CHAPTER 6

Docker Compose

In the previous chapter, I showed you how a complex application can be created by combining containers, volumes, and software-defined networks. The problem with this approach is that each step has to be performed manually, which can be an error-prone process. Not only does each command have to be entered correctly, but the steps have to be performed in the order covered in Chapter 5 because the container relies on being able to send network requests to the container created before it. If you skip a step or perform a step out of order, then you may not end up with a working application.

In this chapter, I show you how to use *Docker Compose*, which is used to describe complex applications in a consistent and predictable manner. A *compose file* contains details of all the volumes, networks, and containers that make up an application and the relationships between them. I explain how Docker Compose is used and demonstrate how to describe the example application using a compose file so that all the components it needs can be created and started in a single step. Table 6-1 puts Docker Compose into context.

Table 6-1. *Putting Docker Compose in Context*

Question	Answer
What is it?	Docker Compose is a tool that is used to describe complex applications and manage the containers, networks, and volumes they require.
Why is it useful?	Docker Compose simplifies the process of setting up and running applications so that you don't have to type in complex commands, which can lead to configuration errors.
How is it used?	The description of the application and its requirements is defined in a compose file, which is processed using the `docker-compose` command. The number of containers in an application is changed using the `docker-compose scale` command.
Are there any pitfalls or limitations?	Docker doesn't provide any way to wait for the application in one container to be ready to receive requests from another container. As a consequence, you must manage the dependencies between the containers in an application directly.
Are there any alternatives?	You do not have to use Docker Compose. You can create and manage your containers manually or use an alternative such as Crowdr (`https://github.com/polonskiy/crowdr`).

A. Freeman, *Essential Docker for ASP.NET Core MVC*, DOI 10.1007/978-1-4842-2778-7_6

Table 6-2 summarizes the chapter.

Table 6-2. *Chapter Summary*

Problem	Solution	Listing
Describe a complex application	Use a Docker Compose file	1–5, 7, 17, 18
Process the contents of a compose file	Use the `docker-compose build` command	6, 8, 20
Create and start the components in a compose file	Use the `docker-compose up` command	9, 21, 22
Remove the components in a compose file	Use the `docker-compose down` command	10, 27
Wait until an application is ready to receive connections	Use the `wait-for-it` package (for Linux containers) or create a PowerShell script (for Windows containers)	11–16, 19
Change the number of containers running for a service	Use the `docker-compose scale` command	23, 25
List the running services	Use the `docker-compose ps` command	24
Stop the containers that have been started using a compose file	Use the `docker-compose stop` command	26

Preparing for This Chapter

This chapter depends on the ExampleApp MVC project created in Chapter 3 and modified in the chapters since. 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 that accompanies this book. See the `apress.com` page for this book.

To ensure that there is no conflict with examples from previous chapters, run the commands shown in Listing 6-1 to remove the containers, networks, and volumes that were created in the previous chapter. Ignore any errors or warnings these commands produce.

Listing 6-1. Removing the Containers, Networks, and Volumes

```
docker rm -f $(docker ps -aq)
docker network rm $(docker network ls -q)
docker volume rm $(docker volume ls -q)
```

Changing the View Message

In this chapter, I demonstrate how to scale up an MVC application by creating multiple containers automatically. To help show which container is handling a specific HTTP request, Listing 6-2 shows a change to the `Home` controller, altering the message passed by the action method to its view.

Listing 6-2. Changing the Banner in the HomeController.cs File in the ExampleApp/Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using ExampleApp.Models;
using Microsoft.Extensions.Configuration;
```

```
namespace ExampleApp.Controllers {
    public class HomeController : Controller {
        private IRepository repository;
        private string message;

        public HomeController(IRepository repo, IConfiguration config) {
            repository = repo;
          message =  $"Essential Docker ({config["HOSTNAME"]})";
        }

        public IActionResult Index() {
            ViewBag.Message = message;
            return View(repository.Products);
        }
    }
}
```

Docker creates a HOSTNAME environment variable inside containers, which is set to the unique ID of the container. The change to the controller ensures that the response from the application indicates which container handles the request. This is not something you should do in real projects, but it is a useful way to demonstrate how key Docker Compose features work.

Installing Docker Compose on Linux

Docker Compose is included in the Docker installers for Windows and macOS but must be installed separately in Linux. If you are a Linux user, run the commands shown in Listing 6-3 to download and install Docker Compose.

Listing 6-3. Installing Docker Compose for Linux

```
sudo curl -L "https://github.com/docker/compose/releases/download/1.11.2/docker-compose-
$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
```

Once the installation is complete, run command shown in Listing 6-4 to check that Docker Compose works as expected.

Listing 6-4. Checking Docker Compose

