CHAPTER 5

Docker Volumes and Networks

In this chapter, I describe two Docker features that are designed to deal with more complex applications and, specifically, applications that are made up of multiple containers.

The first feature, called *volumes*, separates the data files that are generated by an application or a database from the rest of the container's storage, which makes it easier to replace or upgrade a container. The second feature, known as *software-defined networks*, allows containers to communicate, which makes it easier to scale an application to handle larger workloads. Table 5-1 puts volumes and software-defined networks in context.

Table 5-1. *Putting Docker Volumes and Software-Defined Networks in Context*

Question	Answer
What are they?	Volumes allow important data to exist outside of the container, which means you can replace a container without losing the data that it created.
	Software-defined networks are Internet Protocol networks created by Docker that allow the applications in containers to communicate.
Why are they useful?	Volumes make it possible to delete a container without also deleting the data it contains, which allows containers to be changed or upgraded without losing user data.
	Software-defined networks make it possible to create more complex applications that span multiple containers, making it easier to introduce common components like databases.
How are they used?	These features are managed through the `docker volume` and `docker network` commands. Volumes and software-defined networks must be prepared before the containers that use them are created.
Are there any pitfalls or limitations?	Working out which volumes are required by base images can be a difficult process.
	Software-defined networks only connect containers on a single server unless a Docker cluster is used (as described in Chapter 7).
Are there any alternatives?	There are no alternatives to these features, but you require them for every project. Volumes are not required if a containerized application doesn't generate data that you need to save when the container is removed. Software-defined networks are not required if your containers do not need to communicate.

Table 5-2 summarizes the chapter.

Table 5-2. *Chapter Summary*

Problem	Solution	Listing
Ensure that data files are preserved when a container is deleted	Create a volume and use it to provide the contents of a directory in the container's file system	1–11, 13–17
Determine whether a container uses a volume	Use the `docker inspect` command	12
Add a database to an ASP.NET Core MVC application	Create and apply a volume to the database container and configure Entity Framework Core to connect to the database in the container	18–29
Connect containers together	Create a software-defined network and connect containers to it	30–34
Connect a container to a software-defined network	Use the `--network` argument when creating the container or use the `docker network connect` command	35–39
Distribute work between containers connected to a software-defined network	Use a load balancer that directs requests to containers using the Docker DNS feature	40–41

Preparing for This Chapter

This chapter depends on the ExampleApp MVC project created in Chapter 3. If you don't want to work through the process of creating the example, you can get the project as part of the free source code download for which there is a link on the `apress.com` page for this book.

If you are a Windows user and you followed the examples in the previous chapter to create Windows containers, then you must return to working with Linux containers. Right-click the Docker icon in the task bar and select Switch to Linux Containers from the pop-up menu.

To ensure that there is no conflict with earlier examples, run the command shown in Listing 5-1 to remove the containers created in the previous chapter.

Listing 5-1. Removing the Containers

```
docker rm -f $(docker ps -aq)
```

Working with Volumes

There are two kinds of file associated with an application: the files required to run the application and the data files that application generates as it runs, which are typically produced as a result of user actions. In the world of Docker, these two types of files are handled differently.

The files required to run the application are part of the Docker container for an application. When Docker processes the instructions in a Docker file, it builds up the image that forms the template for containers. For an ASP.NET Core MVC application, this means that containers include the .NET Core runtime, the ASP.NET Core packages, the custom C# classes, the Bootstrap CSS stylesheet, the Razor view, and all of the configuration files. Without these files, a containerized MVC application would not be able to run.

Data files are not included in containers. One of the key benefits of using containers is that they are easy to create and destroy. When a container is destroyed, the files in its file system are deleted as well, which would be disastrous for data files because they would be lost forever.

Docker provides a feature called *volumes* to manage application data, and in the following sections, I explain how volumes work, demonstrate the tools that are available for working with them, and show you a common type of application that uses volumes: a database.

Demonstrating the Problem

Volumes can be confusing, and the best place to start is to demonstrate what happens when they are not used. Create a file called `Dockerfile.volumes` in the `ExampleApp` folder and add the commands shown in Listing 5-2.

Listing 5-2. The Contents of the Dockerfile.volumes File in the ExampleApp Folder

```
FROM alpine:3.4

WORKDIR /data

ENTRYPOINT (test -e message.txt && echo "File Exists" \
        || (echo "Creating File..." \
       && echo Hello, Docker $(date '+%X') > message.txt)) && cat message.txt
```

This Docker file uses the minimal Alpine Linux distribution as its base. To simulate an application that generates data, the `ENTRYPOINT` command creates a data file called the `/data/message.txt` file that contains a message and a timestamp.

The data file isn't created until the container is started and won't be part of the image that is created from the Docker file, similar to the content of a database in a real application.

Run the commands in Listing 5-3 from the `ExampleApp` folder to build an image from the Docker file and use that image to create and start a new container.

Listing 5-3. Creating an Image and a Container

```
docker build . -t apress/vtest -f Dockerfile.volumes
docker run --name vtest apress/vtest
```

The container will produce the following output when Docker starts it, although you will see a different timestamp:

```
...
Creating File...
Hello, Docker 20:21:50
...
```

The container exits once it has written out the message, which shows that the `/data/message.txt` data file has been created and that it has been timestamped at `20:21:50`.

Because I have not set up a volume for the data file, it has become part of the container's file system. The file system is persistent, which you can see by running the command in Listing 5-4 to start the same container again.

Listing 5-4. Restarting the Container

```
docker start -a vtest
```

This time you will see output like this:

```
...
File Exists
Hello, Docker 20:21:50
...
```

The output indicates that the `/data/message.txt` file already exists and has the same timestamp.

The problems with data files occur when a container is deleted. As you will learn in later chapters, Docker containers are created and destroyed often, either to reflect a change in the workload or to deploy a new version of the application. Run the command shown in Listing 5-5 to remove the container.

Listing 5-5. Deleting a Container

```
docker rm -f vtest
```

Docker will delete the container, and the `/data/message.txt` file is lost. To confirm that this is the case, run the command shown in Listing 5-6 to create and run another container from the same image.

Listing 5-6. Replacing the Container

```
docker run --name vtest apress/vtest
```

The output from the container shows that a new data file has been created.

```
...
Creating File...
Hello, Docker 20:53:26
...
```

To state the obvious, deleting data files in a real application has serious consequences and should be avoided.

Managing Data with a Docker Volume

Docker volumes solve the data file problem by keeping data files outside the container while still making them accessible to the application that runs inside it. There are three steps to using a volume, which I describe in the sections that follow.

Step 1: Updating the Docker File

The first step to applying a volume is to add a command to the Docker file, as shown in Listing 5-7.

Listing 5-7. Declaring a Volume in the Dockerfile.volumes File in the ExampleApp Folder

```
FROM alpine:3.4

VOLUME /data

WORKDIR /data
```

```
ENTRYPOINT (test -e message.txt && echo "File Exists" \
        || (echo "Creating File..." \
       && echo Hello, Docker $(date '+%X') > message.txt)) && cat message.txt
```

The `VOLUME` command tells Docker that any files stored in `/data` should be stored in a volume, putting them outside the regular container file system. The important point to note is that the application running in the container won't know that the files in the `/data` directory are special: they will be read and written just like any other file in the container's file system.

