CHAPTER 1

Understanding Docker

Docker is a set of tools for creating and running application in containers, which isolates the application from all the other software running on a server. Even though the server may be running dozens—or even hundreds—of containers, each application is shielded from all the other instances and operates as though it is the only application running.

This book explains how you can use Docker for ASP.NET Core MVC applications and how containers make it easier to develop, deploy, and manage those applications in production environments.

What Do You Need to Know?

To get the most from this book, you should be familiar with .NET Core and ASP.NET Core MVC development, including how to create, compile, and run projects in Visual Studio or Visual Studio Code. You should also have a basic familiarity with Windows, Linux, or macOS and be able to run commands using a command prompt.

What Is the Structure of This Book?

This book is split into eight chapters. Chapter 2 provides a quick reference for all the Docker commands and features described in this book so you can easily find what you need in the future without having to search through the rest of the book. Chapter 3 contains instructions for getting ready, including installing Docker, Visual Studio, or Visual Studio Code and the supporting tools that are required. Chapter 3 also includes instructions for creating a simple ASP.NET Core MVC project that is used as an example throughout the rest of the book.

Chapters 4–7 explain how to use Docker with ASP.NET Core MVC projects. Chapter 4 introduces images, which are the templates used to create containers, and shows you how to create your own images and use them to produce containers. Chapter 5 covers the Docker features for storing data and connecting the applications in different containers together. Chapter 6 describes the Docker support for describing complex applications that require different types of containers, and Chapter 7 demonstrates the Docker support for clustering servers together to run larger numbers of containers. The final chapter, Chapter 8, comes full circle and explains how the features from the rest of the book can be applied to the development environment instead of production.

Is This an Exhaustive Docker Reference?

No. This book covers the essential Docker features for working with ASP.NET Core MVC applications. I have left out Docker features that have no bearing on MVC applications or that are unlikely to be used by the majority of projects.

A. Freeman, *Essential Docker for ASP.NET Core MVC*, DOI 10.1007/978-1-4842-2778-7_1

Are There Lots of Examples?

There are *loads* of examples, and every chapter shows you how Docker works by demonstrating, not describing, the features. At the end of this book, you will have a solid understanding of what Docker does, how it does it, and why it is useful when developing and deploying ASP.NET Core MVC applications.

You can download the examples for all the chapters in this book from `apress.com`. The download is available without charge and includes the example ASP.NET Core MVC project and the configuration files so that you don't have to create them yourself. You don't have to download the code, but cutting and pasting the code into your own projects is the easiest way of experimenting with the examples.

THIS BOOK AND THE DOCKER RELEASE SCHEDULE

Docker is actively developed, and new releases appear often. For the most part, new releases fix bugs and add new features, but Docker is a fast-moving target, and sometimes there are breaking changes.

It doesn't seem fair or reasonable to ask you to buy a new edition of this book every few months, especially since the majority of Docker features are unlikely to change. Instead, I will post updates following the major releases to the GitHub repository for this book, for which there is a link on `apress.com`.

This is an experiment for me (and for Apress), and I don't yet know what form those updates may take—not least because I don't know what the future Docker will contain—but the goal is to extend the life of this book by supplementing the examples it contains.

I am not making any promises about what the updates will be like, what form they will take, or how long I produce them before folding them into a new edition of this book. Please keep an open mind and check the repository for this book when new Docker versions are released. If you have ideas about how the updates could be improved as the experiment unfolds, then e-mail me at `adam@adam-freeman.com` and let me know.

Which Operating Systems Are Supported?

All of the examples in this book have been tested with all three operating systems: Windows, macOS, and Linux. The exception is Chapter 7 because the Docker clustering features that it describes are supported by Docker on Linux server only.

Why Should You Care About Docker?

Docker helps to solve two important problems that affect any complex project but that are especially prevalent in ASP.NET Core projects: the consistency problem and the responsiveness problem.

What Is the Consistency Problem?

Most ASP.NET Core MVC applications are made up of multiple components. There will be at least one server running the MVC application and usually a database to persistently store data.

Complex applications may require additional components: more application servers to share the work, load balancers to distribute HTTP requests among the application servers, and data caches to improve performance. As the number of components increases, additional servers are needed, as are networks to link everything together, name servers to aid discovery, and storage arrays to provide data resilience.

Few projects can afford to provide each developer with a complete replica of the production systems. As a consequence, developers create an approximation of the production systems, typically running all the components required by an application on a single development workstation and ignoring key infrastructure such as networks and load balancers.

Working with an approximation of the production system can lead to several different difficulties, all of which arise because the platform that the developer is using is not consistent with the production systems into which the application is deployed.

The first difficulty is that differences in the environment can cause the application to behave unexpectedly when it is deployed. A project developed on Windows but deployed onto Linux servers, for example, is susceptible to differences in file systems, storage locations, and countless other features.