```
docker-compose --version
```

If the installation has been successful, you will see this response: `docker-compose version 1.11.2, build dfed245`.

Creating the Compose File

Docker Compose allows complex applications to be described in a configuration file, known as a compose file, that ensures that all the components are set up correctly when the application is deployed.

The starting point is to create the compose. Create a file called `docker-compose.yml`, which is the conventional name for the compose file, in the `ExampleApp` folder and add the configuration statements shown in Listing 6-5.

Listing 6-5. The Contents of the docker-compose.yml File in the ExampleApp Folder

```
version: "3"

volumes:
  productdata:

networks:
  frontend:
  backend:
```

The `yml` file extension denotes that the compose file is expressed in the YAML format, which is introduced in the "*Getting Started with YAML*" sidebar. Table 6-3 describes the configuration properties in Listing 6-5.

Table 6-3. *The Initial Docker Compose Configuration Properties*

Name	Description
`version`	This setting specifies the version of the Docker Compose schema that the file uses. At the time of writing, the latest version is 3.
`volumes`	This setting is used to configure the volumes that will be used by the containers defined in the compose file. This example defines a volume called `productdata`.
`networks`	This setting is used to configure the software-defined networks that will be used by the containers defined in the compose file. This example defines networks called `frontend` and `backend`.

The `version` section tells Docker which version of the compose file schema is being used. As I write this, the latest version is `3`, which includes support for the latest Docker features. Docker is a rapidly evolving platform, so you may find that there is a later version available by the time you read this, although the examples in this chapter should still work as long as you don't change the `version` setting in the compose file.

GETTING STARTED WITH YAML

YAML is a format used to create configuration files that are human-readable, although it can be confusing when you first start using it. Editing YAML is a lot simpler with the extensions for Visual Studio and Visual Studio Code described in Chapter 3.

The most important aspect of YAML to bear in mind is that indentation with spaces is used to signify the structure of the file and that tabs are forbidden. This runs counter to most file formats that programmers use, where indentation can be adjusted or ignored and tabs can be used freely. In Listing 6-5, for example, the `version` and `networks` keywords must have no indentation because they are top-level configuration sections, and the `frontend` and `backend` entries are indented by two spaces to indicate they are part of the `networks` section.

> The simplest way to avoid problems with YAML for the examples in this chapter is to use the file from the source code download linked from the apress.com page for this book. But as you start using YAML in your own projects, there are three sources of information that will provide you with the details you require to create compose files.
>
> The first source is the Docker compose file reference, https://docs.docker.com/compose/compose-file, which describes each entry that a compose file contains and provides useful YAML examples.
>
> The second source of information is the schema for the compose file, https://github.com/docker/compose/blob/master/compose/config/config_schema_v3.0.json, which explains how an entry is expressed in YAML. YAML represents data using three primitive structures: key/value maps or objects, lists, and scalars (which are strings or numbers). The compose file schema tells you which of the primitives is used and provides details of any additional requirements.
>
> The final source of information is the YAML standard, http://yaml.org/spec/1.2/spec.html, which will help you make sense of the YAML structure. You won't often need to refer to the standard, but it can throw some light on what is important in a YAML document, which can be helpful when Docker Compose reports errors parsing your files.

Compose files are processed using a command-line tool called `docker-compose` (note the hyphen), which builds and manages the application.

There is little useful configuration in the example compose file at the moment, but run the command shown in Listing 6-6 in the `ExampleApp` folder to build the application anyway, just to make sure everything is working.

Listing 6-6. Building the Application

```
docker-compose -f docker-compose.yml build
```

The `-f` argument is used to specify the name of the compose file, although Docker will default to using `docker-compose.yml` or `docker-compose.yaml` if a file isn't specified. The `build` argument tells Docker to process the file and build the Docker images it contains. There are no images defined in the file at the moment, so you will see the following output:

```
...
WARNING: Some networks were defined but are not used by any service: frontend, backend
...
```

This warning indicates that the compose file tells Docker to create some software-defined networks, but they are not used anywhere else in the application and so they were not created.

If you see an error, go back and check that you have used tabs to re-create the structure as shown in Listing 6-5 and that you have remembered to add a colon after the `networks`, `frontend`, and `backend` configuration entries.

Composing the Database

The process for describing the application using Docker Compose follows the same path as creating the containers manually, and the next step is to configure the database container. Listing 6-7 shows the additions to the compose file that sets up the container for MySQL.

Listing 6-7. Defining the Database Container in the docker-compose.yml File in the ExampleApp Folder