Save the changes to the Docker file and run the command shown in Listing 5-8 in the `ExampleApp` folder to re-create the image.

Listing 5-8. Updating the Image to Use a Volume

```
docker build . -t apress/vtest -f Dockerfile.volumes
```

Step 2: Creating the Volume

The second step is to create the volume that will hold the data files. Run the command shown in Listing 5-9 to create the volume that will be used to store the data files for the example application.

Listing 5-9. Creating a Volume

```
docker volume create --name testdata
```

The `docker volume create` command is used to create a new volume and assign it a name, which is `testdata` in this case. The volume is a self-contained file system that will provide the contents for one directory in the container's file system. Since the volume isn't part of the container, the files it contains are not deleted when the container is destroyed.

Step 3: Creating the Container

The third and final step is to tell Docker which volume should be used by the container. Run the command shown in Listing 5-10 in the `ExampleApp` folder to create a container that uses the `testdata` volume created in Listing 5-9 to provide the contents for the `/data` directory configured in Listing 5-7.

Listing 5-10. Associating a Volume with a Container

```
docker run --name vtest2 -v testdata:/data apress/vtest
```

The `-v` argument tells Docker that any data the container creates in the `/data` directory should be stored in the `testdata` volume. The volume appears like a regular directory to the application, which creates the data file like it did before. The output from the command in Listing 5-10 will indicate that a data file has been created, like this:

```
...
Creating File...
Hello, Docker 21:48:20
...
```

Nothing has changed as far as the application is concerned. The simple script I set up in the Docker file's `ENTRYPOINT` command checks to see whether there is a `message.txt` file in the `/data` directory. Volumes are empty when they are first created, which means the application didn't find the file and created it.

The effect of using the volume is apparent only when the container is destroyed and replaced. Run the commands shown in Listing 5-11 to remove the existing container and create and run its replacement.

Listing 5-11. Testing the Volume

```
docker rm -f vtest2
docker run --name vtest2 -v testdata:/data apress/vtest
```

Once again, the `-v` argument is used with the `docker run` command to tell Docker to use the `testdata` volume to provide the contents for the `/data` directory. This time, when the `ENTRYPOINT` script looks for the `/data/message.txt` file, it discovers the one created by the previous container, which has survived because the volume was unaffected when the container was destroyed. You will see the output that shows that the file didn't have to be created, similar to this:

```
...
File Exists
Hello, Docker 21:48:20
...
```

Determining Whether an Image Uses Volumes

There are two ways to check to see whether a Docker image relies on volumes. The first—and most obvious—is to look at the Docker file that was used to create the image. Many publicly available images on Docker include a link to a GitHub repository, where you can easily inspect the Docker file and see whether it contains any `VOLUME` commands. Remember to consider the base image as well when reading Docker files.

The other approach is to examine an image directly, which is useful if you don't have access to the Docker file. Run the command in Listing 5-12 to examine the image used in the previous section.

Listing 5-12. Examining a Docker Image

```
docker inspect apress/vtest
```

The response from the `docker inspect` command is a JSON description of the image, which includes information about the volumes that are used. For the example image, the response from the `docker inspect` command will include the following:

```
...
"Volumes": {
    "/data": {}
},
...
```

The `Volumes` section of the description lists the volumes that the images uses and that can be configured using the `-v` argument to the `docker run` or `docker create` commands.

Adding a Database to the Example Application

Most ASP.NET Core MVC applications rely on a database, which means there are database files that should be stored in a volume so they are not deleted when the database container is destroyed.

However, since the contents of a volume are not included in images, even when the `docker commit` command is used, some special measures are required to ensure that the application's data model schema is created and any seed data applied when the database is first started.

In the sections that follow, I explain how to add a database to the example application, including the steps required to use a volume to contain the data files.

There are lots of database servers available, but the most common choice for containerized ASP.NET Core applications tends to be MySQL, which runs well in Linux containers and has good-quality support in Entity Framework Core (EF Core), which is the object/relational mapping framework used in most ASP.NET Core applications.

To make sure that previous examples don't interfere with the containers in this chapter, run the command shown in Listing 5-13 to remove all the Docker containers.

Listing 5-13. Removing the Existing Docker Containers

```
docker rm -f $(docker ps -aq)
```

SELECTING A DATABASE SERVER

The most common database server choice for ASP.NET Core MVC when Docker isn't being used is Microsoft SQL Server. At the time of writing, the use of SQL Server in Docker is at an early stage. SQL Server runs well in Windows, but Docker support for Windows Containers is still new and has some rough edges. Microsoft has released a preview of SQL Server running on Linux, which will be ideally suited for Docker Linux containers when it stabilizes, but, at least as I write this, the Linux version is not ready for production use. When Microsoft releases the final version, I will post an updated example to this book's source code repository, for which you can find a link on the `apress.com` page for this book.

Pulling and Inspecting the Database Image

When adding a new component to a containerized application, it is important to start by inspecting the image to see how it uses volumes so that you know how to configure the containers that you create. If you skip this step, everything will be fine until you remove a container and find that the data files it contains are removed as well.

Run the commands shown in Listing 5-14 to pull the image for MySQL from the Docker Hub and inspect it.

Listing 5-14. Pulling and Inspecting the MySQL Docker Hub Image

```
docker pull mysql:8.0.0
docker inspect mysql:8.0.0
```

Examine the output from the `docker inspect` command and locate the `Volumes` section, which will look like this:

```
...
"Volumes": {
    "/var/lib/mysql": {}
},
...
```

This tells you that the `mysql:8.0.0` image uses a volume for its `/var/lib/mysql` directory, which is where MySQL stores its data files.

■ **Tip** Don't be tempted to create a Docker image that contains your ASP.NET Core MVC application and the database so they can run in a single container. The convention for Docker is to use a separate container for each component in an application, which makes it easier to upgrade or replace parts of the application and allows a more flexible approach to scaling up the application once it is deployed. You won't benefit from Docker's most useful features if you create a monolithic container that includes all your application components.

Creating the Volume and Container

To prepare for the database container, run the command shown in Listing 5-15 to create the Docker volume called `productdata` that will be used to store the database data files.

Listing 5-15. Creating a Volume for the Database Container

```
docker volume create --name productdata
```

Run the command shown in Listing 5-16 to create and start a new MySQL container that uses the volume to provide the container with the contents of the `/var/lib/mysql` directory. (Enter the command on a single line.)

Listing 5-16. Creating and Starting a MySQL Container

```
docker run -d --name mysql -v productdata:/var/lib/mysql
    -e MYSQL_ROOT_PASSWORD=mysecret -e bind-address=0.0.0.0 mysql:8.0.0
```

As explained in Chapter 4, the `docker run` command creates a container and starts it in a single step. Table 5-3 shows the arguments used to configure the MySQL container.

Table 5-3. *The Arguments Used to Configure the MySQL Container*

Name	Description
-d	This argument tells Docker to run the container in the background without attaching to it to display the output.
`--name`	This argument is used to assign the name `mysql` to the container, which makes it easier to refer to in other Docker commands.
`-e MYSQL_ROOT_PASSWORD`	This argument sets an environment variable. In this case, the MySQL container uses the `MYSQL_ROOT_PASSWORD` environment variable to set the password required to connect to the database. I have set the password to `mysecret` for the example application, but you should use a more secure password for real projects.
`-e bind-address`	This argument sets an environment variable. This environment variable ensures that MySQL accepts requests on all network interfaces.
`-v productdata:/var/lib/mysql`	This argument tells Docker to use a volume called `productdata` to provide the contents of the container's `/var/lib/mysql` directory.

