bind` attribute class is not available in .NET Core.
3. Add the attribute `[BindNever]` to the `AlbumID` property:


```
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace MvcMusicStore.Models
{
    [Bind]
    public class Album
    {
        [ScaffoldColumn(false)]
        [BindNever]
        public int AlbumId { get; set; }

        [...]
    }
}
```

4. In `Models/Order.cs`, apply the same changes in steps 2 and 3: add `using Microsoft.AspNetCore.Mvc.ModelBinding`, replace `[Bind(Exclude="OrderId")]` with `[Bind]`, and add attribute `[BindNever]` to the `OrderId` property.

Refactor the landing page functionality

In this section, you'll refactor the functionality that includes fetching music albums, the genre menu, and the customer shopping cart summary.

1. `Controllers/HomeController.cs` depends on the `MusicStoreEntities DbContext` class to fetch data from the database. Add a new constructor with `MusicStoreEntities` in `HomeController.cs`. Dependency injection will resolve the instance of `MusicStoreEntities` registered in `Startup.cs`, while creating the instance of `HomeController` and injecting it at runtime.
2. Update the code for `HomeController`:

```
public class HomeController : Controller
{
    // replace MusicStoreEntities storeDB = new
    MusicStoreEntities();
    // with the code below
    private readonly MusicStoreEntities storeDB;

    public HomeController(MusicStoreEntities musicStoreEntities)
    {
        storeDB = musicStoreEntities;
    }

    [...]
}
```

Refactor the store functionality

The `MvcMusicStore` application enables customers to browse music albums through music category pages and detail pages. Data for music albums is loaded dynamically from the database.

The following components don't exist in ASP.NET Core:

- `System.Web`
- `ChildActionOnlyAttribute`

1. In `Controllers/StoreController.cs`, add a new constructor and inject `MusicStoreEntities DbContext` class:

```
public class StoreController : Controller
{
    // replace MusicStoreEntities storeDB = new
    MusicStoreEntities();
    // with the code below
    private readonly MusicStoreEntities storeDB;

    public StoreController(MusicStoreEntities musicStoreEntities)
    {
        storeDB = musicStoreEntities;
    }

    [...]
}
```

2. Delete the `GenreMenu()` method. You'll convert it to the `ViewComponent` later in this guide.

3. Delete the corresponding **View for Genre** menu by deleting the file
`Views/Store/GenreMenu.cshtml`

Refactor the shopping cart functionality

Users of the MvcMusicStore application are allowed to add items to the cart, either as anonymous or as authenticated users. If a customer adds items to the cart as an anonymous user, after their authentication, shopping cart functionality attempts to merge the current cart with the cart items stored in the database.

The Shopping Cart business logic is embedded in `Models/ShoppingCart.cs`. This class depends on the `MusicStoreEntities DbContext` to fetch the cart items from the database and it uses `session` to store `CartId`.

1. Add new constructor to `Models/ShoppingCart.cs` to inject `DbContext` and `IHttpContextAccessor`. You have already registered `MusicStoreEntities DbContext` and `HttpContextAccessor` classes to `ServiceCollection` in `Startup.cs`.
2. Set `ShoppingCardId` as a part of the constructor to ensure `CartId` is always set as a part of cart object:

```
public partial class ShoppingCart
{
    // replace MusicStoreEntities storeDB = new
    MusicStoreEntities();
    // with the code below
    private readonly MusicStoreEntities storeDB;
    private readonly IHttpContextAccessor httpContextAccessor;

    public ShoppingCart(MusicStoreEntities musicStoreEntities,
        IHttpContextAccessor httpContextAccessor)
    {
        this.storeDB = musicStoreEntities;
        this.httpContextAccessor = httpContextAccessor;

        this.ShoppingCardId = this.GetCartId();
    }

    [...]
}
```

3. Delete static methods `GetCart (HttpContextBase context)` and `GetCart (Controller controller)` from `ShoppingCart.cs`, as these are not needed anymore. These static methods return the `ShoppingCart` object and set `cartId` before returning shopping cart object. You'll address this in the coming steps.
4. Update the `GetCartId` method to use `ISession`:

```
public string GetCartId()
{
    HttpContext context = this.HttpContextAccessor.HttpContext;

    if (context.Session.GetString(CartSessionKey) == null)
    {
        if (!string.IsNullOrWhiteSpace(context.User?.Identity?.Name))
        {
            context.Session.SetString(CartSessionKey,
                context.User.Identity.Name);
        }
        else
        {
            // Generate a new random GUID using System.Guid class
            Guid tempCartId = Guid.NewGuid();

            // Send tempCartId back to client as a cookie
            context.Session.SetString(CartSessionKey,
                tempCartId.ToString());
        }
    }
    return context.Session.GetString(CartSessionKey).ToString();
}
```

5. In `Startup.cs`, add dependency injection for the shopping cart in the `ConfigureServices` method, as shown in the following code example. This enables you to inject the shopping cart object to classes that depend on it:

```
public void ConfigureServices(IServiceCollection services)
{
    [...]
    services.AddScoped<ShoppingCart>();
}
```

6. Refactor `Controllers/ShoppingCartController.cs`. The `ShoppingCartController` loads customer cart data from the database using `MusicStoreEntities.cs`, and also manages it using the `ShoppingCart.cs` object. Add a dependency to `MusicStoreEntities` and `ShoppingCart` objects as a part of constructor and update the class:

```
public class ShoppingCartController : Controller
{
    // replace MusicStoreEntities storeDB = new
    MusicStoreEntities();
    // with the code below
    private readonly MusicStoreEntities storeDB;
    private readonly ShoppingCart shoppingCart;

    public ShoppingCartController(MusicStoreEntities
    musicStoreEntities, ShoppingCart shoppingCart)
    {
        this.storeDB = musicStoreEntities;
        this.shoppingCart = shoppingCart;
    }

    [...]
}
```

7. Now that you have the `ShoppingCart` object dependency injected through the constructor, you can replace the static method `ShoppingCart.GetCart(this.HttpContext)` with the `ShoppingCart` object, as shown in the following code example. Do this for the `Index()`, `AddToCart(int id)`, and `RemoveFromCart(int id)` Actions.

```
public ActionResult Index()
{
    // replace var cart = ShoppingCart.GetCart(this.HttpContext);
    // with the code below
    var cart = shoppingCart;
    [...]
}
```

8. Lastly, delete the `CartSummary()` method, as it's not needed anymore. You'll convert it in the next section.

Create view components

In the .NET Framework version of the MvcMusicStore application, there are few reusable components, Cart Summary and Genre Menu, that are rendered from the `_Layout.cshtml` [Razor page](#) using the `@Html.RenderAction()` method. These reusable components consist of both Razor markup and some backend logic. In the .NET Framework version of the app, the backend logic is implemented as controller actions marked with a `[ChildActionOnly]` attribute, and the markup is implemented as a normal Razor view.

Child Actions are not supported in ASP.NET Core MVC, and they are replaced with a new View Component feature. Conceptually, View Components are very similar to Child Actions, but they are lightweight and they do not participate in controller lifecycle (they cannot leverage the filter pipeline `OnActionExecuting()` and `OnActionExecuted()` actions, nor do they use model binding). Generally, a View Component is created by creating a class where the name ends with the suffix `ViewComponent`. It fully supports constructor dependency injection.