```
version: "3"

volumes:
  productdata:

networks:
  frontend:
  backend:

services:

  mysql:
    image: "mysql:8.0.0"
    volumes:
      - productdata:/var/lib/mysql
    networks:
      - backend
    environment:
      - MYSQL_ROOT_PASSWORD=mysecret
      - bind-address=0.0.0.0
```

The `services` keyword is used to denote the section of the compose file that contains the descriptions that will be used to create containers. The term *service* is used because the description can be used to create more than one container. Each service is given its own section, as you will see as the compose file is expanded throughout the rest of the chapter.

The service described in Listing 6-7 is called `mysql`, and it describes how database containers should be created using the configuration properties described in Table 6-4.

Table 6-4. *The Configuration Properties for the MySQL Service*

Name	Description
`services`	This property denotes the start of the `services` section of the compose file, which describes the services that will be used to create containers.
`mysql`	This property denotes the start of a service description called `mysql`.
`image`	This property specifies the Docker image that will be used to create containers. In this example, the official MySQL image will be used.
`volumes`	This property specifies the volumes that will be used by containers and the directories they will be used for. In this example, the `productdata` volume will be used to provide the contents of the `/var/lib/mysql` directory.
`networks`	This property specifies the networks that containers will be connected to. In this example, containers will be connected to the `backend` network. (The `networks` keyword used in a service has a different meaning than the top-level `network` keyword described in Table 6-3.)
`environment`	This property is used to define the environment variables that will be used when a container is created. In this example, `MYSQL_ROOT_PASSWORD` and `bind-address` variables are defined.

Run the command shown in Listing 6-8 in the ExampleApp folder to check that the changes you have made to the file can be processed.

Listing 6-8. Building the Application

```
docker-compose build
```

I am able to omit the name of the file because I am using the compose file name that Docker looks for by default. This command will produce output like this:

```
...
WARNING: Some networks were defined but are not used by any service: frontend
mysql uses an image, skipping
...
```

The warning about the unused networks has changed because the database container will be connected to the backend network. The other part of the output indicates that no action is needed for the mysql service at the moment because it is based on an existing image, which will be used to create and configure a container when the compose file is used to start the application.

Testing the Application Setup and Teardown

There is just enough configuration in the compose file to run a simple test. Run the command shown in Listing 6-9 in the ExampleApp folder to tell Docker Compose to process the compose file and start the application.

Listing 6-9. Running the Composed Application

```
docker-compose up
```

The docker-compose up command tells Docker to process the contents of the compose file and set up the volumes, networks, and containers that it specifies. Docker will pull any images that are required from the Docker Hub so they can be used to create a container.

Details of the setup process are shown in the command prompt, along with the output from the containers that are created. There is only one container at the moment, but others are added in the sections that follow. The first part of the output shows the configuration process that Docker Compose goes through.

```
...
WARNING: Some networks were defined but are not used by any service: frontend
Creating network "exampleapp_backend" with the default driver
Creating volume "exampleapp_productdata" with default driver
Creating exampleapp_mysql_1
Attaching to exampleapp_mysql_1
...
```

The name of the network, the volume, and the container that Docker creates are prefixed with example_. The network is called exampleapp_backend, the volume is called exampleapp_productdata, and the container is called example_mysql_1. (The _1 part of the MySQL container name refers to how Docker scales up applications described using compose files, which I describe in the "*Scaling the MVC Service*" section.)

The prefix is taken from the name of the directory that contains the compose file, and it ensures that different compose files can use the same names for networks, volumes, and containers without them conflicting when the application is started. (You can change the prefix used with the `-p` argument for the `docker-compose up` command, which I demonstrate in Chapter 8.)

Once the database has finished its initialization process, type `Control+C` to terminate the `docker-compose` command and stop the containers described in the compose file.

You can explore the effect of the compose file using the standard Docker commands. Run the `docker ps -a` command to see the containers that have been created. Run the `docker network ls` command to see the software-defined network (only one of the two networks specified in the compose file has been created because no container currently connects to the `front_end` network). Run the `docker volume ls` command to see the volumes that have been created.

Run the command shown in Listing 6-10 in the `ExampleApp` folder to remove the containers and networks that are described in the compose file (but not the volume, which persists unless you use the `-v` argument).

Listing 6-10. Removing the Networks and Containers