Run the command shown in Listing 5-17 to monitor the database startup.

Listing 5-17. Monitoring the Database Startup

```
docker logs -f mysql
```

It will take a while for MySql to initialize, during which time it will write out log messages. Many of these messages will be warnings that can be ignored. The messages will stop once the database is ready, and one of the last messages shown will be like the following, which indicates that the database is ready to accept network connections:

```
...
08:42:00.729460Z 0 [Note] mysqld: ready for connections.
...
```

Subsequent initializations will be much faster because they will be able to use the files that have been created in the productdata volume. Once the database is running, type Control+C to stop monitoring the output and leave the database running in its container in the background.

Preparing the Example Application

Data for an ASP.NET Core MVC application is typically provided by the Entity Framework Core (known as EF Core), which is the ASP.NET object/relational mapping framework. Edit the ExampleApp.csproj file to add the packages shown in Listing 5-18.

Listing 5-18. Adding Packages in the ExampleApp.csproj File in the ExampleApp Folder

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp1.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore" Version="1.1.1" />
    <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="1.1.2" />
    <PackageReference Include="Microsoft.AspNetCore.StaticFiles" Version="1.1.1" />
    <PackageReference Include="Microsoft.Extensions.Logging.Debug" Version="1.1.1" />
    <PackageReference Include="Microsoft.VisualStudio.Web.BrowserLink"
         Version="1.1.0" />

    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="1.1.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
         Version="1.1.0" />
    <PackageReference Include="Pomelo.EntityFrameworkCore.MySql" Version="1.1.0" />
     <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet"
         Version="1.0.0" />
  </ItemGroup>
</Project>
```

The new packages add the EF Core packages from Microsoft and the Pomelo.EntityFrameworkCore.MySql package, which contains a database provider for MySql. There is an official database provider available from the MySql project, but I find that the Pomelo provider used in this chapter gets updated more rapidly and is easier to work with.

Save the changes to the ExampleApp.csproj file and run the command shown in Listing 5-19 in the ExampleApp folder to download the new packages.

Listing 5-19. Updating the Example Project Packages

```
dotnet restore
```

Creating the Repository Class

At the moment, the example application has dummy data that is provided by a placeholder implementation of the repository interface, which I put in place in Chapter 3 just to get the application started.

Replacing the dummy data with data accessed through EF Core requires a database context class. Add a file called ProductDbContext.cs to the ExampleApp/Models folder and add the code shown in Listing 5-20.

Listing 5-20. The Contents of the ProductDbContext.cs File in the ExampleApp/Models Folder

```
using Microsoft.EntityFrameworkCore;

namespace ExampleApp.Models {

    public class ProductDbContext : DbContext {

        public ProductDbContext(DbContextOptions<ProductDbContext> options)
            : base(options) {
        }

        public DbSet<Product> Products { get; set; }
    }
}
```

The ProductDbContext class will provide access to the Product objects in the database through its Products property. To provide access to the data elsewhere in the application, add a file called ProductRepository.cs to the ExampleApp/Models folder and add the code shown in Listing 5-21.

Listing 5-21. The Contents of the ProductRepository.cs File in the ExampleApp/Models Folder

```
using System.Linq;

namespace ExampleApp.Models {

    public class ProductRepository : IRepository {
        private ProductDbContext context;

        public ProductRepository(ProductDbContext ctx) {
            context = ctx;
        }

        public IQueryable<Product> Products => context.Products;
    }
}
```

This is a simple repository that exposes the Product objects in the database through a Products property. In a real project, the repository would also provide methods to create and modify objects, but I have omitted these to keep the focus on Docker.

To define seed data that will be added to the application when the database is empty, create a file called SeedData.cs in the ExampleApp/Models folder and add the code shown in Listing 5-22.

Listing 5-22. The Contents of the SeedData.cs File in the ExampleApp/Models Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using System.Linq;

namespace ExampleApp.Models {

    public static class SeedData {

        public static void EnsurePopulated(IApplicationBuilder app) {
            EnsurePopulated(
                app.ApplicationServices.GetRequiredService<ProductDbContext>());
        }

        public static void EnsurePopulated(ProductDbContext context) {
            System.Console.WriteLine("Applying Migrations...");
            context.Database.Migrate();

            if (!context.Products.Any()) {

                System.Console.WriteLine("Creating Seed Data...");
                context.Products.AddRange(
                    new Product("Kayak", "Watersports", 275),
                    new Product("Lifejacket", "Watersports", 48.95m),
                    new Product("Soccer Ball", "Soccer", 19.50m),
                    new Product("Corner Flags", "Soccer", 34.95m),
                    new Product("Stadium", "Soccer", 79500),
                    new Product("Thinking Cap", "Chess", 16),
                    new Product("Unsteady Chair", "Chess", 29.95m),
                    new Product("Human Chess Board", "Chess", 75),
                    new Product("Bling-Bling King", "Chess", 1200)
                );
                context.SaveChanges();
            } else {
                System.Console.WriteLine("Seed Data Not Required...");
            }
        }
    }
}
```

The static EnsurePopulated method creates a database context object and uses it to add Product objects to the database. The most important statement in the SeedData class is this one:

```
...
context.Database.Migrate();
...
```

EF Core manages the schema of the database using a feature called *migrations*, which are usually applied to the database through a command-line tool. This doesn't work when using Docker because it is difficult to perform manual configuration steps when deploying an application. Instead, the `Database.Migrate` method is called during application startup to apply any pending migrations to the database. This ensures that the database schema is created without needing any command-line intervention, but, as you will learn, different techniques are required for production applications, as described in later chapters.

Configuring the Application

To configure the application and enable the EF Core services, make the modifications shown in Listing 5-23 to the `Startup` class.

Listing 5-23. Configuring the Application in the Startup.cs File in the ExampleApp Folder

```
using ExampleApp.Models;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;

namespace ExampleApp {
    public class Startup {

        private IConfigurationRoot Configuration;

        public Startup(IHostingEnvironment env) {
            Configuration = new ConfigurationBuilder()
                .SetBasePath(env.ContentRootPath)
                .AddEnvironmentVariables()
                .Build();
        }

        public void ConfigureServices(IServiceCollection services) {

            var host = Configuration["DBHOST"] ?? "localhost";
            var port = Configuration["DBPORT"] ?? "3306";
            var password = Configuration["DBPASSWORD"] ?? "mysecret";

            services.AddDbContext<ProductDbContext>(options =>
                options.UseMySql($"server={host};userid=root;pwd={password};"
                    + $"port={port};database=products"));

            services.AddSingleton<IConfiguration>(Configuration);
            services.AddTransient<IRepository, ProductRepository>();
            services.AddMvc();
        }
        public void Configure(IApplicationBuilder app,
                IHostingEnvironment env, ILoggerFactory loggerFactory) {

            loggerFactory.AddConsole();
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();

            SeedData.EnsurePopulated(app);
        }
    }
}
```

In conventional MVC projects, the connection string for the database is defined in a JSON configuration file, but when working with containers, it is simpler to use environment variables because they can be easily specified when a container is started so that you don't have to rebuild the image and re-create the containers each time there is a configuration change. In this example, the host name, TCP port, and password are configured by reading the value of environment variables called `DBHOST`, `DBPORT`, and `DBPASSWORD`. There are default values for these configuration settings that will be used if the environment variables are not defined.