1. Add a new folder at the root level of the project called `ViewComponents`.
2. Right-click on the `ViewComponents` folder and choose **Add > New Item**.
3. Choose **C# class**.
4. Name the class `CartSummaryViewComponent.cs`.
5. Choose **Add**.
6. Add a new constructor with `ShoppingCart` and add an `Invoke()` method by copying the code from [this file](#).
7. Add a `CartSummary` Razor View Component: right-click **Views > Shared** and add a new Components folder.
8. Right-click the **Components** folder and add the new sub-folder `CartSummary`.
9. Right-click the **CartSummary** folder, select **Add > View**, and choose **Razor View - Empty**.
10. Name the View `Default.cshtml`.
11. Add the following code example to `Default.cshtml`, which corresponds to the code from `Views/ShoppingCart/CartSummary.cshtml`.
12. Delete `Views/ShoppingCart/CartSummary.cshstml`.

```
@Html.ActionLink("Cart (" + ViewData["CartCount"] + ")",
"Index",
"ShoppingCart",
new { id = "cart-status" })
```

13. Add a new Genre View Component class: right-click the **ViewComponents** folder and select **Add > New Item**.
14. Choose **C# class**.
15. Name the class `GenreMenuViewComponent.cs`.
16. Choose **Add**.
17. Add a new constructor with `MusicStoreEntities`.
18. Add an `Invoke()` method by copying the code from [this file](#).
19. Add the Genre Menu Razor View Component: right-click **Views > Shared > Components folder**.
20. Add new sub-folder `GenreMenu`.
21. Right-click the `GenreMenu` folder, choose **Add > View**, and choose **Razor View - Empty**.
22. Name the View `Default.cshtml`.
23. Add the following code example to `Default.cshtml`, which corresponds to the code from `Views/Store/GenreMenu.cshtml`.
24. Delete `Views/Store/GenreMenu.cshtml`.

```
@model IEnumerable<MvcMusicStore.Models.Genre>

<ul id="categories">
  @foreach (var genre in Model)
  {
    <li>
      @Html.ActionLink(genre.Name,
        "Browse", "Store",
        new { Genre = genre.Name }, null)
    </li>
  }
</ul>
```

25. Replace the `RenderAction` code in `Views/Shared/_Layout.cshtml` with the code to invoke the `ViewComponent`:

```
<body>
  <div id="header">
    <ul id="navlist">
      [...]
      <li><a href="@Url.Content("~/Store/")">Store</a></li>
      <li>@await Component.InvokeAsync("CartSummary")</li>
      [...]
    </ul>
  </div>

  @await Component.InvokeAsync("GenreMenu")

  <div id="main">
    @RenderBody()
  </div>

  [...]

</body>
```

Configure identity and authentication

In the .NET Framework version of the `MvcMusicStore` application, user management and authentication are configured using `Membership APIs` and `Form authentication`. Both of these libraries are not supported in `ASP.NET Core`. They are replaced by `ASP.NET Core Identity`. This migration requires changing the database schema and data that the application uses to manage users and handle authentication. You will create the new schema and run the data migration at the end of the code refactor.

1. Unlike the `Membership API` that uses `ADO.NET` to access `SQL` database, `ASP.NET Core Identity` uses `Entity Framework` to manage users. Update `Startup.cs` with the following code example to configure `Entity Framework` and `Identity services` and set the login path to `/Account/Logon`. This redirects an unauthenticated user to the `Logon` page if they try to access functionality that requires authentication. For example, the `checkout` page will redirect an unauthenticated user to the `Logon` page for authentication. Note that there are two types in this code snippet, `ApplicationDbContext` and `User`, that haven't been created yet. You create them in subsequent steps.


```
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Identity;

namespace MvcMusicStoreCore
{
    [...]
    public void ConfigureServices(IServiceCollection services)
    {
        [...]

        services.AddDbContext<ApplicationDbContext>(options =>
options.UseSqlServer(Configuration.GetConnectionString("Identity
Connection")));

        services.AddIdentity<User, IdentityRole>()
            .AddEntityFrameworkStores<ApplicationDbContext>()
            .AddDefaultTokenProviders();

        services.ConfigureApplicationCookie(options =>
options.LoginPath = "/Account/Logon");
    }
    [...]
}
```

2. Enable Identity by calling `UseAuthentication` in the `Configure` method, as shown in the following code example. `UseAuthentication` adds authentication middleware to the request pipeline for the application.

```
public void Configure(IApplicationBuilder app,
IWebHostEnvironment env)
{
    [...]

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseSession();

    [...]
}
```

3. Add the `ApplicationDbContext` and `User` types. Go to the `Models` folder and add a new class, `User.cs`, that extends the `IdentityUser`.

4. Add the properties that map back to the `AspNetUser` columns in the database.

```
using Microsoft.AspNetCore.Identity;

namespace MvcMusicStore.Models
{
    public class User : IdentityUser
    {
        public string PasswordQuestion { get; set; }
        public string PasswordAnswer { get; set; }
        public bool IsApproved { get; set; }
    }
}
```

5. Create another class, `ApplicationDbContext.cs`, that extends `IdentityDbContext`, passing in the `User` class created in the preceding step. ASP.NET Core Identity requires an Entity Framework `DbContext` class to talk to tables and populate the models. The base `IdentityDbContext<TUser>` class is available in the `Microsoft.AspNetCore.Identity.EntityFrameworkCore.dll` file, which interacts with the Identity tables to retrieve and store information. The `TUser` class can be any class that extends the `IdentityUser` class.

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

namespace MvcMusicStore.Models
{
    public class ApplicationDbContext : IdentityDbContext<User>
    {
        public
        ApplicationDbContext(DbContextOptions<ApplicationDbContext>
            options)
            : base(options)
        {
        }

        protected override void OnModelCreating(ModelBuilder
            builder)
        {
            base.OnModelCreating(builder);
        }
    }
}
```

Refactor registration and login

In this section, you migrate the business logic and views for registration and login. The business logic is implemented in `AccountController.cs`, and markup is implemented as normal Razor views. You refactor the code to replace:

- The Membership API with ASP.NET Core Identity to manage users
- Forms authentication with ASP.NET Core Identity to manage authentication
- `HttpContextBase` with `IHttpContextAccessor` to manage session data

1. In `Controllers/AccountController.cs`, remove `using System.Web.Security` and add a new constructor with dependency to `userManager<User>`, `ShoppingCart`, `IHttpContextAccessor`, and `SignInManager<User>`.

- `userManager<User>` class manages users, passwords, profile data, etc.
- `SignInManager<TUser>` manages authentication cookie and handles login / logout functionality
- `IHttpContextAccessor` allows access to `HttpContext` and `Session`
- `ShoppingCart` manages the cart business logic

```
using MvcMusicStore.Models;
using System;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace MvcMusicStore.Controllers
{
    public class AccountController : Controller
    {
        private readonly UserManager<User> userManager;
        private readonly ShoppingCart shoppingCart;
        private readonly IHttpContextAccessor httpContextAccessor;
        private readonly SignInManager<User> signInManager;
        public AccountController(UserManager<User> userManager,
            ShoppingCart shoppingCart,
            IHttpContextAccessor httpContextAccessor,
            SignInManager<User> signInManager)
        {
            this.userManager = userManager;
            this.shoppingCart = shoppingCart;
            this.httpContextAccessor = httpContextAccessor;
        }
    }
}
```

```
this.signInManager = signInManager;
}

[...]
}
}
```

2. Now that you have `ShoppingCart` and `HttpContextAccessor` injected through the constructor, fix the `MigrateShoppingCart` method:

```
private void MigrateShoppingCart(string UserName)
{
    // Associate shopping cart items with logged-in user
    var cart = shoppingCart;

    cart.MigrateCart(UserName);

    httpContextAccessor.HttpContext.Session.SetString(ShoppingCart.CartSessionKey, UserName);
}
```

3. Add reusable methods for redirecting the user back to the local return URL and add error messages to `ModelState`:

```
private ActionResult RedirectToLocal(string returnUrl)
{
    if (Url.IsLocalUrl(returnUrl))
    {
        return Redirect(returnUrl);
    }
    else
    {
        return RedirectToAction("Index", "Home");
    }
}

private void AddErrors(IdentityResult result)
{
    foreach (var error in result.Errors)
    {
        ModelState.AddModelError("", error.Description);
    }
}
```

4. Update the `LogOn` method's POST action method to validate user credentials and create an authentication cookie. `SignInManager.PasswordSignInAsync` replaces `Membership.ValidateUser` and `FormAuthentication.SetAuthCookie`:

```
[HttpPost]
public async Task<ActionResult> LogOn(LogOnModel model, string returnUrl)
{
    if (ModelState.IsValid)
    {
        var result = await
            signInManager.PasswordSignInAsync(model.UserName,
            model.Password, model.RememberMe, lockoutOnFailure: true);
        if (result.Succeeded)
        {
            MigrateShoppingCart(model.UserName);
            return RedirectToLocal(returnUrl);
        }
        else
        {
            ModelState.AddModelError("", "The user name or password provided
            is incorrect.");
        }
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