```
docker-compose down
```

Database Preparation and MVC Composition

In the previous chapter, I relied on the ASP.NET Core MVC application to ensure that the schema in the MySQL database was up-to-date by automatically applying the Entity Framework Core migrations during startup.

This approach is fine in development and for simple projects, but automatically applying database migrations can lead to data loss in production systems and should be done with caution. Entity Framework Core migrations are groups of SQL commands that alter the database schema to reflect changes in the MVC application's data model classes. If you remove a property from a model class, for example, the migration that reflects that change will drop a corresponding column from a table in the database. Automating any process that affects a production database is risky. I recommend you apply database migrations explicitly only as part of a controlled upgrade and not implicitly as part of the regular application startup.

That doesn't mean that database migrations have to be hand-typed into the command prompt as part of an upgrade. The approach that I take in this chapter uses the same ASP.NET Core project to create two different containers. One container will perform the database initialization and apply the Entity Framework Core migrations, while the other will run the ASP.NET Core MVC application.

Modifying the MVC Application

Rather than create a new .NET project to initialize the database, I am going to reconfigure the existing application so that it has two modes: database initialization and normal ASP.NET Core MVC service.

Modify the database context class, as shown in Listing 6-11, so that it can be instantiated using a parameterless constructor, allowing it to be used outside of the ASP.NET Core startup sequence.

Listing 6-11. Preparing for Initialization in the ProductDbContext.cs File in the ExampleApp/Models Folder

```
using Microsoft.EntityFrameworkCore;
using System;

namespace ExampleApp.Models {

    public class ProductDbContext : DbContext {

        public ProductDbContext() { }
```

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

        protected override void OnConfiguring(DbContextOptionsBuilder options) {

            var envs = Environment.GetEnvironmentVariables();

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

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

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

The `OnConfiguring` method will be called when a new database context is created and provides Entity Framework Core with the connection string that it needs to connect to the database. The configuration for the connection string is obtained from the application's environment variables, accessed through the `System.Envionment` class.

Make the changes shown in Listing 6-12 to the `Program` class, which is responsible for the startup of the application.

Listing 6-12. Preparing for Database Initialization in the Program.cs File in the ExampleApp Folder

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;

namespace ExampleApp {
    public class Program {
        public static void Main(string[] args) {
            var config = new ConfigurationBuilder()
                 .AddCommandLine(args)
              .AddEnvironmentVariables()
                 .Build();

            if ((config["INITDB"] ?? "false") == "true") {
                System.Console.WriteLine("Preparing Database...");
                Models.SeedData.EnsurePopulated(new Models.ProductDbContext());
                System.Console.WriteLine("Database Preparation Complete");
            } else {
                System.Console.WriteLine("Starting ASP.NET...");
                var host = new WebHostBuilder()
```

```
                .UseConfiguration(config)
                .UseKestrel()
                .UseContentRoot(Directory.GetCurrentDirectory())
                .UseIISIntegration()
                .UseStartup<Startup>()
                .Build();

                host.Run();
            }
        }
    }
}
```

If the configuration setting called `INITDB` is set to `true`, then the database will be initialized when the .NET runtime invokes the `Main` method. If the configuration setting is not `true` or is not defined, then the ASP.NET Core MVC application will be started instead. The value for the `INITDB` setting will be read from the command line and from the environment variables, which will allow the database to be easily upgraded from a command prompt and within a container.

Reading configuration data from the command line requires the addition of a NuGet package. Edit the `ExampleApp.csproj` file to make the addition shown in Listing 6-13.

Listing 6-13. Adding a Package 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" />
    <PackageReference Include="Microsoft.Extensions.Configuration.CommandLine"
    Version="1.1.1" />
  </ItemGroup>
</Project>
```

Finally, comment out the statement in the `Startup` class that was previously responsible for initializing the database, as shown in Listing 6-14.

Listing 6-14. Disabling the Initialization Call in the Startup.cs File in the ExampleApp Folder

```
...
public void Configure(IApplicationBuilder app,
    IHostingEnvironment env, ILoggerFactory loggerFactory) {

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

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

Describing the Database Initialization and MVC Services

When the application was set up step by step in Chapter 5, each component was allowed to go through its startup routine before the components that depended on it were started. This meant, for example, that MySQL was given the time to go through its lengthy first-start procedure before the MVC application was started, ensuring that the MySQL database was ready to start receiving SQL queries before the MVC application started making them.

Docker doesn't support this gradual model of initializing an application when a compose file is used. Docker has no insight into whether MySQL is ready to accept network connections and just starts one container after another, with the potential outcome that the MVC application will fail because it tries to make a request to a database that isn't ready.

To ensure that queries are not sent until the database is ready, I am going to use a package called `wait-for-it`, which waits until a TCP port is accepting connections. Run the command shown in Listing 6-15 in the `ExampleApp` folder to download the `wait-for-it` package using NPM so that it is installed in a folder called a folder called `node_modules`.

Listing 6-15. Downloading the wait-for-it Package

```
npm install wait-for-it.sh@1.0.0
```

Once the download is complete, apply the changes shown in Listing 6-16 to update the Docker file for the MVC application. These changes use the `wait-for-it` package to defer the startup of the .NET Core application until the database is ready to receive connections.