Creating the Database Migration

The next step is to create the initial Entity Framework Core migration that will define the schema for the application. This uses the EF Core Code First feature, which generates database schemas automatically from the data model classes in an ASP.NET Core application. Run the command shown in Listing 5-24 from the `ExampleApp` folder to create the migration.

Listing 5-24. Creating the Initial Database Migration

```
dotnet ef migrations add Initial
```

If you are using Visual Studio, you can open the Package Manager Console (available in the Tools ➤ NuGet Package Manager menu) and run the PowerShell command shown in Listing 5-25 instead.

Listing 5-25. Creating the Database Migrations Using the Visual Studio Package Manager Console

```
Add-Migration Initial
```

Regardless of which command you use, Entity Framework Core will create an `Example/Migrations` folder that contains C# classes that will be used create the database schema. (If you see an error telling you that the term `Add-Migration` is not recognized, then restart Visual Studio and load the example project again. Once the project has loaded, you should be able to run the `Add-Migration` command from the Package Manager Console.)

■ **Note** You might be used to running the `dotnet ef database update` or `Update-Database` command at this point, but these are not helpful when working with Docker. Instead, the MVC application has been configured to apply the migration when it starts, as shown in Listing 5-23 or by using the techniques described in later chapters.

Changing the View

Earlier examples modified the banner message in the Razor view to indicate when content has been altered. Return the view to a simpler message that displays just the value provided by the controller, as shown in Listing 5-26.

Listing 5-26. Resetting the Contents of the Index.cshtml File in the ExampleApp/Views/Home Folder

```
@model IEnumerable<ExampleApp.Models.Product>
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>ExampleApp</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
</head>
<body>
    <div class="m-1 p-1">
        <h4 class="bg-success text-xs-center p-1 text-white">@ViewBag.Message</h4>
        <table class="table table-sm table-striped">
            <thead>
                <tr><th>ID</th><th>Name</th><th>Category</th><th>Price</th></tr>
            </thead>
            <tbody>
                @foreach (var p in Model) {
                    <tr>
                        <td>@p.ProductID</td>
                        <td>@p.Name</td>
                        <td>@p.Category</td>
                        <td>$@p.Price.ToString("F2")</td>
                    </tr>

                }
            </tbody>
        </table>
    </div>
</body>
</html>
```

Creating the MVC Application Image

The final preparatory step is to update the image for the MVC application so that it includes the code changes to support the database, the Entity Framework Core packages, and the initial migration that will create the database schema and apply the seed data. Run the commands shown in Listing 5-27 in the `ExampleApp` folder to publish the application and create a new Docker image.

Listing 5-27. Updating the MVC Application Image

```
dotnet publish --framework netcoreapp1.1 --configuration Release --output dist
docker build . -t apress/exampleapp -f Dockerfile
```

The `dotnet publish` command prepares the MVC application for containerization and writes all of the files that it requires into the `dist` folder, as explained in Chapter 4. The `docker build` command uses the `Dockerfile` to generate a new image, which is assigned the `apress/exampleapp` tag.

Testing the Application

All that remains is to test the new image by creating a container for the ASP.NET Core MVC application and ensuring that it can communicate with the MySQL database, which is already running in its own container, created in Listing 5-16. This testing requires a little work, but it provides some useful insights into how Docker works.

When you start a container, Docker connects it to an internal virtual network and assigns it an Internet Protocol (IP) address so that it can communicate with the host server and with other containers on the same network. This is the entry point for a key Docker feature that I describe in the next section of this chapter (known as *software-defined networks*).

To have the MVC container talk to the database, I need to know the IP address that Docker assigned to the MySQL container. Run the command shown in Listing 5-28 to examine the configuration of the Docker virtual network.

Listing 5-28. Examining the Docker Virtual Network

```
docker network inspect bridge
```

The response from this command will show you how Docker has configured the virtual network and will include a `Containers` section that shows the containers that are connected to the network and the IP addresses that are assigned to them.

There should be only one container, and its `Name` field will be `mysql`. Make a note of the `IPv4Address` field, as follows:

```
...
"Containers": {
    "72753560ccb4d876bdeaad36e0b39354a08e90ad30ac1a78b20aad3e52b7a101": {
        "Name": "mysql",
        "EndpointID": "04f18afbf953a030ddfbbadf4a8f86c1dbf6c6a6736449326",
        "MacAddress": "02:42:ac:11:00:02",
        "IPv4Address": "172.17.0.2/16",
        "IPv6Address": ""
    }
},
...
```

This is the IP address that Docker has assigned to the container. For me, the address is `172.17.0.2`, but you may see a different address. This is the IP address that the MVC application must use in its database connection to communicate with MySQL. This address can be provided to the application through the `DBHOST` environment variable, which is read by the additions to the MVC application's `Startup` class that I made in Listing 5-22.

Run the commands shown in Listing 5-29 to create and start an MVC container in the background and then to monitor its output. Make sure you use the IP address that has been assigned to the MySQL container on your system, shown in the output you received from the command in Listing 5-29.

Listing 5-29. Creating and Starting a New MVC Application Container

```
docker run -d --name productapp -p 3000:80 -e DBHOST=172.17.0.2 apress/exampleapp
docker logs -f productapp
```

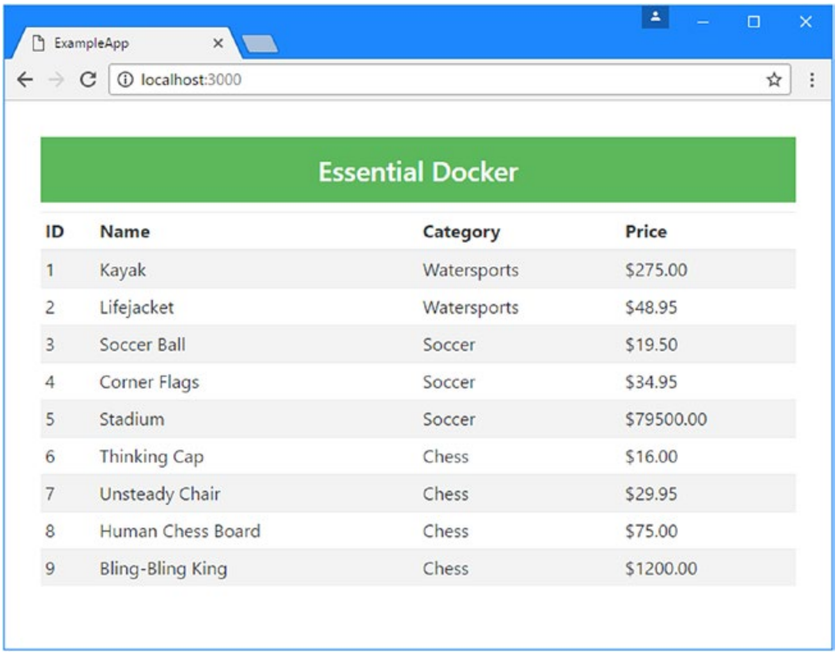
As the MVC application starts up, you will see messages that show that Entity Framework Core has applied its migrations to the database, which ensures that the schema is created and that the seed data is added.

```
...
Applying Migrations...
info: Microsoft.EntityFrameworkCore.Migrations.Internal.Migrator[12]
      Applying migration '20161215112411_Initial'.
Creating Seed Data...

...SQL statements omitted for brevity...

Hosting environment: Production
Content root path: /app
Now listening on: http://+:80
Application started. Press Ctrl+C to shut down.
...
```

The `docker run` command in Listing 5-29 maps port 3000 on the host operating system to port 80 in the container, which is the port that Kestrel is using to receive HTTP requests for the ASP.NET Core runtime. To test the application, open a new browser tab and request the URL `http://localhost:3000`. The browser will send an HTTP request that Docker will receive and direct to port 80 in the MVC container, producing the response shown in Figure 5-1.



