Running your microservice anywhere via Docker

In this lesson you will learn how Docker can help you run your microservices in any environment.

From the Dev Box to Production

Let's quickly recap the current setup of your local development environment. Let's also focus on one specific microservice, the Identity microservice, just to keep things simple.

The microservice is running on top of the ASP.NET Core Runtime, which provides all the services needed to run your .NET based microservices. And, that runtime, in turn, runs on top of whichever operating system you have installed in your box.

There are a series of dependencies, in the form of NuGet packages, that your microservices require, including the Play.Common library, the contracts libraries you created in this course, and several other open source libraries you have used in several places.

The Identity microservice also interacts with a Mongo database to store data and a RabbitMQ message broker to send and receive asynchronous messages, both running as Docker containers in your box.

Finally, the are several configuration values the Identity microservice needs to run properly, like the MongoDB and RabbitMQ host to use, and there are even secrets like the admin user password that are essential for the service to work properly.

Now, as you start thinking about moving your microservice from your Dev Box to a Production environment, you will need to start thinking about a few things like:

- How to prepare a different box for production purposes, with the correct versions of both the operating system and the ASP.NET Core runtime?
- How to bring in all the required dependencies into this production box?
- Once you have the box ready, how to take it to a place where others can reach it?
- Then, what database and message broker would your production microservice use?
- And lastly, how would you provide the correct configuration and secret values that make sense for a production environment?

Here's where Docker can certainly help us.

Docker to the rescue

Let's go back to your developer box and to your local instance of the Identity microservice. Instead of trying to figure out how to create a second box with all the requirements and settings needed to run your microservice, what you can do is create what we call a Dockerfile.

This is a simple text file where, via a series of instructions supported by Docker, you can declare the exact version of the operating system to use and the version of the .NET or ASP.NET Core Runtime that your microservice is built for.

You can declare how to bring in all the dependencies needed by your microservice, how to build the microservice itself and where in the file system to put all the resulting files, using the correct directory structure. And, of course, you can also specify here how to start the microservice process.

Once you have the dockerfile ready, you can use it in combination with the Docker engine that comes with your Docker Desktop installation to build what we call a Docker image. This Docker image contains everything needed to run your microservice, as described in the dockerfile.

By now you have some experience with docker images, since you have been using them to run MongoDB and RabbitMQ in your box, but as you can see, you can use dockerfiles to create your own images.

Now comes the question of how to get this docker image to a Production box. Don't worry too much about where this production box is, or how we made it available to the outside world, we'll get into those details in future modules.

Assuming there is a Production box somewhere, how do you get your image deployed there? Well, what you can do first is publish it, or push it to a container registry. The container registry is nothing more than storage for your images, usually somewhere in the cloud.

Also, the container registry can be public, so anyone can get access to your docker images, or it can be private, so you can only get access to it by presenting the appropriate credentials.

With the image available in the container registry, the Production box can now pull it down and create a Docker container. A Docker container is a sandboxed process, based on a Docker image, which is isolated from all the other processes on the box.

In this example, this docker container is in fact a running version of the Identity microservice, and it can run in the Production box with no additional installation or setup necessary only because of the Docker engine, which is also running here and is the exact same engine you have running in your Dev box.

The Production box can dynamically provide all the configuration values and secrets required by the microservice as environment variables.

Since your ASP.NET Core based microservice treats environment variables just as any other configuration source, it will be able to easily read them and use them to figure out details like which database server to connect to and which message broker to use to send and receive messages.

Also, notice that it is as easy to run one instance of your microservice container as it is to run multiple instances, so as your system needs to handle more load, the Production box will be able to scale the number of containers needed by your system to as many instances as resources are available in the box.

Docker containers bring in lots of benefits that where only a dream in the old era of virtual machines:

- They let you take advantage of your compute resources very efficiently, because they use only the fraction of CPU and RAM that your microservice really needs and they are layered in such a way that they only require a small amount of disk space.

- Containers start fast because they don’t need to start an entire operating system. They run as individual processes in the host operating system.

- They run in their own isolated environment, which ensures no other process can have detrimental effects in your running microservice.

- They can run anywhere, as long as the host has the same base operating system and the docker engine is up and running.

- And, as mentioned before, you can easily scale to multiple instances of your containers in a single host to handle the required load.

In the next lesson you will learn how to create your first dockerfile.