Listing 6-16. Waiting for the Database in the Dockerfile File in the ExampleApp Folder

```
FROM microsoft/aspnetcore:1.1.1

COPY dist /app

COPY node_modules/wait-for-it.sh/bin/wait-for-it /app/wait-for-it.sh

RUN chmod +x /app/wait-for-it.sh

WORKDIR /app

EXPOSE 80/tcp
```

```
ENV WAITHOST=mysql WAITPORT=3306

ENTRYPOINT ./wait-for-it.sh $WAITHOST:$WAITPORT --timeout=0 \
    && exec dotnet ExampleApp.dll
```

The host name and TCP port that will be checked by the `wait-for-it` package are specified using environment variables called `WAITHOST` and `WAITPORT`. Listing 6-17 adds services to the compose file that uses the new application configuration and the waiting feature to prepare the database and run the MVC application.

Listing 6-17. Describing the Database Initialization and MVC Application in the docker-compose.yml File

```
version: "3"

volumes:
  productdata:

networks:
  frontend:
  backend:

services:

  mysql:
    image: "mysql:8.0.0"
    volumes:
      - productdata:/var/lib/mysql
    networks:
      - backend
    environment:
      - MYSQL_ROOT_PASSWORD=mysecret
      - bind-address=0.0.0.0

  dbinit:
    build:
      context: .
      dockerfile: Dockerfile
    networks:
      - backend
    environment:
      - INITDB=true
      - DBHOST=mysql
    depends_on:
      - mysql

  mvc:
    build:
      context: .
      dockerfile: Dockerfile
    networks:
      - backend
      - frontend
```

```
    environment:
      - DBHOST=mysql
    depends_on:
      - mysql
```

The new services are called `dbinit` and `mvc`. I have taken a different approach for these services, providing Docker with the information it needs to build the images for the services, rather than specifying an image that it can pull from a repository, which is what I did for MySQL. Table 6-5 describes the configuration options used for these services.

Table 6-5. *The Configuration Properties for the Database Initialization and MVC Services*

Name	Description
`build`	This property denotes the start of the build section, which tells Docker how to create the image required for this service's containers.
`build/context`	This property defines the context directory that will be used to create the image. For this example, the content is the current directory, which will be the `ExampleApp` folder when the image is produced.
`build/dockerfile`	This property specifies the Docker file that will be used to create the image.
`networks`	This property specifies the software-defined networks that the containers will be connected to. The database initialization container will be connected to the `backend` network because it needs to communicate with the database. The MVC application containers will be connected to both networks. (The `networks` keyword used in a service has a different meaning than the top-level `network` keyword described in Table 6-3.)
`environment`	This property specifies the environment variables for the containers. For this example, they are used to select the initialization/MVC modes and to provide the name of the database container, which is used in the Entity Framework Core connection string.
`depends_on`	This property tells Docker the order in which containers must be created. In this example, the containers for the `dbinit` and `mvc` containers will be started after the `mysql` container.

Don't be misled by the `depends_on` configuration option, which tells Docker the order in which containers should be started. Docker will start the containers in the specified sequence but still won't wait until the applications they contain are initialized, which is why the `wait-for-it` package is required.

WAITING IN WINDOWS CONTAINERS

The `wait-for-it` package can be used only in Linux containers. To achieve a similar effect in Windows containers, create a file called `wait.ps1` in the `ExampleApp` folder with the following content:

```
Write-Host "Waiting for:" $env:WAITHOST $env:WAITPORT
do {
    Start-Sleep 1
} until(Test-NetConnection $env:WAITHOST -Port $env:WAITPORT `
    | Where-Object { $_.TcpTestSucceeded });
Write-Host "End of waiting."
```

This is a simple PowerShell script that waits for a TCP port to become available. Use the ADD command to include the script in a Docker file so that it can be used with the ENTRYPOINT command, like this:

```
FROM microsoft/dotnet:1.1.1-runtime-nanoserver

COPY dist /app

COPY wait.ps1 /app/wait.ps1

WORKDIR /app

EXPOSE 80/tcp

ENV ASPNETCORE_URLS http://+:80

ENV WAITHOST=mysql WAITPORT=3306

ENTRYPOINT powershell ./wait.ps1; dotnet ExampleApp.dll
```

The result is the same as using the wait-for-it package, although you may see some warnings displayed in the container output telling you that PowerShell modules cannot be found. These warnings can be ignored.

Composing the Load Balancer

The final service required in the application is the load balancer. I am going to use HAProxy again, which is the package that was used in Chapter 5, but using an image that is published by Docker. Listing 6-18 shows the changes to the docker-compose.yml file to describe the load balancer.