Essential Docker

ID	Name	Category	Price
1	Kayak	Watersports	$275.00
2	Lifejacket	Watersports	$48.95
3	Soccer Ball	Soccer	$19.50
4	Corner Flags	Soccer	$34.95
5	Stadium	Soccer	$79500.00
6	Thinking Cap	Chess	$16.00
7	Unsteady Chair	Chess	$29.95
8	Human Chess Board	Chess	$75.00
9	Bling-Bling King	Chess	$1200.00

Figure 5-1. *Testing the example application*

The data displayed in the response from the MVC application was obtained from the MySQL server using Entity Framework Core. The Docker network allowed the EF Core to open a network connection to the MySQL container and use it to send a SQL query. Both components exist in their own containers and make network requests as they would normally, unware that the other container exists on the same server and that the network that connects them is virtual and has been created and managed by Docker.

Once you have tested that the MVC application works, type `Control+C` to stop monitoring the output and leave the container running in the background. You can check that the container is still running with the `docker ps` command.

Working with Software-Defined Networks

The virtual network that allowed the containers to communicate in the previous section is an example of a *software-defined network* (SDN). As the name suggests, SDNs are networks that are created and managed using software. SDNs behave like traditional networks and use regular IP addresses ports, but there are no physical network interfaces, and the infrastructure for the networks, such as name services and routing, are provided by Docker.

SDNs allow containers to communicate. Simple SDNs are restricted to a single server. The default network that I used in the previous section, known as the *default bridge network*, is a good example. This network is one of a set that Docker creates when it starts. Run the command shown in Listing 5-30 to see the default networks that Docker provides.

Listing 5-30. Listing the Docker Networks

```
docker network ls
```

This command lists all the current SDNs and produces the following output:

```
NETWORK ID          NAME                DRIVER              SCOPE
3826803e0a8c        bridge              bridge              local
0e643a56cce6        host                host                local
9058e96d6113        none                null                local
```

The `host` network is the host server's network, and the `none` network is a network that has no connectivity and that can be used to isolate containers completely. The network whose name is `bridge` is the one that is of most interest because Docker adds all containers to this network when it creates them. Run the command shown in Listing 5-31 to inspect the default bridge network.

Listing 5-31. Inspecting the Default Bridge Network

```
docker network inspect bridge
```

Now that there are two containers running, you will see an additional entry in the `Containers` section, showing the network configuration that Docker has applied to the MVC container, like this (although you may see different addresses):

```
...
"Containers": {
    "1be5d5609e232335d539b29c49192c51333aaea1fd822249342827456aefc02e": {
        "Name": "productapp",
        "EndpointID": "3ff7acdaf3cce9e77cfc7156a04d6fa7bf4b5ced3fd13",
        "MacAddress": "02:42:ac:11:00:03",
        "IPv4Address": "172.17.0.3/16",
        "IPv6Address": ""
    },
    "9dd1568078b63104abc145840df8502ec2e171f94157596236f00458ffbf0f02": {
        "Name": "mysql",
        "EndpointID": "72b3df290f3fb955e1923b04909b7f27fa3854d8df583",
        "MacAddress": "02:42:ac:11:00:02",
        "IPv4Address": "172.17.0.2/16",
        "IPv6Address": ""
    }
},
...
```

The best way to understand the effect of the Docker SDN is to understand the composition of the application that was created in the previous section and to consider the connectivity of the MVC and MySQL containers, as illustrated in Figure 5-2.

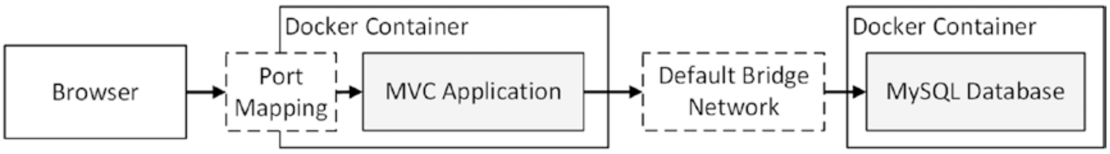


Figure 5-2. *Using the default bridge network*

The browser sends its HTTP request to a port on the host operating system that Docker maps to a port in the MVC application's container. The MVC application asks Entity Framework Core to provide it with data, which it does by using the default bridge network to communicate with the MySQL application running in a separate container. The command in Listing 5-16 that created the MySQL container didn't include a port mapping, which means that the MySQL container isn't accessible via a port on the host operating system.

■ **Note** The software-defined networks created in this chapter exist only on a single host server, which means that all the containers connected to the network must be running on that server. Docker also supports software-defined networks that span all the nodes in a cluster of servers, which I demonstrate in Chapter 7.

Scaling the MVC Application

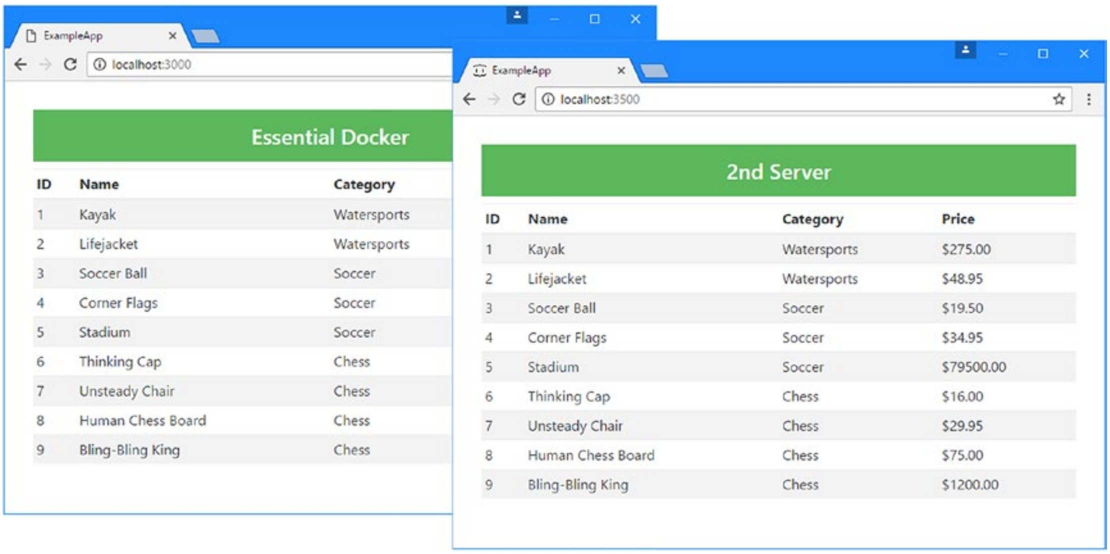
Software-defined networks behave like physical networks, and containers behave like servers connected to them. This means that scaling up the MVC application so that there are multiple ASP.NET Core servers is as simple as creating and starting additional containers. Run the command in Listing 5-32 to add a new MVC container. (Enter the command as a single line.)

