USING .NET CORE, DOCKER, AND KUBERNETES

SUCCINCTLY

BY MICHELE APONTE

SUCCINCTLY EBOOK SERIES

Syncfusion

Using .NET Core, Docker, and Kubernetes Succinctly

By
Michele Aponte

Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Tres Watkins, content development manager, Syncfusion, Inc.
Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The Succinctly series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

I was born in Naples, Italy, but as Sofia Villani Scicolone (a.k.a. Sophia Loren) and Carlo Perdersoli (a.k.a. Bud Spencer) both like to say: “I'm not Italian, I'm from Naples. It is different!”

Passionate programmer since 1993, I’ve worked as a Java, .NET, and JavaScript programmer for Italian software houses and IT consulting companies. In 2013, I founded the IT company Blexin, whose first aim is to help its customers to migrate from their old solutions to new technologies by offering them training, consulting, and software development.

I have always believed in the value of sharing. For this reason, in December 2008 I founded DotNetCampania, a Microsoft user group for which I organized many free regional conferences. Microsoft recognized me with the Microsoft Valuable Professional (MVP) award for these activities.

I am an active member of a local meetup named DevDay that gathers all the developers from my region using any technology. I am also a speaker at some of the most important conferences on Microsoft and JavaScript technologies.

Let’s connect!

You can connect with me on LinkedIn, Twitter, Facebook, and Instagram, or send me an email. I will be happy to answer all questions about this book and receive your feedback to improve myself:

- LinkedIn: https://www.linkedin.com/in/michele-aponte-1996038
- Twitter: https://twitter.com/apomic80
- Facebook: https://www.facebook.com/apomic80
- Instagram: https://www.instagram.com/apontemichele
- Email: michele.aponte@blexin.com

Acknowledgments

Writing a book is hard work that requires much time and inevitably involves the lives of those closest to you. I want to thank my life partner, Raffaella, for her support for this project and her patience with me. I also want to thank my old friend, whose name is Raffaella as well, for the help she gave me in revising my English.

A special thanks goes also to all the members of Blexin, my company, who are the best team one can have and with whom I can experiment every day with the technologies you find described in this book.

A big thank you to Lorenzo Barbieri too, with whom for years I have shared my love for the community, technology, and food! Download and read his *Succinctly* book on public speaking, if you haven't done so yet. If today I am a better speaker, I certainly owe it to him! In the end, many thanks to the fantastic Syncfusion team that supported me during this project.

I want to dedicate this book to my son Francesco Paolo, born only seven months ago. I hope that one day he can read it and be as proud of me as I am of every smile that he makes when he catches my eye.

Chapter 1 ASP.NET and Docker Together

How it all began

With the .NET Core framework, Microsoft has enabled all .NET developers to cross-platform develop, to the amazement of some, and the joy of others. The first question is: why? Today it's clear that the focus of the leading software company of the world has changed: the future of software is the cloud, and the priority is to provide a cloud platform to everyone.

Thanks to Azure, Microsoft is today one of the three leading cloud-services providers, with a complete platform that allows developers to do everything they need, using any language or technology stack. One of the most interesting collateral effects of this change, especially in web development, is the break with the Windows ecosystem, like Microsoft's Internet Information Services (IIS), the application server that's provided for free with all the versions of Windows Server.

We have gone from hosting our ASP.NET application on Windows with IIS to the infinite number of possibilities offered by the open-source market. Consequently, terms like *container* and *Docker*, unknown to Microsoft programmers until now, have recently become essential in technical discussions about the development, distribution, and management of our applications. Moreover, the idea that an integrated development environment (IDE) such as Visual Studio can hide the complexity of using these technologies is disappearing, which is powering the CLI (Command Line Interface) and requiring us to be familiar with the command line.

However, why should I change my working method? Why should I choose containers for my applications? What value do containers add to my work? Sorry for the spoiler, but to have a microservices architecture is not the answer: if you have a microservices architecture, you probably need container technology to make it sustainable. This doesn't mean you can't take advantage of containers if you are not Netflix or Amazon.

Containers can solve some of programmers' everyday problems, such as configuring the development environment or recreating a production problem. Often, containers can help you overcome the fear of not being production-ready, because they can make it simple to restore a specific version of your application in case of problems. The automation of the release of your updates to the various stages of your development pipeline is much easier with containers, and all the tasks that you can automate give you more time to spend on business problems instead of technical problems.

As with most useful things, containers also have negative sides. By their nature, they tend to grow in number with the applications, and the communication among them can become hard to manage. In the passage from development to production, it could be crucial to consider, for example, how many instances of a single container can be running based on the state of the system, or where I store my data.