Listing 6-18. Describing the Load Balancer in the docker-compose.yml File in the ExampleApp Folder

```
version: "3"

volumes:
  productdata:

networks:
  frontend:
  backend:

services:

  mysql:
    image: "mysql:8.0.0"
```

```
    volumes:
      - productdata:/var/lib/mysql
    networks:
      - backend
    environment:
      - MYSQL_ROOT_PASSWORD=mysecret
      - bind-address=0.0.0.0

  dbinit:
    build:
      context: .
      dockerfile: Dockerfile
    networks:
      - backend
    environment:
      - INITDB=true
      - DBHOST=mysql
    depends_on:
      - mysql

  mvc:
    build:
      context: .
      dockerfile: Dockerfile
    networks:
      - backend
      - frontend
    environment:
      - DBHOST=mysql
    depends_on:
      - mysql

  loadbalancer:
    image: dockercloud/haproxy:1.2.1
    ports:
      - 3000:80
    links:
      - mvc
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    networks:
      - frontend
```

This version of HAProxy is set up to receive events from Docker as containers are created, started, stopped, and destroyed in order to generate configuration files for HAProxy to dynamically reflect changes in the application. Table 6-6 describes the configuration settings that are used to describe the load balancer service.

Table 6-6. *The Configuration Properties for the Load Balancer Service*

Name	Description
`image`	This property specifies the image that will be used for the load balancer container. In this example, the image is one provided by Docker.
`ports`	This property specifies the host port mappings that will be applied to the containers. In this example, requests sent to port 3000 on the host operating system will be directed to port 80 inside the container, which is the default port on which HAProxy listens for HTTP requests.
`links`	This property is used to provide HAProxy with the name of the service whose containers will receive HTTP requests, which is `mvc` in this example. The load balancer will respond automatically when a new container is created for the `mvc` service and start forwarding HTTP requests for it to process.
`volumes`	This property is used to specify the volumes used by the container. In this example, the `volumes` property is used to allow the load balancer access to the Docker runtime on the host operating system so that it can receive notifications when new containers are created in order to monitor for changes in the service specified by the `links` property.
`networks`	This property is used to specify the software-defined networks that the containers will be connected to. In this example, the load balancer will be connected to the `frontend` network, which will allow it to forward HTTP requests to the MVC application. (The `networks` keyword used in a service has a different meaning from the top-level `network` keyword described in Table 6-3.)

Running the Application

The compose file now contains descriptions of all the components required for the example application: a volume for the database files, two software-defined networks, a database container, a container that will prepare the database for its first use, a container for the MVC application, and a load balancer that will receive and distribute HTTP requests. In the sections that follow, I show you how to use these descriptions to run the application.

Processing the Compose File

Run the commands shown in Listing 6-19 in the `ExampleApp` folder to prepare the .NET Core project, ensuring that the required NuGet packages are installed and that the latest version of the project is published as a self-contained set of files in the `dist` folder.

Listing 6-19. Preparing the Application for Composition

```
dotnet restore
dotnet publish --framework netcoreapp1.1 --configuration Release --output dist
```

Run the command shown in Listing 6-20 in the `ExampleApp` folder to process the contents of the compose file.

Listing 6-20. Processing the Compose File

```
docker-compose build
```

When you run the `docker-compose build` command, Docker will generate new images for the `dbinit` and `mvc` services, as described in the compose file and that you can see if you run the `docker images` command. Docker is smart enough to know that these two services require the same content and generates one image with two tags, which you can see because they have the same ID.

Preparing the Database

Run the command shown in Listing 6-21 in the `ExampleApp` folder to start and prepare the database.

Listing 6-21. Starting and Preparing the Database

```
docker-compose up dbinit
```

The `docker-compose up` command can be used to selectively start services. This command specifies the `dbinit` service, and because of the `depends_on` setting in the compose file in Listing 6-18, Docker will also start the `mysql` service. When a service is specified as an argument to the `docker-compose up` command, only the output from that service is shown, which means you will see output from the database initialization container and not MySQL, like this:

```
...
Creating network "exampleapp_frontend" with the default driver
Creating network "exampleapp_backend" with the default driver
Creating exampleapp_mysql_1
Creating exampleapp_dbinit_1
Attaching to exampleapp_dbinit_1
dbinit_1        | wait-for-it.sh: waiting for mysql:3306 without a timeout
dbinit_1        | wait-for-it.sh: mysql:3306 is available after 10 seconds
dbinit_1        | Preparing Database...
dbinit_1        | Applying Migrations...
dbinit_1        | Created Seed Data...
dbinit_1        | Database Preparation Complete
exampleapp_dbinit_1 exited with code 0
...
```

The `wait-for-it` script in the `dbinit` container will wait until the database is ready to accept connections, at which point the Entity Framework Core migration is applied and the seed data is created. The `dbinit` container will exit once the preparation is complete.