Listing 5-32. Creating an Additional MVC Application Container

```
docker run -d --name productapp2 -p 3500:80 -e DBHOST=172.17.0.2
    -e MESSAGE="2nd Server" apress/exampleapp
```

When creating additional containers for an application, it is important to ensure the new containers have different names (`productapp2` in this case) and different port mappings (port 3500, instead of port 3000, is mapped to port 80 for this container). To further differentiate the new container, the `-e` argument has been used to set the `MESSAGE` environment variable, which will be displayed in the MVC application's output. The `-d` argument tells Docker to start the container in the background.

The result is that there are containers for the MVC application that will handle requests on ports 3000 and 3500 of the host server. To test the containers, open browser tabs and request `http://localhost:3000` and `http://localhost:3500`, which will produce the results shown in Figure 5-3. Notice that the banner in the response from the new container says "2nd Server," differentiating it from the original container.



Essential Docker

ID	Name	Category
1	Kayak	Watersports
2	Lifejacket	Watersports
3	Soccer Ball	Soccer
4	Corner Flags	Soccer
5	Stadium	Soccer
6	Thinking Cap	Chess
7	Unsteady Chair	Chess
8	Human Chess Board	Chess
9	Bling-Bling King	Chess

2nd Server

ID	Name	Category	Price
1	Kayak	Watersports	$275.00
2	Lifejacket	Watersports	$48.95
3	Soccer Ball	Soccer	$19.50
4	Corner Flags	Soccer	$34.95
5	Stadium	Soccer	$79500.00
6	Thinking Cap	Chess	$16.00
7	Unsteady Chair	Chess	$29.95
8	Human Chess Board	Chess	$75.00
9	Bling-Bling King	Chess	$1200.00

Figure 5-3. *Creating additional application containers*

Figure 5-4 shows the composition of the application following the addition of a second MVC application container.

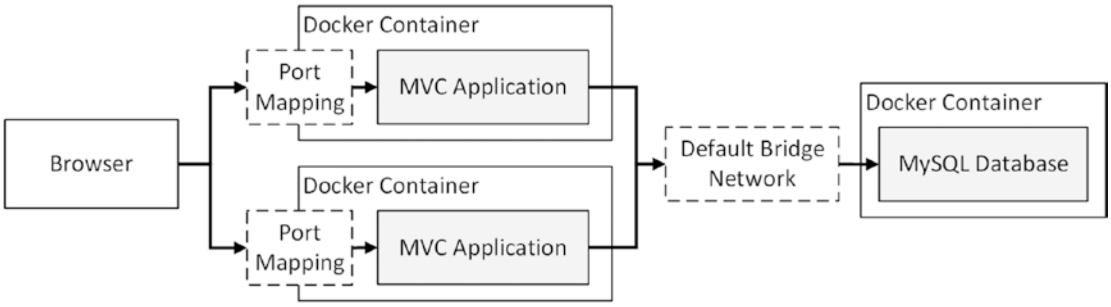


Figure 5-4. *The effect of adding an additional container*

Docker assigns each container its own IP address, and the containers can communicate freely with one another.

Creating Custom Software-Defined Networks

Using the default bridge network demonstrates the basic networking features of Docker, but it has two main limitations. The first limitation is the awkward process of inspecting the network to get the IP address of the MySQL container to configure the MVC containers.

The second limitation is that all the containers are connected to the same network, while large-scale applications are generally designed with multiple networks that separate out the different functional areas so they can be monitored and managed independently. Fortunately, both of these limitations can be addressed by creating custom SDNs instead of using the default bridge network.

To prepare for this section, run the command shown in Listing 5-33 to stop and remove the database and MVC containers.

Listing 5-33. Removing the Containers

```
docker rm -f $(docker ps -aq)
```

Creating Custom Networks

Custom software-defined networks are created using the `docker network create` command, followed by the name for the new network. Run the commands shown in Listing 5-34 to create two new software-defined networks called `frontend` and `backend`.

Listing 5-34. Creating Custom Software-Defined Networks

```
docker network create frontend
docker network create backend
```

The `frontend` network will be used to receive HTTP requests by the MVC containers. The `backend` network will be used for SQL queries between the MVC containers and the MySQL container. Run the `docker network ls` command and you will see that the output includes the new networks, like this:

```
NETWORK ID          NAME                DRIVER              SCOPE
778d9eb6777a        backend             bridge              local
fa1bc701b306        bridge              bridge              local
2f0bb28d5716        frontend            bridge              local
0e643a56cce6        host                host                local
9058e96d6113        none                null                local
```

Connecting Containers to Custom Networks

Once you have created custom networks, you can connect containers to them using the `--network` argument, which can be used with the `docker create` and `docker run` commands. Run the command shown in Listing 5-35 to create a new database container that is connected to the `backend` network. (Enter the command as a single line.)

Listing 5-35. Creating a Container Connected to a Network

```
docker run -d --name mysql -v productdata:/var/lib/mysql --network=backend
    -e MYSQL_ROOT_PASSWORD=mysecret -e bind-address=0.0.0.0 mysql:8.0.0
```

There is one new argument in this command, which is described in Table 5-4.

Table 5-4. *Additional Argument Used to Create the MySQL Container*

Name	Description
`--network`	This argument is used to assign a container to a network. In this case, the container is assigned to the network called `backend`.

Notice that there are no port mappings in this command, which means the database cannot be reached through the host operating system. Instead, it will only be able to receive connections through the backend software-defined network.

Understanding the Docker DNS Service

Docker configures containers so their Domain Name System (DNS) requests resolve the names assigned to containers to the IP addresses they have been given on custom software-defined networks. (This feature is not available on the default bridge network.)

The DNS configuration means that container names can be used as host names, avoiding the need to locate the IP address assigned to a container. Run the command shown in Listing 5-36 to perform a simple test of the Docker DNS feature.

Listing 5-36. Testing the Docker DNS Feature

```
docker run -it --rm --network backend alpine:3.4 ping -c 3 mysql
```

This command creates and runs an image using the Alpine Linux distribution and executes the `ping` command to a host called `mysql`, which Docker will automatically resolve to the IP address that has been assigned to the `mysql` container on the `backend` software-defined network. The command will produce the following results:

```
...
PING mysql (172.19.0.2): 56 data bytes
64 bytes from 172.19.0.2: seq=0 ttl=64 time=0.070 ms
64 bytes from 172.19.0.2: seq=1 ttl=64 time=0.124 ms
64 bytes from 172.19.0.2: seq=2 ttl=64 time=0.128 ms

--- mysql ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.070/0.107/0.128 Microsoft
...
```

Once the `ping` command is complete, the container will exit and be removed automatically.

Creating the MVC Containers

The embedded DNS feature and the fact that port mappings are not required makes it simpler to create multiple MVC application containers because only the container names must be unique. Run the commands shown in Listing 5-37 to create three containers for the MVC application. Each command should be entered on a single line.