5. Update the `LogOff` method to replace `FormsAuthentication.SignOut` with `SignInManager.SignOutAsync` and `HttpContextAccessor`, which allows access to `Session`.

```
public async Task<ActionResult> LogOff()
{
    await signInManager.SignOutAsync();

    httpContextAccessor.HttpContext.Session.Clear();

    return RedirectToAction("Logon", "Account");
}
```

6. Update the Register POST method to create a new user and set the authentication cookie. Note that `UserManager.CreateAsync` replaces `Membership.CreateUser`, and `SignInManager.SignInAsync` replaces `FormsAuthentication.SetAuthCookie`.

```
[HttpPost]
public async Task<ActionResult> Register(RegisterModel model)
{
    if (ModelState.IsValid)
    {
        var user = new User()
        {
            UserName = model.UserName,
            Email = model.Email,
            PasswordQuestion = "question",
            PasswordAnswer = "answer",
            IsApproved = true
        };
        IdentityResult result = await userManager.CreateAsync(user,
            model.Password);

        if (result.Succeeded)
        {
            MigrateShoppingCart(model.UserName);

            await signInManager.SignInAsync(user, isPersistent: true);
            return RedirectToAction("Index", "Home");
        }
        else
        {
            AddErrors(result);
        }
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

7. Update the ChangePassword POST method to update user credentials. Note that `UserManager.FindByNameAsync` replaces `Membership.GetUser` and `UserManager.ChangePasswordAsync` replaces `MembershipUser.ChangePassword`.

```
[Authorize]
[HttpPost]
public async Task<ActionResult>
ChangePassword(ChangePasswordModel model)
{
    if (ModelState.IsValid)
    {
        // ChangePassword will throw an exception rather
        // than return false in certain failure scenarios.
        bool changePasswordSucceeded;
        try
        {
            var user = await
            userManager.FindByNameAsync(User.Identity.Name);
            var result = await userManager.ChangePasswordAsync(user,
            model.OldPassword, model.NewPassword);

            changePasswordSucceeded = result.Succeeded;
        }
        catch (Exception)
        {
            [...]
        }
        [...]
    }
}
```

8. Delete the `ErrorCodeToString` method, as it is no longer needed.
9. In `Views/Account/ChangePassword.cshtml` and `Views/Account/Register.cshtml`, change the reference to `@Membership.MinRequiredPasswordLength` to **6**, as that is the default in ASP.NET Core Identity.

Set up membership crypto

Password criteria and password salts don't migrate between Membership and Identity. After the migration, existing users will not be able to log in with their existing password. To allow existing users to log in without changing their password, you need to migrate the `PasswordSalt` and `PasswordFormat` fields from the Membership database, and create a new implementation of the `IPasswordHasher` interface provided by ASP.NET Identity.

1. The ASP.NET Core Identity API uses the different crypto algorithm than the one used by Membership API. To allow the existing users to log in without changing their password, you need to create a new implementation of the `IPasswordHasher` interface provided by ASP.NET Core Identity. This class allows the existing user to validate their password using the membership crypto algorithm, and new users to verify their password using ASP.NET Core Identity default crypto algorithm. Add the code from [this file](#) to a new class called `SQLPasswordHasher.cs` in the `Models` folder.
2. Add the dependency injection for `IPasswordHasher` in `Startup.cs` in the `ConfigureServices` method:

```
public void ConfigureServices(IServiceCollection services)
{
    [...]

    services.AddScoped<IPasswordHasher<User>,
        SQLPasswordHasher<User>> ();
}
```

Refactor checkout functionality

In this section, you will migrate the checkout functionality. The business logic for checkout is implemented in `CheckoutController.cs`, and the markup is implemented as normal Razor views.

The following components **do not** exist in ASP.NET Core:

- `System.Web` namespace
- `FormCollection`
- `TryUpdateModel`

1. In `Controllers/CheckoutControllers.cs`, add a new constructor with a dependency to `MusicStoreEntites` and `ShoppingCart`:


```
using MvcMusicStore.Models;
using System;
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace MvcMusicStore.Controllers
{
    [Authorize]
    public class CheckoutController : Controller
    {
        const string PromoCode = "FREE";
        private readonly MusicStoreEntities storeDB;
        private readonly ShoppingCart shoppingCart;

        public CheckoutController(MusicStoreEntities musicStoreEntities,
            ShoppingCart shoppingCart)
        {
            storeDB = musicStoreEntities;
            this.shoppingCart = shoppingCart;
        }

        [...]

    }
}
```

2. The `FormCollection` object and `TryUpdateModel` method are replaced by `IFormCollection` and `TryUpdateModelAsync` in ASP.NET Core. Update the code for `AddressAndPayment` method as in the following code example.

```
[HttpPost]
public async Task<ActionResult>
AddressAndPayment(IFormCollection values)
{
    var order = new Order();
    await TryUpdateModelAsync(order);

    try
    {
        if (string.Equals(values["PromoCode"], PromoCode,
            StringComparison.OrdinalIgnoreCase) == false)
        {
            return View(order);
        }
        else
        {
            order.Username = User.Identity.Name;
            order.OrderDate = DateTime.Now;

            //Save Order
            storeDB.Orders.Add(order);
            storeDB.SaveChanges();

            //Process the order
            var cart = shoppingCart;
            cart.CreateOrder(order);

            return RedirectToAction("Complete",
                new { id = order.OrderId });
        }
    }
    catch
    {
        //Invalid - redisplay with errors
        return View(order);
    }
}
```

3. Update `ShoppingCart.GetCart(this.HttpContext)` with the `shoppingCart` object injected in through constructor.

Refactor sample data

The Porting Assistant automatically changed EF to **EntityFrameworkCore**. You must undo those changes in `Models/SampleData.cs` for the project to build:

- In `Models/SampleData.cs`, replace `using Microsoft.EntityFrameworkCore` with `using System.Data.Entity`

Refactor the application to be stateless

To take full advantage of containers, it's important to either make the application stateless or move the state management out-of-process. Stateless applications allow seamless scaling-out and horizontal deployment across multiple servers.

ASP.NET supports different ways to maintain state information on the server-side. One of the popular state management techniques is **Session State**. Session state is a mechanism to store information in the form of a key-value pair that's persisted between multiple web requests. The `MvcMusicStore` application uses server-side sessions to maintain login and shopping cart state.

Before containerizing the application, follow the instructions to use Microsoft SQL Server for storing application session state. These steps migrate the schema in the database to support the added tables. This allows horizontal scaling of the application containers without disrupting existing user sessions.

To use Microsoft SQL Server for storing application session state:

1. Add the required NuGet packages to the project:

```
Install-Package Microsoft.Extensions.Caching.SqlServer
Install-Package
Microsoft.AspNetCore.DataProtection.EntityFrameworkCore
Install-Package Microsoft.Bcl.AsyncInterfaces
Install-Package Microsoft.EntityFrameworkCore.Design
```

2. Add a new class `CacheTable.cs` in the `Models` folder and copy-paste the code from [this file](#). `CacheTable` defines the schema to store session state in SQL Server cache.
3. Modify `Models/ApplicationDbContext.cs` to include `CacheTable` and `DataProtectionKeys DbSet`. This enables the application to use Entity Framework Core to store data protection keys in a database. You can do this by replacing the existing code in `ApplicationDbContext.cs` with code in [this file](#).

4. Modify `Startup.cs` to use `AddDistributedSqlServerCache` in place of `AddDistributedMemoryCache` service, to maintain the state cache as an external service to the application container that accesses it. You can do this by replacing the existing code in `Startup.cs` with the code in [this file](#), and replacing namespace `MvcMusicStoreCore` with namespace `MvcMusicStore`.
5. Code changes to move session state to SQL Server are completed. Run following commands from .NET CLI in the `MvcMusicStore` project directory to apply the database changes:

```
dotnet tool install --global dotnet-ef
dotnet ef migrations add stateless
dotnet ef database update
```

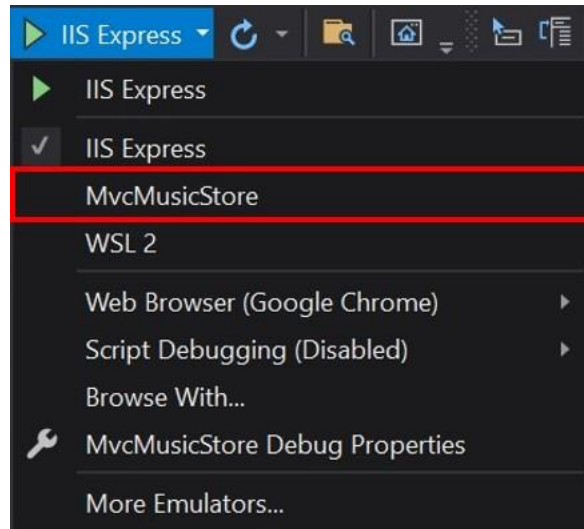
6. Run [this script](#) on your SQL Server instance to migrate the existing data.
7. Update `PasswordHash` in the `dbo.AspNetUser` table with the `PasswordSalt`, `PasswordFormat` and `PasswordHash` fields separated by the `'|'` character from the `dbo.aspnet_Membership` table. Note you may have to run CTRL + SHIFT + R to refresh the SQL Management Studio cache:

```
UPDATE dbo.AspNetUsers
SET
PasswordHash =
(aspnet_Membership.Password+'|'+CAST(aspnet_Membership.PasswordFormat
as varchar)+'|'+aspnet_Membership.PasswordSalt)
FROM dbo.AspNetUsers
LEFT OUTER JOIN dbo.aspnet_Membership
ON AspNetUsers.Id = aspnet_Membership.UserId
WHERE AspNetUsers.PasswordHash = ''
```

Build and run the project

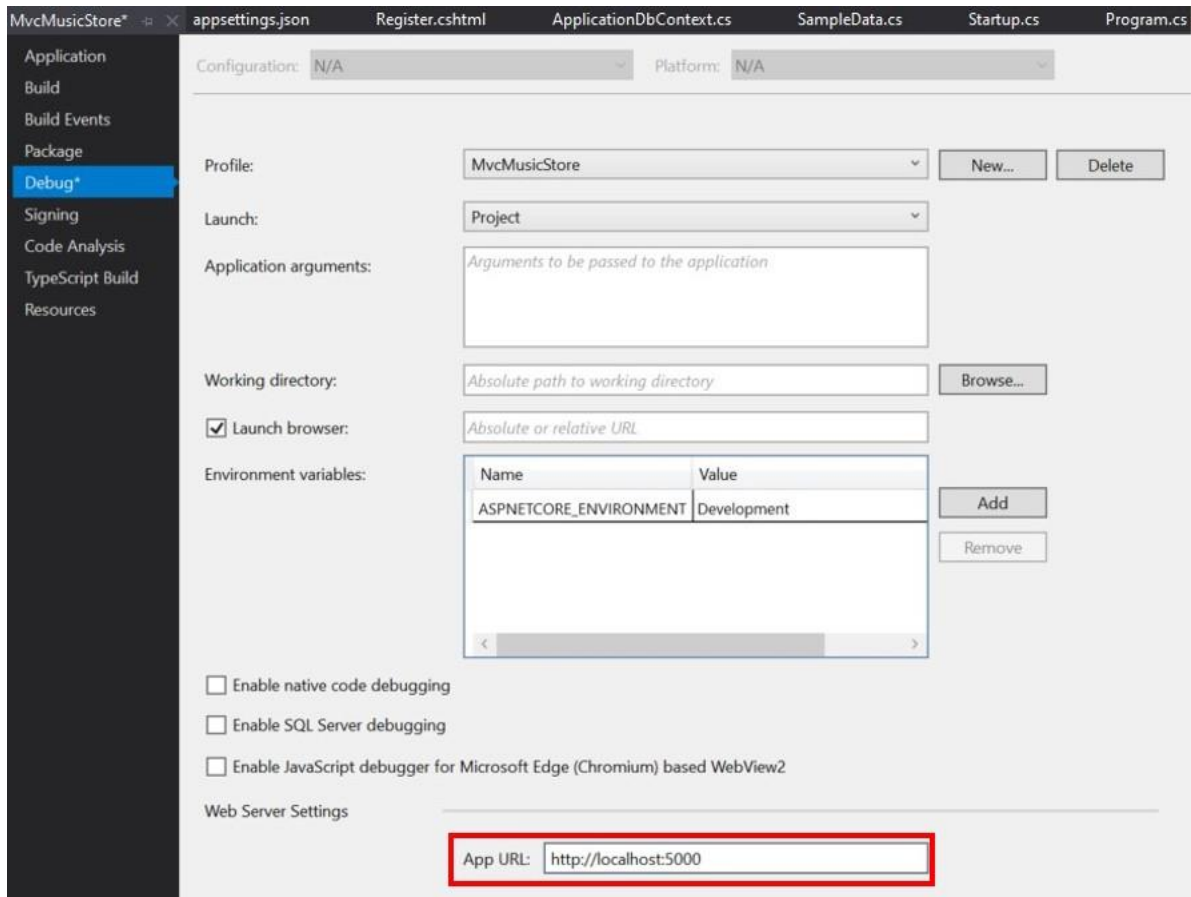
Now you will build and run the project to make sure everything is functioning correctly.

1. In Visual Studio, on the **IIS Express** menu, choose **MvcMusicStore** to enable Console app mode launch.



Visual Studio hosting configuration

2. Remove the SSL debugging endpoint: First, right click on the project, go to Properties, then choose Debug. Then, for App URL change the prefix https:// to http://.



Visual Studio debug properties configuration

3. Click the Debug menu, and click Start Debugging to start the application in debug mode. You have successfully refactored the application from .NET Framework to .NET 5.

Replatforming from Windows VMs to Linux containers

Now that you have successfully refactored the MvcMusicStore application from .NET Framework to .NET 5, it's time to containerize it and deploy it to Amazon ECS backed by AWS Fargate, so you don't have to worry about managing the underlying infrastructure. The starting branch for this step of the walkthrough is `framework-to-core-completed`. The target branch is `core-ecs-cdk-completed`. You can follow along in GitHub. You can also jump directly to the completed branch if you want to deploy and run the application without going through the steps.

For this section of the walkthrough, there are four stages:

1. [Prerequisites](#)

1. [Containerize the application](#)
2. [Create the infrastructure as code](#)
3. [Create the CI/CD pipeline](#)

Prerequisites

- Install Docker. See [Docker Desktop](#).
- Install the AWS CDK. See AWS CDK Toolkit (`cdk` command).
- Install the AWS CLI. See [AWS Command Line Interface](#).

Containerize the application

Before you deploy the application to AWS, you will containerize it locally and ensure that it builds.

1. Create a file with the following content and name it `Dockerfile`. Save it in the same folder as the `MvcMusicStore.csproj` file.

```
FROM mcr.microsoft.com/dotnet/aspnet:5.0 AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /src

COPY ["MvcMusicStore.csproj", ""]
RUN dotnet restore "./MvcMusicStore.csproj"
COPY . .
WORKDIR "/src/"
RUN dotnet build "MvcMusicStore.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "MvcMusicStore.csproj" -c Release -o
/app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "MvcMusicStore.dll"]
```

2. Ensure Docker is started locally, and from the command line, set the working directory to the folder containing the above `Dockerfile`.

3. Run the following command to ensure that the application container image builds successfully.