Starting the MVC Application

Now that the database has been prepared, it is time to start the MVC application. Run the command shown in Listing 6-22 in the `ExampleApp` folder to start the MVC application.

Listing 6-22. Starting the MVC Application

```
docker-compose up mvc loadbalancer
```

This command tells Docker to start the `mvc` and `loadbalancer` services. To do that, Docker follows the chain of `depends_on` settings to determine the other services that must be started. In this case, the `mvc` service depends on the `mysql` service, for which there is already a container. Docker will automatically pull any images it requires, create the software-defined networks if they are not already present, and, finally, create and start the containers for each service.

Once the new containers have started, open a new browser tab and request the URL `http://localhost:3000`, which corresponds to the port mapping defined for the load balancer in Listing 6-18. The load balancer will forward the request over the `frontend` network to the MVC container, which queries the database over the `backend` network and uses the data it receives to generate the response shown in Figure 6-1. The response from the MVC application includes the ID of its container.

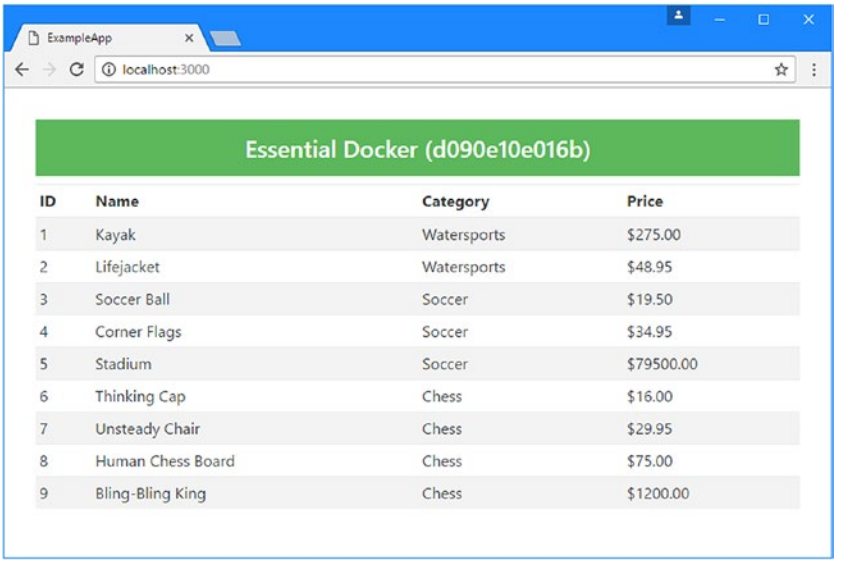

Figure 6-1. *Testing the example application*

Scaling the MVC Service

Compose files describe services for which containers are created. So far, there has been a one-to-one mapping between the descriptions contained in the compose file and the containers that have been created. Docker Compose has an important feature, which is that it will create and manage multiple containers from the same description, meaning that a service can be made up of more than one container, which can share the workload for the application.

When you ran the `docker-compose up` command in Listing 6-21, Docker created and started three containers: one database container from the `mysql` service, one MVC container from `mvc` service, and a load balancer container from the `loadbalancer` service. It can be hard to see in all of the output from MySQL, but Docker writes messages to the command prompt as it creates each component required for the application, like this:

```
...
exampleapp_mysql_1 is up-to-date
Creating exampleapp_mvc_1
Creating exampleapp_loadbalancer_1
Attaching to exampleapp_mvc_1, exampleapp_loadbalancer_1
...
```

The exampleapp_mysql_1 container was left over from the initialization process, and Docker knows that it doesn't need to be re-created. The exampleapp_mvc_1 and exampleapp_loadbalancer_1 containers had to be created because no containers for those services existed before the docker-compose up command was run. By the time the docker-compose up command completed, there was one container for each of the services specified and for the services named in the depends_on properties.

Docker assigns names to the container with the _1 suffix so that multiple instances of a container can be created without causing a name conflict. Use a second command prompt to run the command shown in Listing 6-23 in the ExampleApp folder.

Listing 6-23. Scaling Up the MVC Application Containers

```
docker-compose scale mvc=4
```

The docker-compose scale command is used to change the number of containers for a service. In this case, the command tells Docker that there should be four instances of the container described by the mvc service in the compose file. The output of the command is the messages that Docker displays as it creates the new containers.

```
...
Creating and starting exampleapp_mvc_2 ... done
Creating and starting exampleapp_mvc_3 ... done
Creating and starting exampleapp_mvc_4 ... done
...
```

You will also see the output from the MVC application displayed in the original command prompt as the containers are started and the ASP.NET Core runtime starts up.