Listing 5-37. Creating the MVC Application Containers

```
docker create --name productapp1 -e DBHOST=mysql -e MESSAGE="1st Server"
    --network backend apress/exampleapp
docker create --name productapp2 -e DBHOST=mysql -e MESSAGE="2nd Server"
    --network backend apress/exampleapp
docker create --name productapp3 -e DBHOST=mysql -e MESSAGE="3rd Server"
    --network backend apress/exampleapp
```

The `docker run` and `docker create` commands can connect a container only to a single network. The commands in Listing 5-37 specified the `backend` network, but this is only one of the connections needed for the MVC containers. Run the commands in Listing 5-38 to connect the MVC containers to the `frontend` network as well.

Listing 5-38. Connecting the Application Containers to Another Network

```
docker network connect frontend productapp1
docker network connect frontend productapp2
docker network connect frontend productapp3
```

The `docker network connect` command connects existing containers to software-defined networks. The commands in the listing connect the three MVC containers to the `frontend` network.

Now that the MVC application containers have been connected to both Docker networks, run the command in Listing 5-39 to start them.

Listing 5-39. Starting the Application Containers

```
docker start productapp1 productapp2 productapp3
```

Adding a Load Balancer

The MVC containers were created without port mappings, which means they are accessible only through the Docker software-defined networks and cannot be reached through the host operating system.

The missing piece of the puzzle is a load balancer, which will receive HTTP requests on a single port mapped to the host operating system and distribute them to the MVC application containers over the `frontend` Docker network.

There are lots of options when it comes to load balancers, and the choice is usually tied to the infrastructure platform to which the application is deployed. For example, if you are deploying your application into a corporate network, you are likely to find that there are load balancers already installed and running. Equally, most cloud platforms will offer a load balancer service, although its use is often optional.

For this example, I have selected HAProxy, which is an excellent HTTP load balancer that has been adopted by Docker for use in its cloud environment and provided through an image that I use in Chapter 6.

■ **Tip** HAProxy can be used in a Linux container but doesn't provide support for Windows. If you are using Windows containers and require a load balancer, then consider Nginx (`http://nginx.org`).

To configure HAProxy, add a file called `haproxy.cfg` to the `ExampleApp` folder with the content shown in Listing 5-40.

Listing 5-40. The Contents of the haproxy.cfg File in the ExampleApp Folder

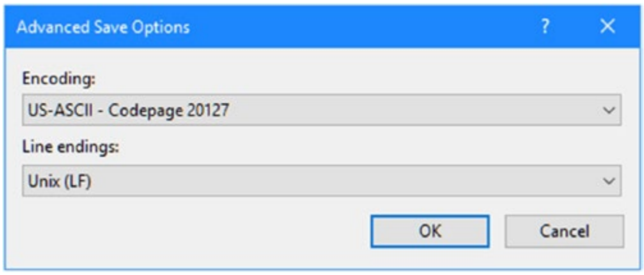
```
defaults
    timeout connect 5000
    timeout client  50000
    timeout server  50000

frontend localnodes
    bind *:80
    mode http
    default_backend mvc
```

```
backend mvc
    mode http
    balance roundrobin
    server mvc1 productapp1:80
    server mvc2 productapp2:80
    server mvc3 productapp3:80
```

This configuration tells HAProxy to receive connections on port 80, which will be exposed through a mapping on the host operating system, and distribute them to the three MVC containers, which are identified using their container names, to take advantage of the Docker DNS feature. HTTP requests will be distributed to each MVC container in turn, which makes it easier to test the load balancer and show that all the MVC containers are being used.

If you are using Visual Studio, you must make sure the character encoding and line endings in the `haproxy.cfg` file meet the expectations of the HAProxy application. Select File ➤ Save As and click the down arrow on the Save button to select the Save With Encoding option. Select the options shown in Figure 5-5 and save the file. If you don't change the configuration, the HAProxy process won't be able to read its configuration file and will report an error.



Encoding: US-ASCII - Codepage 20127

Line endings: Unix (LF)

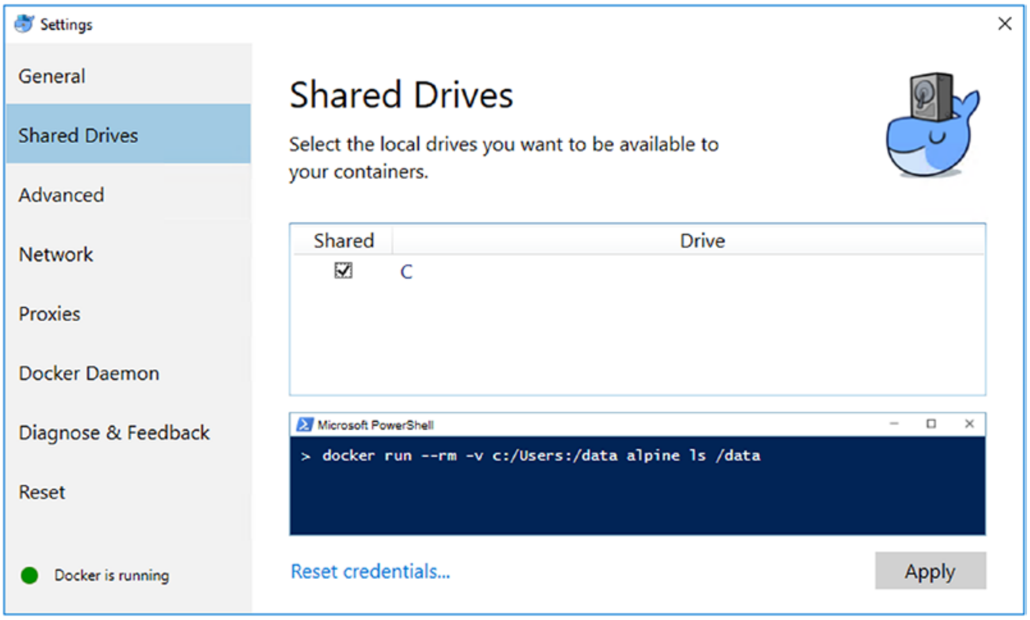
Figure 5-5. *Configuring the save options for the load balancer configuration file*

MATCHING LOAD BALANCING AND SESSION STORAGE STRATEGIES

This configuration uses the HAProxy round-robin load balancing strategy, which sends a request to each of the servers in turn. If your application uses session data, you must ensure that your load balancing strategy suits your session storage model. If each application container maintains its own isolated storage, then you must configure the load balancer so that subsequent requests from a client return to the same server. This can mean that HTTP requests are not evenly distributed but ensures that sessions work as expected. If you store your session data so that it is accessible from any of the application containers, then round-robin load balancing will be fine. See your load balancer configuration settings for the options available. If you are using HAProxy, then the configuration options can be found at `www.haproxy.org`.

To make the configuration file for the load balancer available, I am going to use a Docker volumes feature that allows files and folders from the host operating system to provide the contents of a directory in a container's file system, rather than requiring a dedicated Docker volume. For Windows and macOS, some configuration is required (no such configuration is required for Linux).

For Windows users, right-click the Docker icon in the task bar and select Settings from the pop-up menu. Click Shared Drives and check the box for the drive that contains the `ExampleApp` folder, as shown in Figure 5-6.



Shared Drives

Select the local drives you want to be available to your containers.