The second difficulty is that the approximations developers use to represent the production environment can drift apart. Dependencies on different versions of development tools, NuGet packages, and even versions of the .NET Core and ASP.NET Core runtimes can occur, leading to code that makes assumptions that are not valid in production or on other developers' workstations, which have their own approximation of production.

The third difficulty is performing the actual deployment. The differences between development and production systems require two configurations, one of which is difficult to test until the moment of deployment. I have lost track of the amount of time that I have spent over the years trying to deploy applications only to find that a single character is missing from a configuration setting or that there is a hard-coded assumption that the database can be accessed via `localhost`.

The fourth difficulty is that it can be difficult to ensure that all the servers for an application are configured consistently. A misconfigured server may cause only periodic problems, especially if a user's HTTP requests are distributed to a large group of servers, and identifying the problem and isolating the cause can be a difficult task, made fraught by having to perform diagnostics on a live production system.

How Does Docker Solve the Consistency Problem?

When you put an ASP.NET Core MVC application in a container—a process known as *containerization*—you create an image, which is a template for containers that includes the complete environment in which the application will exist. Everything that will be used to run the application is part of the image: the .NET Core runtime, the ASP.NET Core packages, third-party tools, configuration files, and the custom classes and Razor views that provide the application functionality.

Docker uses the image to create a container, and any container created from the same image will contain an identical instance of the ASP.NET Core MVC application.

If you adopt Docker for the development phase of your project, the developers will all use a single image to create and test the application. The development image is still an approximation of the production system, but it is a more faithful replica and will differ only by including development tools such as a compiler and a debugger. In all other regards, the development image will have the same contents as the image used to deploy the application, with the same file system, network topology, NuGet packages, and .NET runtime.

A production image is created when the application is ready to be deployed. This image is similar to the one used by the developers but omits the development tools and contains compiled versions of the C# classes. The production image is used to create all of the containers in production, which ensures that all the instances are configured consistently. And, since the development and production images contain the same content, there is no need to change configuration files in production because the database connection strings that worked in development, for example, will work in production without modification.

What Is the Responsiveness Problem?

Traditional methods for deploying ASP.NET Core MVC applications make it hard to respond to changes in workload. The approach of deploying an application to Internet Information Services (IIS) running on Windows Server means that adding capacity is a substantial task, requiring additional hardware and configuration changes to add servers to the environment.

The overhead required to increase capacity makes it difficult to scale up an application to short-term surges in demand, and the process of decommissioning capacity makes it difficult to scale down an application once peak demand has passed. The result is that ASP.NET applications have historically struggled to provision just the right amount of capacity to deal with their workload, either suffering from too little capacity at peak times (which affects user experience) or too much capacity outside of the peaks (which drives up the cost of running the application and ties up capacity that could be used for other services).

How Does Docker Solve the Responsiveness Problem?

Containers are lightweight wrappers around an application, providing just enough resources for the application to run while ensuring isolation from other containers. Depending on the application, a single server can run many containers, and Docker provides integrated clustering, known as a *swarm*, that allows containers to be deployed without any special awareness of the cluster or configuration changes. The combination of the low resource demands and the integrated clustering means that scaling a containerized ASP.NET Core MVC application is just a matter of adding or removing containers. And, since containers isolate applications, any unused capacity can be repurposed to running containers from another application, allowing workloads to be rebalanced dynamically.

Aren't Docker Containers Just Virtual Machines?

At first glance, containers seem a lot like virtual machines, and there are similarities in the way that containers and virtual machines are used, even if they work in different ways. Both can be used to scale applications by adding or removing instances, and both can be used to create standardized environments for running applications.

But containers are not virtual machines. A virtual machine provides a completely isolated software stack, including the operating system. A single server, for example, can be used to run a mix of virtual machines, each of which can be a different operating system, allowing applications that require Linux and Windows to run side by side in different virtual machines.

Docker only isolates a single application, and all of the containers on a server run on the server's operating system. This means that all the applications run in Linux containers on a Linux server and in Windows containers on a Windows server.

Because Docker containers only isolate applications, they require fewer resources than a virtual machine, which means that a single server can run more containers than it can virtual machines. This doesn't automatically mean that a server running containers can handle more work overall, but it does mean that there are fewer resources spent handling lower-level operating system tasks, which are duplicated in each virtual machine. Figure 1-1 shows the difference between ASP.NET Core MVC applications running in Docker containers and virtual machines.