Run the command shown in Listing 6-24 in the ExampleApp folder to confirm the state of the application.

Listing 6-24. Inspecting the Application

```
docker-compose ps
```

The docker-compose ps command shows the running containers that have been created from the compose file and will produce output like this:

```
Name                          Command                          State     Ports
-------------------------------------------------------------------------------------
exampleapp_dbinit_1           /bin/sh -c ./wait-for-it.s ...   Exit 0
exampleapp_loadbalancer_1     dockercloud-haproxy              Up        1936/tcp,
                                                                          443/tcp,
                                                                 0.0.0.0:3000->80/tcp
exampleapp_mvc_1              /bin/sh -c ./wait-for-it.s ...   Up          80/tcp
exampleapp_mvc_2              /bin/sh -c ./wait-for-it.s ...   Up          80/tcp
exampleapp_mvc_3              /bin/sh -c ./wait-for-it.s ...   Up          80/tcp
exampleapp_mvc_4              /bin/sh -c ./wait-for-it.s ...   Up          80/tcp
exampleapp_mysql_1            docker-entrypoint.sh mysqld      Up        3306/tcp
```

You can see that the dbinit container has exited, which was intentional after checking and preparing the database. There are six running containers: the database, four MVC containers, and the load balancer.

The image used for the load balancer service configures HAProxy to respond to changes in the number of containers in the service specified by the `link` configuration property in the compose file. This property was set to `mvc` in Listing 6-18, which means the load balancer automatically starts directing HTTP requests to `mvc` containers when they are created, without the need for any configuration changes.

Open a new browser tab and request `http://localhost:3000`. Reload the browser to send additional HTTP requests to the load balancer, which will be distributed between the MVC containers. The output in the command prompt used to start the `mvc` service will indicate which container has handled the request, and you will see different host names displayed in the responses, as shown in Figure 6-2.

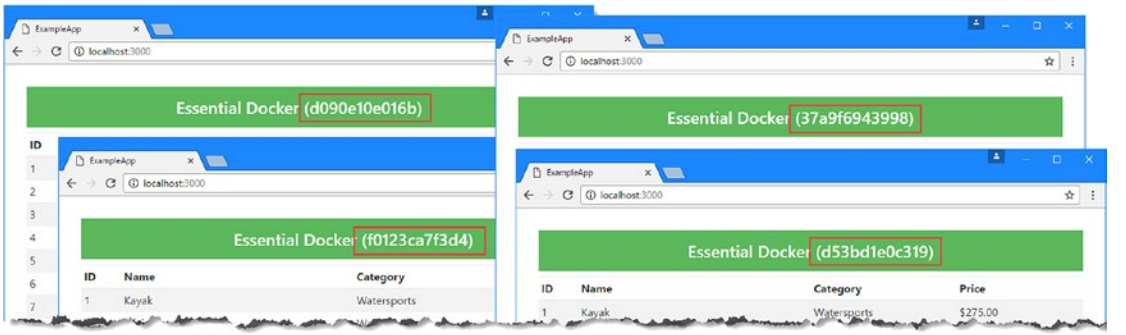

Figure 6-2. *Dynamically configuring the load balancer*

■ **Tip** Not all containers are suitable for scaling up. The MySQL container cannot be scaled up, for example, because the `mysql` service specifies a Docker volume for the data files and MySQL doesn't support two database processes sharing the same files. Containers that rely on port mappings to the host operating system, like the load balancer, can't be scaled up either because only the first container will be able to use the port.

Run the command shown in Listing 6-25 in the `ExampleApp` folder to scale down the number of MVC application containers to one.

Listing 6-25. Scaling Down the MVC Containers

```
docker-compose scale mvc=1
```

All the HTTP requests received by the load balancer will be directed to the remaining MVC container.

Stopping the Application

Run the command shown in Listing 6-26 in the `ExampleApp` folder to stop all the containers.

Listing 6-26. Stopping the Application Containers

```
docker-compose stop
```

Finally, run the command shown in Listing 6-27 in the ExampleApp folder to remove the containers, networks, and volumes. (Be careful when using the -v flag in production systems.)

Listing 6-27. Removing the Application Components

```
docker-compose down -v
```

Summary

In this chapter, I explained how a compose file can be used to describe a complex application, including the containers it requires, the volumes they depend on, and the software-defined networks that connect them. I demonstrated how a complex application can be brought to life using the docker-compose command and how the number of containers in an application can be scaled up and down as needed. In the next chapter, I explain how to use a Docker swarm, which allows a containerized application to be deployed across multiple servers that have been clustered together.