```
docker build . -t music-store
```

4. Run the `docker images` command from the command line. You should see a newly created `music-store` Docker image in the output:

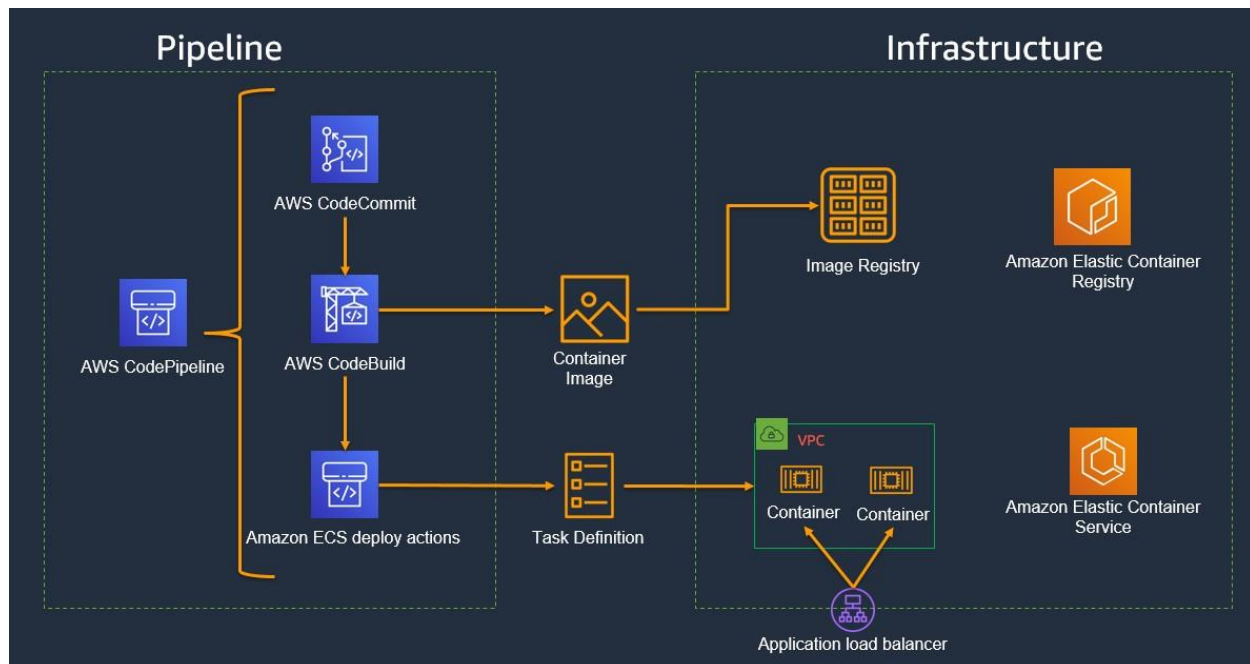
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
music-store	latest	19fa5c6fb0c3	2 minutes ago	233MB

Your application is now containerized and ready for deployment.

Create the infrastructure as code

There are multiple options to run containers on AWS. If you are new to containers on AWS, consider using Amazon ECS and AWS Fargate. There are various ways to create a Fargate service, such as using [AWS Management Console](#), [AWS CLI](#), [AWS CloudFormation template](#), and [AWS Cloud Development Kit](#) (AWS CDK). AWS CDK is an open-source software development framework to define your cloud application resources using familiar programming languages. In the following section of the guide, you will use AWS CDK for .NET to create AWS Cloud infrastructure as code and provision it through CloudFormation.

Review the infrastructure you will build using the AWS CDK:



Common CI/CD deployment

- **Infrastructure** — AWS services used to host the MvcMusicStore application include Amazon VPC, AWS Fargate for Amazon Elastic Container Service (ECS) to run the application Docker containers, and Amazon Elastic Container Registry (ECR) to store and share Docker container images. Amazon Application Load Balancer routes traffic to AWS Fargate.
- **Pipeline** — The CI/CD pipeline, which triggers on code change. It builds a new container image on code push, pushes that image to Amazon ECR, and updates the Fargate service with a newer version of the container image.

In the next section, you create the code to define the pipeline and infrastructure.

Create the CI/CD pipeline

One of the advantages of containerizing your applications is to automate software delivery, reducing the total time-to-market. In the following section you will create a continuous integration and continuous delivery (CI/CD) pipeline. These steps include initiating automatic builds, pushing the container image to Amazon ECR, and updating the Fargate service when there is a code change.

1. Create a new folder called `MusicStoreInfra` in the top-level directory for the project.
2. Move to that directory as the working directory.

3. Run the following command to create a new AWS CDK application in that folder.

```
cdk init app --language csharp
```

4. After running the previous command, you should see a `src` folder in the `MusicStoreInfra` folder. Open the solution in that folder in Visual Studio and install the following packages using the NuGet Package Manager Console:

```
Install-Package Amazon.CDK.AWS.CodeBuild
Install-Package Amazon.CDK.AWS.CodeCommit
Install-Package Amazon.CDK.AWS.CodePipeline
Install-Package Amazon.CDK.AWS.CodePipeline.Actions
Install-Package Amazon.CDK.AWS.IAM
```

5. Rename the `MusicStoreInfraStack.cs` file to `BuildInfraStack.cs` and add the code from [this file](#) to define the CI/CD pipeline as code.
6. Replace the existing code in `Program.cs` with the following code, so you can reference this stack by name when you run the `cdk` command.

```
using Amazon.CDK;

namespace MusicStoreInfra
{
    sealed class Program
    {
        public static void Main(string[] args)
        {
            var app = new App();

            new BuildInfraStack(app, "BuildInfraStack");

            app.Synth();
        }
    }
}
```

7. Deploy the CI/CD pipeline using the AWS CDK. From the `AWS-Pipeline` directory, run the following command to produce or synthesize an AWS CloudFormation template for the stack defined in your application. The output of this command is a CloudFormation template.

```
cdk synth
```

8. Run the following command to bootstrap the stack in your environment. Bootstrap is required the first time you deploy an AWS CDK app in your AWS environment (account and Region). You are now ready to deploy your AWS CodePipeline.

```
cdk bootstrap
```

9. Run the following command to deploy.

```
cdk deploy BuildInfraStack
```

10. Press **y** when prompted for confirmation to proceed. AWS CDK apps are deployed through AWS CloudFormation. You can see the progress of your deployment from the CloudFormation console in the AWS Management Console. This will take several minutes to complete.
11. When the process completes, you should see an output similar to the following:

```
✔ BuildInfraStack  
  
Stack ARN:  
arn:aws:cloudformation:us-east-1:<account-  
id>:stack/BuildInfraStack/98a70620-af66-11eb-b017-0ed5356bca17
```

12. With the pipeline deployed, you are ready to push your application through the automated deployment. Push the MvcMusicStore app code to the CodeCommit repo from the application root folder. If you are using CodeCommit HTTPS for the first time, refer to [Setup for HTTPS users using Git credentials](#) to set up GitHub credentials.
13. Replace `<CLONE_URL>` in the following command with the HTTPS Clone URL available at **AWS Management Console > Developer Tools > CodeCommit > Repositories > music-store**.

```
git init
git add .
git commit -m "first commit"
git remote add codecommit <CLONE_URL>
git push --set-upstream codecommit net48-upgrade-completed:main
```

14. Navigate to the **AWS Management Console > CodePipeline** console. You should see that the `Checkout-Source-Code` and `Build-container-image` stages succeeded and that the `Recycle-ECS-tasks` stage failed. This is expected because you have not yet deployed your infrastructure, which you will do in the next section.

Create the infrastructure

1. Open the `MusicStoreInfra` solution in Visual Studio and install the following packages:

```
Install-Package Amazon.CDK.AWS.ECS
Install-Package Amazon.CDK.AWS.RDS
Install-Package Amazon.CDK.AWS.ECS.Patterns
Install-Package Amazon.CDK.AWS.IAM
Install-Package Amazon.CDK.ECR.Assets
```

2. Create a new class called `HostingInfraStack.cs` and add the code from [this file](#) to `HostingInfraStack.cs`. This creates a new VPC, an empty ECS Cluster, an ECR repository, an ECS task definition, a Fargate instance for ECS service, an RDS SQL Server instance, and a public Application Load Balancer to route user requests to the Fargate containers.
3. Update `Program.cs` with the following code:

```
using Amazon.CDK;

namespace MusicStoreInfra
{
    sealed class Program
    {
        public static void Main(string[] args)
        {
            var app = new App();

            new BuildInfraStack(app, "BuildInfraStack");
        }
    }
}
```

```
        new HostingInfraStack(app, "HostingInfraStack");
        app.Synth();
    }
}
```

4. From the `MusicStoreInfra` directory, run the following command to produce or synthesize an AWS CloudFormation template for the stack defined in your application. The output of this command is a CloudFormation template.

```
cdk synth HostingInfraStack
```

5. Run the following command to bootstrap the stack in your environment. Bootstrap is required the first time you deploy an AWS CDK app in your AWS environment (account and Region).