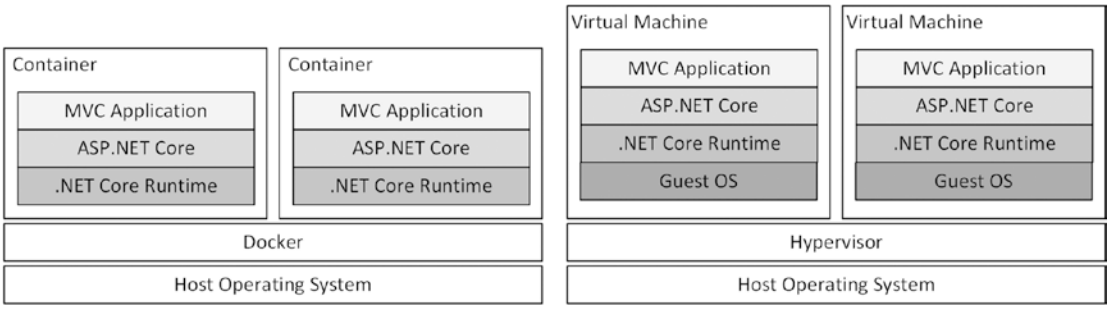

Figure 1-1. *Containers versus virtual machines*

The figure shows the classic comparison between using Docker and virtual machines, but the key difference for ASP.NET Core MVC projects is that Docker provides features that make it easy to create duplicate containers that run alongside one another without any additional configuration, automatically sharing in the application workload by accepting HTTP requests as part of a cluster of containers. This, more than any other difference, is what makes Docker useful because it solves the consistency and responsiveness problems in an elegant way, which is difficult to achieve using traditional virtual machines.

■ **Note** The comparison in Figure 1-1 shows the difference between containers and virtual machines in a production environment, where containers are run on the host server's operating system. Linux and Windows Server can be used in production. To allow Linux Docker images to be created using macOS and Windows, Docker installs a Linux virtual machine that is used to run containers. The installation and configuration of this virtual machine is done automatically during the installation process in Chapter 3.

Do Docker Containers Have Limitations?

Docker containers are not suited to every project. Containers work best for MVC applications that are stateless, such that a series of HTTP requests from a single client can be handled by more than one instance of the application, running in different containers. This doesn't mean the MVC application cannot have any state data, but it does mean that the state data needs to be stored so it can be accessed from any container, such as by using a database. (I describe how to create an MVC application that accesses a containerized database in Chapter 4.)

Part of the benefit conferred by using Docker is the ability to create and destroy containers as needed, which means that MVC applications that have complex initialization routines, that require manual intervention, or that can run only a single instance are not suitable for containerization.

Docker relies on the containerization support included in server operating systems. The Linux support for containers is mature and reliable and is supported by all of the major distributions. At the time of writing, however, containers are a new addition to Windows Server and are not as mature or as well-supported as their Linux counterparts. Windows containers are available only on Windows Server 2016 and can be developed only using Windows 10 with pre-release versions of Docker. Not all the public cloud platforms provide support for Windows containers, which can restrict your deployment choices.

The good news is that .NET Core and ASP.NET Core work well on Linux, which means you can take advantage of Docker on Linux servers, including those provided by public clouds such as Amazon Web Services and Microsoft Azure. Your options are limited if your MVC application depends on the traditional (non-Core) .NET Framework running on Windows. I explain how to create Windows containers using Docker in Chapter 4, but for the moment at least, careful consideration should be given to moving the project to .NET Core and ASP.NET Core so that it can run in Linux containers, othersiemore conventional deployment methods, such as virtual machines or hosted IIS servers, should be used.

Do You Have to Use Docker to Use Containers?

No. Docker is just one set of tools that work with the container features provided by Linux and Windows. Support for containers has been part of Linux for a long time and has matured into a stable and reliable feature. Microsoft has also embraced containers and has included support for them in Windows Server 2016, although it is not as widely used or as well supported as its Linux counterpart.

Docker has become popular because it makes container functionality easy to use, providing tools that create and manage the images from which containers are created and that cluster those containers together to easily scale applications.

Docker may not suit every need, and since the container support is built into the operating system, there are alternatives to Docker that might be better suited to a specific project. A standardization effort called the Open Container Initiative (`https://www.opencontainers.org`) aims to standardize the use of containers, which should also make it easier to mix tools and runtimes from other providers. Docker is participating in the standards process, so you should be able to build on the features described in this book even if you use tools or features from other providers.

At the time of writing, the main competitor to Docker is `rkt`, which is produced by a company called CoreOS and which you can learn about at `https://coreos.com/rkt`. CoreOS is best known for its lightweight CoreOS Container Linux distribution, which is an excellent server for running containers, including those from Docker. I use CoreOS in Chapter 7 when I demonstrate how to create clusters of servers running containers. CoreOS can also be used on most of the public cloud services that support Docker containers, including Amazon Web Services and Microsoft Azure. See `https://coreos.com/why` for details of CoreOS Container Linux.

How Do You Set Up Your Development Environment?

Chapter 3 provides detailed setup instructions for Windows, Linux, and macOS.

Contacting the Author

If you have problems making the examples in this book work or if you find a problem in the book, then you can e-mail me at `adam@adam-freeman.com` and I will try my best to help.

Summary

In this chapter, I described the purpose and content of this book, explained how you can download the project used for each chapter of the book, and explained how Docker solves the consistency and responsiveness problems that face ASP.NET Core MVC applications. In the next chapter, I provide a quick reference to using Docker before showing you how to set up your environment for working with Docker and ASP.NET Core in Chapter 3.