Nevertheless, when summarizing these questions, we can reconduct all the main difficulties to a problem of management, where configuration simplicity is the key for the right choice. If I could set up in a simple way the condition I want the system to be in, and if I could easily configure where to store my passwords safely, or where to save my data, I would probably solve my main problems.

If my system is a collection of containers, the main challenge is managing their lifecycles and the communication among them. Technically, I need an orchestrator, and in my opinion, the best choice today is Kubernetes. Let's see why.

Execute .NET Core application with Docker

As I've already said, a container is a good solution for both the development and release of an application. Until now, you probably developed and tested your ASP.NET applications on your machine. When you finished, or it was time to release your work, perhaps you published your application from Visual Studio to an FTP storage, a local folder copied next to the server, or a remote publishing endpoint on-premise or in a cloud. How many times have you found that it doesn't all work the first time? And that similar problems also occurred in later releases?

A possible solution is the development of an environment similar to the production, but this option requires the installation of all dependencies on your machine. Many times, it can be more productive to create a virtual machine because in many cases even the production environment is a collection of virtual machines. The main problem with a virtual machine is that it requires the installation of the operating system to work, consuming more CPU and memory than working on your host machine. Often, you would need more than one virtual machine, but other CPU and memory consumption would make your development and testing experience very stressful.

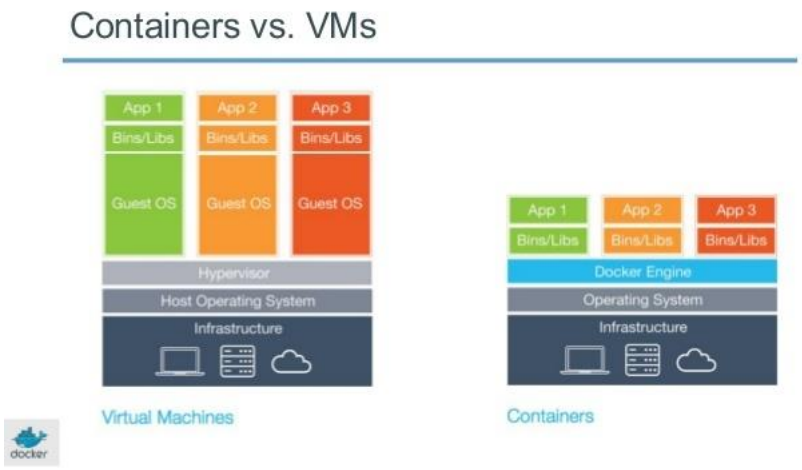

Figure 1: Virtual Machines vs. Containers, Source: Docker

A container is a solution to all these problems because, contrary to a virtual machine, you don't need to replicate the operating system—you can share the host operating system with your container, so you only have to add the files to run the applications.

It all starts with the creation of an image that contains your code and the libraries needed to run your application. Remember that an image is always read-only, and it can be executed to become a container. You can think of an image as a C# class that, when instantiated, becomes an object in memory with the properties defined in the class, but with its own values. In the same way, you can see a container like an instance of an image, where the properties are the files used to create the image itself.

To start working with containers, we need to choose a container technology, and Docker is surely the most famous and commonly used. The first step is to download the Docker tools for your operating system and install them. You can download the community edition for free here, and it is perfect for development purposes. The download requires free registration to Docker Hub, the largest registry of free images available in the open-source community. We will use it in our examples.

The setup program installs all you need to work with Docker: the Docker Engine, a client-server application that interacts with our installation. The Docker Engine is composed of a daemon process called Docker daemon, through which we can communicate by a REST API or a Command Line Interface (CLI). Docker Engine also contains other components that we will discover step by step when we need them.

To be sure that everything will work fine after the installation, we can run the following command in the terminal window.

Code Listing 1: Command to show executing Docker version

```
docker --version
```

The terminal should respond with the installed version of Docker, like in the following image.

```
Micheles-MBP:~ micheleaponte$ docker --version
Docker version 18.09.1, build 4c52b90
Micheles-MBP:~ micheleaponte$
```

Figure 2: Output of the docker --version *command*

If you have an error instead, something went wrong. Thanks to this first command, we can analyze the communication process between our Docker CLI and the Docker daemon. But the first question is: where is the Docker daemon executed? Containers were born on Linux, so if we use a MacOS or Windows to run Docker commands, we need a virtual machine with a Linux-based operating system to execute the Docker daemon and the containers.

If you are using a Mac, the Docker tools set up for us an Alpine Linux distribution on top of an Xhyve virtual machine. Xhyve is a lightweight hypervisor built on top of OS X's Hypervisor framework that runs virtual machines in user spaces. If you are using Windows instead, you have two options: use Linux Container from a Hyper-V virtual machine or, with Windows 10 or Windows Server 2016, use Windows containers.