```
cdk bootstrap
```

6. You are now ready to deploy your AWS Infrastructure. Run the following command to deploy:

```
cdk deploy HostingInfraStack
```

7. Press **y** when prompted for confirmation to proceed. AWS CDK apps are deployed through AWS CloudFormation. You can see the progress of your deployment from the CloudFormation console in the AWS Management Console. This takes several minutes to complete.
8. When the process completes, you should see an output similar to the following:

```
✔ HostingInfraStack

Outputs:
HostingInfraStack.msucstoreecsserviceLoadBalancerDNS87D6E21D =
Music-Store-ALB-862894238.us-east-1.elb.amazonaws.com
HostingInfraStack.msucstoreecsserviceServiceURL99C1C434 =
http://Music-Store-ALB-862894238.us-east-1.elb.amazonaws.com

Stack ARN:
```

```
arn:aws:cloudformation:us-east-1:<account-id>:stack/HostingInfraStack/15239b80-af7d-11eb-b064-1214c28caebf
```

You now have a containerized .NET Core application running on Amazon ECS and Fargate backed by a CI/CD pipeline.

To view the website, choose the **Public Load Balancer** link from the `cdk deploy` step. In this example it is `http://Music-Store-ALB-862894238.us-east-1.elb.amazonaws.com`.

You can see the entire CodePipeline in action by going to **AWS Management Console > Developer Tools > CodePipeline > Pipelines** and triggering a new build. Each time a new commit is pushed to the repository, the new build is automatically deployed to the Amazon ECS Fargate cluster.

Note: Remember to remove your deployed AWS resources by running `cdk destroy` from the directory of the `MusicStoreInfra` AWS CDK application.

Logging and monitoring

Monitoring is an important part of maintaining the reliability, availability, and performance of your applications running on Amazon ECS. You should collect monitoring data from all of the parts of your AWS stack so that you can more easily debug a multi-point failure if one occurs. AWS provides several tools for monitoring your Amazon ECS resources and responding for potential incidents. See the following table for details on each service.

Table 8 — Service details

Service	Description
Amazon CloudWatch Alarms	For clusters with tasks or services using the EC2 launch type, you can use CloudWatch Alarms to scale in and scale out the container instances based on CloudWatch metrics, such as cluster memory reservation.
Amazon CloudWatch Logs	Monitor, store, and access the log files from the containers in your Amazon ECS tasks by specifying the <code>awslogs</code> log driver in your task definitions. This is the only supported method for accessing logs for tasks using the Fargate launch type, but it also works with tasks using the EC2 launch type.
Amazon CloudWatch Events	Match events and route them to one or more target functions or streams to make changes, capture state information, and take corrective action.
AWS CloudTrail	CloudTrail provides a record of actions taken by a user, role, or an AWS service in Amazon ECS. Using the information collected by CloudTrail, you can determine the request that was made to Amazon ECS, the IP address from which the request was made, who made the request, when it was made, and additional details.
AWS Trusted Advisor	Trusted Advisor draws upon best practices learned from serving hundreds of thousands of AWS customers. Trusted Advisor inspects your AWS environment and then makes recommendations when opportunities exist to save money, improve system availability and performance, or help close security gaps.
Amazon ECS events and EventBridge	Use Amazon ECS events for EventBridge to receive near real-time notifications regarding the current state of your Amazon ECS clusters. If your tasks use the Fargate launch type, you can see the state of your tasks. If your tasks use the EC2 launch type, you can see the state of both the container instances and the current state of all tasks running on those container instances. For services, you can see events related to the health of your service.

Service	Description
AWS X-Ray	AWS X-Ray helps developers analyze and debug production distributed applications, such as those built using a microservices architecture. With AWS X-Ray, you can understand how your application and its underlying services are performing to identify and troubleshoot the root cause of performance issues and errors. You can use the AWS X-Ray SDK and AWS service integration to instrument requests to your applications that are running locally or on AWS compute services such as Amazon EC2, AWS Elastic Beanstalk, Amazon ECS, and AWS Lambda.

Security

Security is and will always be the top priority at AWS. We have a shared responsibility model with the customer: AWS manages and controls the components from the host operating system and virtualization layer down to the physical security of the facilities in which the services operate, and AWS customers are responsible for building secure applications.

This section of the guide details the primary areas for you to understand when deploying and running .NET applications in containers on Amazon ECS to meet your security and compliance objectives:

- [User to application authentication and authorization](#)
- [Application to database authentication and authorization](#)
- [Identity and access management for Amazon ECS](#)
- [Compliance validation for Amazon ECS](#)
- [In-flight data protection using encryption](#)

User to application authentication and authorization

During the walkthrough section, you migrated the .NET Framework Membership APIs and Form authentication to ASP.NET Core Identity to manage users, passwords, roles, and the other aspects of login functionality for the application. While the application in this guide stores user management data in SQL Server, Amazon also provides an ASP.NET Core Identity Provider for Amazon Cognito, which allows ASP.NET Core

applications to easily integrate with Amazon Cognito in their web applications for user authentication and authorization. The following integrations are also supported:

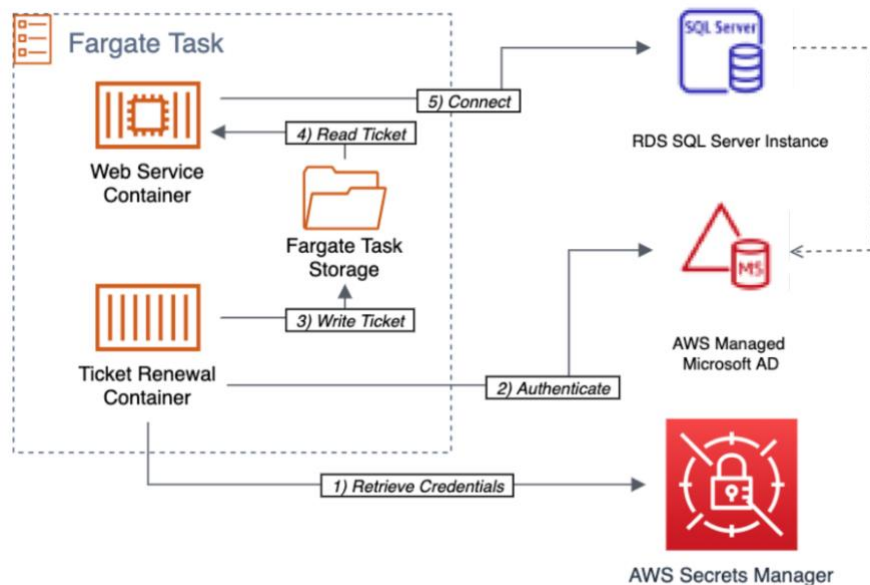
- User authentication through an identity provider (IdP) that is OpenID Connect (OIDC) compliant
- User authentication through well-known IdPs such as Amazon, Facebook, and Google through [Amazon Cognito user pools](#)
- User authentication through corporate identities using SAML, LDAP, or Microsoft Active Directory through Amazon Cognito user pools

Application to database authentication and authorization

Amazon RDS supports both Windows and SQL Server authentication modes when connecting an application to the database.