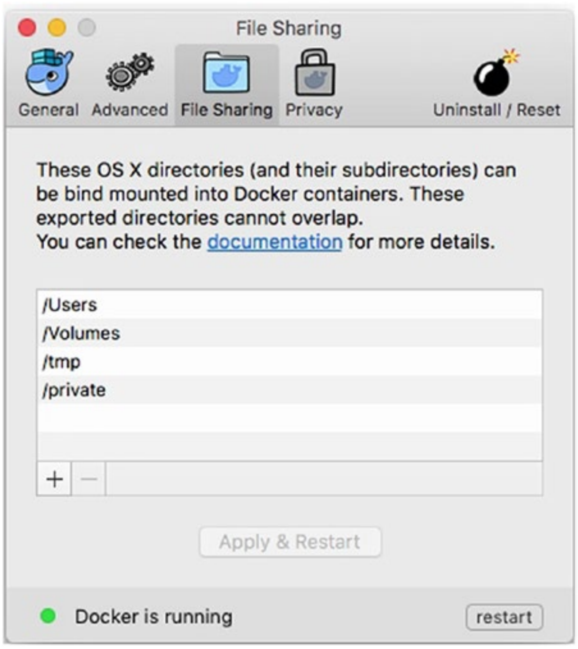
Shared	Drive
☑	C

```
> docker run --rm -v c:/Users:/data alpine ls /data
```

Figure 5-6. *Enabling drive sharing on Windows*

Click Apply, provide the administrator's account password if prompted, and close the Settings window.

For macOS users, click the Docker menu bar icon and select Preference from the pop-up menu. Click File Sharing and ensure that the folder that contains the `ExampleApp` folder is on the list of shared directories, as shown in Figure 5-7.



These OS X directories (and their subdirectories) can be bind mounted into Docker containers. These exported directories cannot overlap.
You can check the documentation for more details.

/Users
/Volumes
/tmp
/private

Figure 5-7. *Enabling drive sharing on macOS*

For all operating systems, run the command in Listing 5-41 from the ExampleApp directory to create and start new container that runs HAProxy, connected to the front-end Docker network and configured to listen for HTTP requests on port 80 of the host operating system. Take care to enter the command on a single line.

Listing 5-41. Creating and Starting the Load Balancer Container

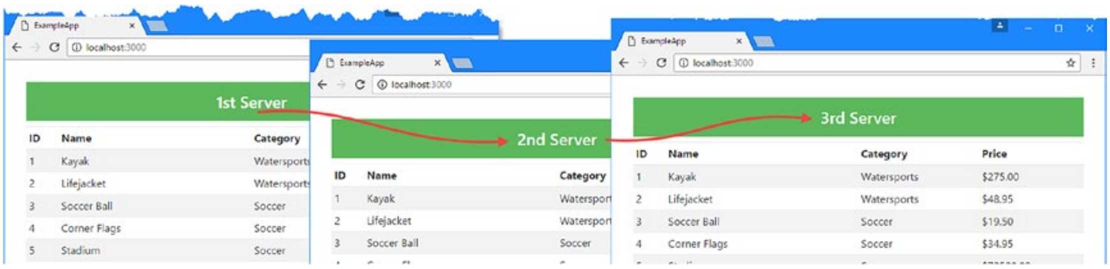
```
docker run -d --name loadbalancer --network frontend
    -v "$(pwd)/haproxy.cfg:/usr/local/etc/haproxy/haproxy.cfg"
    -p 3000:80 haproxy:1.7.0
```

Enter the command on a single line. The `-v` argument in this command tells Docker to mount the `haproxy.cfg` file that is in the `ExampleApp` folder in the container so that it appears in the `/usr/local/etc/haproxy` directory. This is a useful feature for creating containers that require configuration files. (It is also useful for containerized development environments, which I describe in Chapter 8.)

The `--network` argument connects the load balancer container to the `frontend` network so that it can communicate with the MVC containers. The `-p` argument maps port 3000 on the host operating system to port 80 in the container so that the load balancer can receive requests from the outside world.

■ **Tip** If you see an error when the container starts, then you most likely forgot to change the character encoding for the `haproxy.cfg` file, as shown in Figure 5-4. Change the settings, save the configuration file, and then run `docker rm -f loadbalancer` to remove the container before repeating the command in Listing 5-41.

Once the load balancer has started, open a browser tab and request the URL `http://localhost:3000`. Reload the page and you will see that the load balancer distributes the request between the MVC application containers, which is reflected in the banner shown in the response, as illustrated by Figure 5-8. (The first request to each MVC container may take a second as the application starts up, but if you keep reloading, subsequent requests should be handled immediately.)



1st Server

ID	Name	Category
1	Kayak	Watersports
2	Lifejacket	Watersports
3	Soccer Ball	Soccer
4	Corner Flags	Soccer
5	Stadium	Soccer

2nd Server

ID	Name	Category
1	Kayak	Waterspor
2	Lifejacket	Waterspor
3	Soccer Ball	Soccer

3rd Server

ID	Name	Category	Price
1	Kayak	Watersports	$275.00
2	Lifejacket	Watersports	$48.95
3	Soccer Ball	Soccer	$19.50
4	Corner Flags	Soccer	$34.95

Figure 5-8. *Load balancing between multiple application containers*

The final result is an application that contains five containers, two software-defined networks, and a single port mapping. The composition of the application is shown in Figure 5-9 and is similar to the way that applications are conventionally composed, albeit all the components are running on a single host server and are isolated by containers.

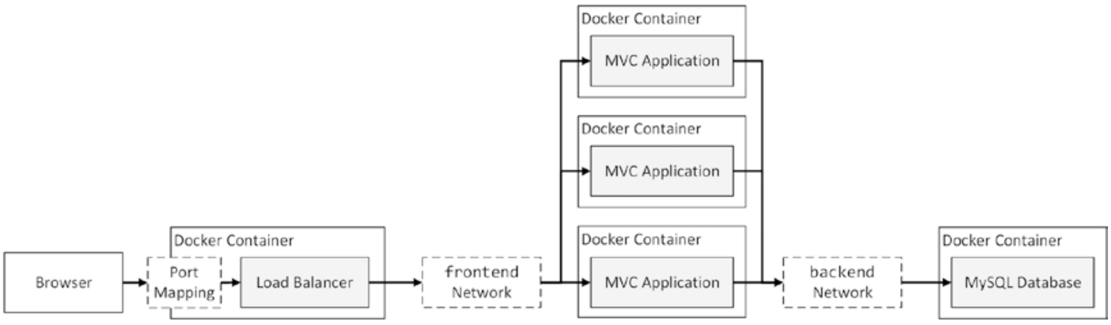


Figure 5-9. *The composition of an application with multiple software-defined networks*

The database container is connected only to the `backend` network. The MVC application containers are connected to both the `frontend` and `backend` networks; they receive HTTP requests through the `frontend` network and make SQL queries over the `backend` network. The load balancer container is connected only to the `frontend` network and has a port mapping onto the host operating system; it receives HTTP requests via the port mapping and distributes them to the MVC application containers.

Summary

In this chapter, I described the Docker volume and software-defined networking features, which allow more complex applications to be composed by combining containers. I explained how volumes are used to keep data files outside of a container's storage and how this changes the way that Entity Framework Core is used to create database schemas. I also introduced software-defined networks and showed you how they can be used to create a familiar structure for an application, even though all the components exist in Docker containers rather than on dedicated server hardware. In the next chapter, I show you how to describe more complex applications without having to create and manage the containers, volumes, and networks individually.