Windows Authentication

Windows (or Integrated) Authentication is the common mechanism for clients and applications to connect to SQL Server databases. Typically, these applications are joined to the same domain as the SQL Server database, but because individual containers are ephemeral, joining them to a domain is not optimal. The best practice is to use a separate Amazon ECS task as a ticket renewal “sidecar,” which stores the Kerberos ticket in Fargate task storage for the application, which reads the ticket from Fargate task storage and connects to the database using Windows Authentication. For more information on this process, reference [Using Windows Authentication with Linux Containers on Amazon ECS](#), and the following diagram.



Fargate deployment with ticket renewal sidecar

SQL Server Authentication

For .NET applications running in Linux containers on AWS, you can also use SQL Authentication mode. When using SQL Server Authentication, logins are created in SQL Server that are not based on Windows user accounts. Both the user name and the password are created by using SQL Server, and stored in SQL Server.

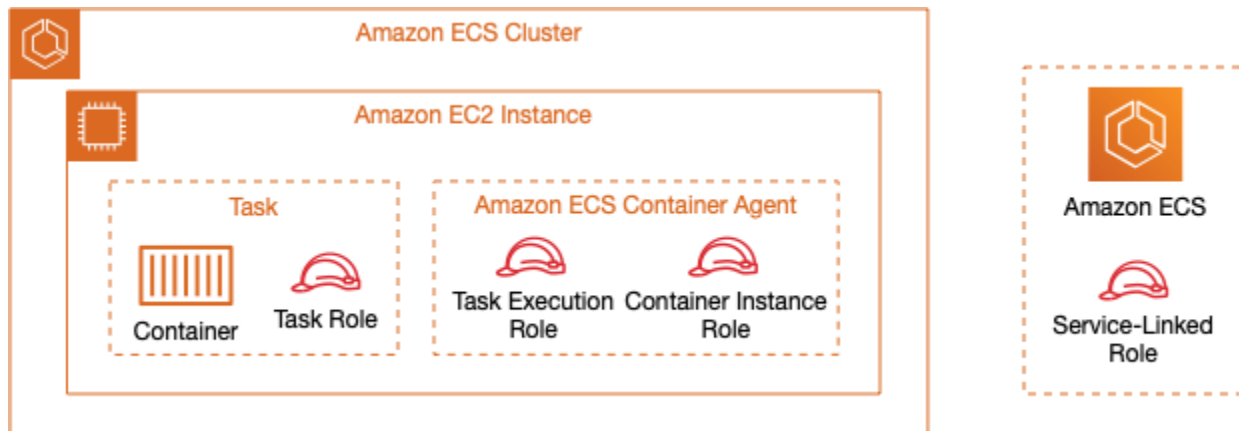
The MvcMusicStore application used in the walkthrough of this guide uses SQL Server authentication, and stores the connection strings as environment variables at the Task-level. While this technique is used for simplicity of the demonstration, it's a best practice to store the password as an AWS Secrets Manager-encrypted secret, and to make it a standalone configuration setting so the passwords can be recycled dynamically.

An example of storing just the password component of the connection string as an AWS Secret Manager encrypted secret can be found in the UnicornStore sample application in the [Fargate Stack definition](#).

If you are considering migrating away from SQL Server as part of your modernization strategy, note that Amazon IAM is the primary mechanism to authenticate applications to Amazon purpose-built database services such as Amazon Aurora and you should plan your authentication and authorization modernization strategy accordingly.

Identity and access management for Amazon ECS

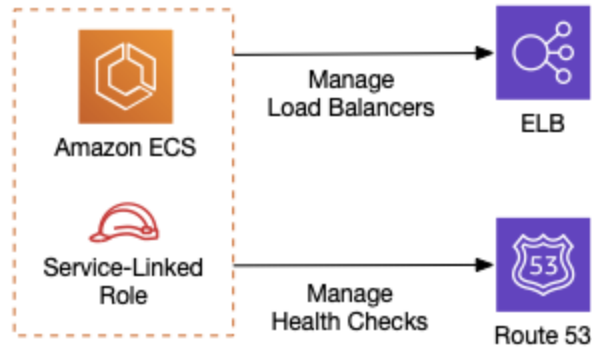
With containerized applications running on Amazon ECS, your application, Amazon ECS, and the Amazon ECS Container Agent will carry out multiple activities which require access to other AWS services and resources. As an administrator, [AWS Identity and Access Management](#) (IAM) is the AWS service to help security control access from the cluster you are responsible for to other AWS services and resources. The following diagram shows the types of roles that Amazon ECS supports. This paper details the purpose of each role in the following sections.



ECS roles conceptual diagram

Service-linked role

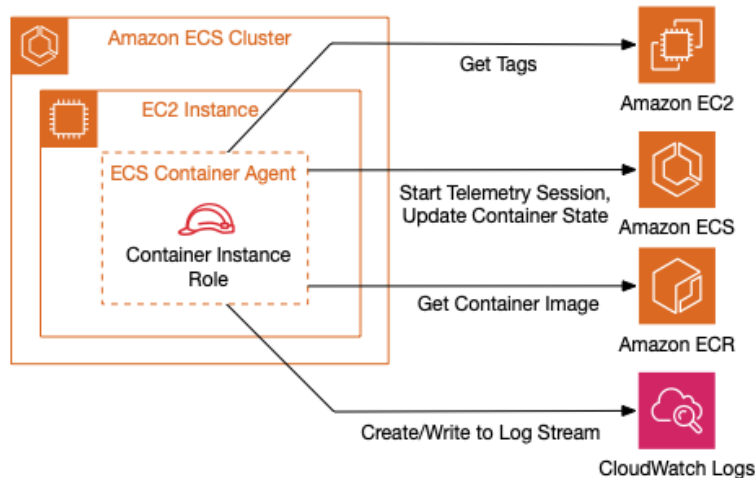
There are multiple activities that the Amazon ECS service runs as it orchestrates your container workloads. Amazon ECS uses a service-linked role for the permissions it requires to call other AWS services on your behalf. These include services like Amazon EC2 to manage elastic network interfaces, ELB to manage targets, and Amazon Route 53 for creating health checks, amongst others. A more detailed list can be found in the [Service-linked role for Amazon ECS](#) page.



Service-linked role conceptual diagram

Container Instance role

The [container instance role](#) is the IAM role used as the Instance role by Amazon EC2 instances running your containers. This role is also leveraged by the Amazon ECS container agent to make calls to AWS services and connect with the Amazon ECS service to register container instances, report status, and get commands. Other examples include the agent starting a telemetry session, or creating the Amazon ECS cluster if one does not already exist. A more detailed list can be found on the [Amazon ECS container instance IAM role](#) page.



Container instance role conceptual diagram

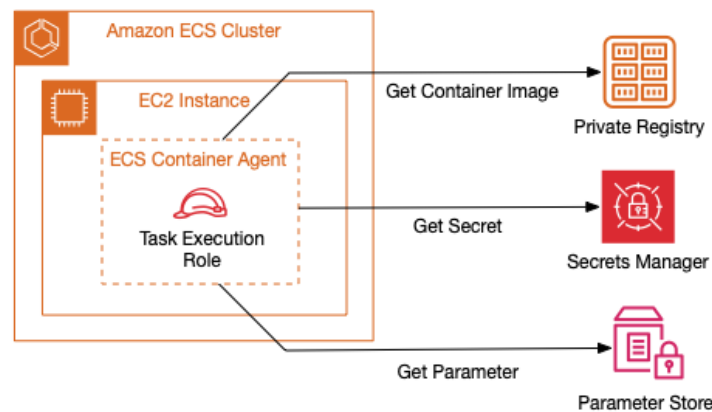
Task execution role

The [task execution role](#) grants the Amazon ECS container agent permission to make AWS API calls on your behalf when an [Amazon ECS task](#) is started. An example of an activity that the Amazon ECS Agent runs during this time is pulling container images

from a private repository, in which case [private registry authentication](#) needs to be configured.

Another use case where this role is required is that of [injecting sensitive data into your containers](#). You may choose to store sensitive data (such as database connection strings) in either [AWS Secrets Manager](#) or [AWS Systems Manager Parameter Store](#), and reference them in your container definition. Sensitive data is injected into your container as environment variables when the container is initially started without having to write code to retrieve the values.

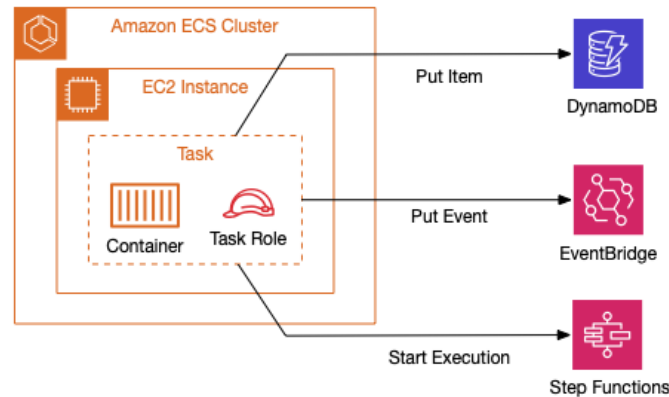
Note that if the secret or Parameter Store parameter is subsequently updated or rotated, the container will not receive the updated value automatically. You must either launch a new task, or if your task is part of a service, you can update the service and use the **Force new deployment** option to force the service to launch a fresh task.



Task execution role conceptual diagram

Task role

[Task role](#) is the IAM role assigned to the containers instances created as part of the Amazon ECS Task. This role provides applications with the AWS credentials they need to make API requests to other AWS services. For example, you can set the policy associated with the Task role to allow your application to read or write items from/to Amazon DynamoDB, publish an event to an Amazon EventBridge bus, or start the running of a AWS Step Functions workflow.



Task role conceptual diagram

Compliance validation for Amazon ECS

Your compliance responsibility when using Amazon ECS is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. Based on your specific requirements, you could apply the respective security controls such as the IAM roles detailed above, encryption in-transit, and encryption at-rest which is supported in Amazon ECR for the container images. Images can also be scanned for vulnerabilities to ensure that there are no security risks in the images themselves.

Image scanning

Amazon ECR image scanning helps identify software vulnerabilities in your container images. Amazon ECR uses the Common Vulnerabilities and Exposures (CVEs) database from the open-source [Clair project](#), and provides a list of scan findings.

Amazon ECR uses the severity for a CVE from the upstream distribution source if available. If not available, ECR uses the Common Vulnerability Scoring System (CVSS) score. The CVSS score can be used to obtain the NVD vulnerability severity rating. For more information, see [NVD Vulnerability Severity Ratings](#).

You can manually scan container images stored in Amazon ECR. Alternatively, you can configure your repositories to scan images when you push them to a repository. For more information on image scanning, see the [Amazon ECR documentation](#).

In-flight data protection using encryption

By default, API calls to the Amazon ECS service travel through the public internet. In order to keep that traffic within the AWS global network, you can configure Amazon

ECS to use an interface VPC endpoint. Interface endpoints are powered by AWS PrivateLink, a technology that enables you to privately access Amazon ECS APIs by using private IP addresses. PrivateLink restricts all network traffic between your VPC and Amazon ECS to the Amazon network. You don't need an internet gateway, a NAT device, or a virtual private gateway.

You can [create VPC endpoints](#) for Amazon ECS, Amazon ECS Container Agent, and Amazon ECS Telemetry in the Region where your containers are deployed. VPC endpoints currently do not support cross-Region requests, so if you have a multi-Region deployment, consider following the recommendations in [Integrating cross VPC ECS cluster for enhanced security with AWS App Mesh](#).

For more information on building a scalable, multi-Region architecture on AWS, see [Building a Scalable and Secure Multi-VPC AWS Network Infrastructure](#). If you are using Amazon ECS integration with Secrets Manager or Systems Manager Parameter Store for sensitive data, you will also need to [configure VPC endpoints for each of these services](#).

In addition to securing network traffic by restricting it to the AWS network, you can encrypt data in transit between your application and AWS services by [enforcing the use of TLS 1.2 when using the AWS SDK for .NET](#). In the following sections, we review the various approaches to using encryption in transit for applications running on Amazon ECS.

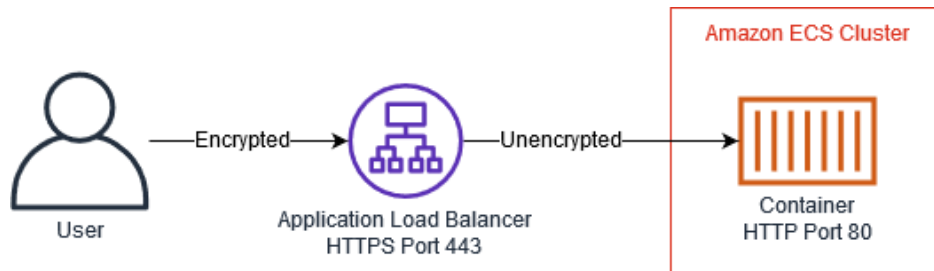
Ending TLS at the load balancer

It's a best practice to enforce ending TLS at the load balancer. Both Application Load Balancers and Network Load Balancers support ending TLS. Ending TLS connections at the Load Balancer frees up your backend containers from the work of encrypting and decrypting your traffic.

Your containers handle plain HTTP requests, while offloading the complexity of managing HTTPS connections to the Load Balancers. This approach also simplifies certificate management, because the certificates are now deployed to the Load Balancers instead of backend containers.

Additionally, you can use AWS Certificate Manager (ACM) at no charge to securely store, expire, rotate, and update your certificates. This process involves adding a TLS listener to your load balancer, configuring the backend container to listen on an unencrypted port such as port 80 (HTTP), and configuring the listener on the load

balancer to forward traffic to the unencrypted port used by your container. See [TLS Termination for Network Load Balancers](#) for more information.



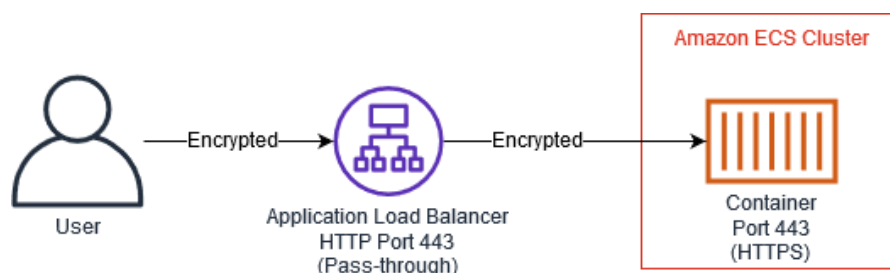
TLS termination at load balancer

End-to-end encryption

Terminating TLS connections at the Load Balancer and using HTTP on the backend may be sufficient for your application. However, if you are developing an application that needs to comply with strict external regulations, you may be required to secure all network connections. You can configure the load balancer to either pass TLS traffic through untouched (end TLS at container), or decrypt and re-encrypt for end-to-end encryption.

Ending TLS at the container-level

This process involves adding an unencrypted listener to your load balancer, configuring backend containers to listen on the secure port and end HTTPS connections, and configuring the listener on the load balancer to forward traffic to the secure port used by the backend containers.

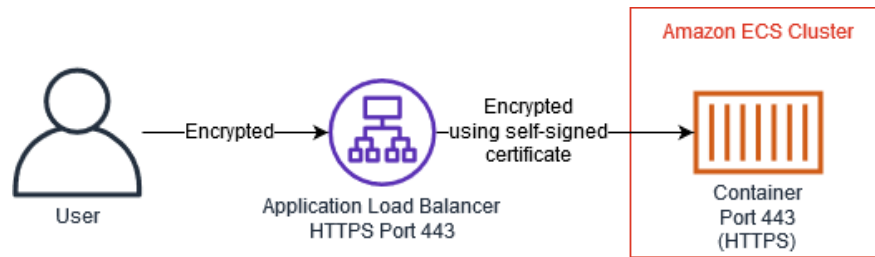


Ending TLS at the container level

Decrypt and re-encrypt

This process involves adding a TLS listener to your load balancer, configuring backend containers to listen on the secure port, ending HTTPS connections, using a self-signed

certificate, and configuring the listener on the load balancer to forward traffic to the secure port used by the backend containers.



Encrypted traffic diagram

Source code

The source code used in this guide is hosted on GitHub at [aws-samples/dotnet-modernization-music-store](https://github.com/aws-samples/dotnet-modernization-music-store). The starting point for the [Refactoring](#) section of the walkthrough is the [net48-upgrade-completed](#) branch and the target is the [framework-to-core-completed](#) branch. The starting point for the Replatforming section of the guide is the [framework-to-core-completed](#) branch, and the target is the [core-ecs-cdk-completed](#) branch.

Conclusion

This guide describes the business and technical aspects of modernizing existing .NET Framework applications to the latest, cross-platform version of .NET and Linux containers. Anyone tasked with evaluating modernization of Windows applications can use this guide to better understand how to approach and run a refactoring and replatforming strategy to accelerate innovation, lower TCO, and increase developer productivity for their organization.

Contributors

Contributors to this document include:

- Daniel Maldonado, Specialist Solutions Architect, Amazon Web Services
- Runeet Vashisht, Specialist Solutions Architect, Amazon Web Services
- Sathish Arumugam, Partner Solutions Architect, Amazon Web Services
- Sanjay Gulati, Sr. Partner Solutions Architect, Amazon Web Services

- Vlad Hrybok, Sr. Partner Solutions Architect, Amazon Web Services
- Chris Splinter, Sr. Product Manager, Amazon Web Services

Document revisions

Date	Description
August 5, 2021	First published

Notes

- ¹ [Developer Economics: State of the Developer Nation 19th Edition](#)
- ² [Gartner Forecasts Strong Revenue Growth for Global Container Management Software and Services Through 2024](#)
- ³ [Container Infrastructure Software Market Assessment: x86 Containers Forecast, 2018–2023](#)
- ⁴ [Guidebook: Containers and Kubernetes on AWS](#)