It's true that containers were born on Linux, but as a consequence of the interest in this technology, Microsoft decided to create native Windows support for containers on its latest operating systems. This choice opens exciting scenarios, but it is essential to understand that containers share the operating system of the host. For this reason, we can only run Windows-compatible applications on Windows containers, and only Linux-compatible applications can run in a Linux container. Therefore, in Windows 10 or Windows Server 2016, you can choose (during or after installation) to use a Linux or a Windows container. In the first case, you will display in Hyper-V Manager a virtual machine named MobyLinuxVM, an Alpine Linux distribution configured with Hyper-V integration tools.

In the examples offered, you can use either Windows or Linux containers because .NET Core is cross-platform, and usually products like Redis, Mongo, and Elasticsearch are available for both operating systems. Even SQL Server 2017 is available for Linux and Windows, so feel free to use your favorite one.

It's time to run our first .NET Core container. First, you need to download an image from Docker Hub with a .NET Core example application. Microsoft released an image for this scope: **`microsoft/dotnet-samples`**. You can see all the images downloaded on your Docker machine with either of the following commands.

Code Listing 2: Commands to list docker images

```
docker image ls
docker images
```

For each image, you can display the repository name, a tag that usually specifies the repository version, the image ID, the creation date, and the size. To download the latest version of the **`microsoft/dotnet-samples`** image, you can execute the following command.

Code Listing 3: Command to pull .NET samples image

```
docker pull microsoft/dotnet-samples
```

If you want a specific version of the image, for example, the version that contains only the ASP.NET Core sample, you can use the following command.

Code Listing 4: Command to pull ASP.NET Core sample image

```
docker pull microsoft/dotnet-samples:aspnetapp
```

If you list the images, you can now see the downloaded images.

```
MacBook-Pro-di-Michele:~ micheleaponte$ docker image ls
REPOSITORY                          TAG                 IMAGE ID            CREATED             SIZE
microsoft/dotnet-samples            aspnetapp           d20323c39641        7 days ago          263MB
microsoft/dotnet-samples            latest              9c22d06482c2        7 days ago          180MB
```

Figure 3: Downloaded images

To create and run a container from the downloaded image, we can execute the command **`docker run`** followed by the image identifier (you can also use the first digits of the ID).

Code Listing 5: Command to run the .NET Core sample container

```
docker run 9c22
```

If it all works fine, you see the message **`Hello from .NET Core!`** in the terminal. What has happened? A Docker container has been created from the image with the ID **`9c22`** and it has been run. A container can be in one of the following states:

- **`created`**: It's been created, but not yet started.
- **`restarting`**: It's in the process of being restarted.
- **`running`**: It is executing.
- **`paused`**: Its execution is suspended.
- **`exited`**: It's been run and has completed its job.
- **`dead`**: The daemon tried and failed to stop.

To show all containers in a running state, you can execute the following command.

Code Listing 6: Command to list running containers

```
docker ps
```

As you can see, the container was just created; it is not displayed. This happens because the container is not in a running state, as you can verify by executing the command to show the containers in all statuses.

Code Listing 7: Command to list all containers

```
docker ps -a
```

The status of our container is exited. Why? If you display the code used to create the container, you can see that we are talking about a simple console application that terminates its execution at the end of the **`main`** method. So, at the end of process execution, the container was terminated. This behavior may seem weird if you usually use virtual machines, because a virtual machine remains in an execution state if your program has been executed. Obviously, the operating system running in the virtual machine doesn't stop its execution. But a container is not a virtual machine—it doesn't contain the operating system, just your code, and if your code has completed its job, the container terminates.

Try now to execute the image **`d203`** (the ASP.NET application).

Code Listing 8: Command to run the ASP.NET Core sample container

```
docker run d203
```

This container starts an ASP.NET Core application, so the requested listener remains in waiting for the HTTP request.

```
MacBook-Pro-di-Michele:~ micheleaponte$ docker run d203
warn: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[35]
      No XML encryptor configured. Key {1b7c7fd1-5eb2-49ee-b94f-02d82b49a495} may be persisted to storage in
unencrypted form.
Hosting environment: Production
Content root path: /app
Now listening on: http://[::]:80
Application started. Press Ctrl+C to shut down.
```

Figure 4: Output of the ASP.NET Core sample container execution

If you now list all the running containers (**`docker ps`**) from another terminal window, you can see the container that's just launched.

```
MacBook-Pro-di-Michele:~ micheleaponte$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
4afc8fa55569        d203                "dotnet aspnetapp.dll"   3 minutes ago       Up 3 minutes                            condescending_blackwell
```

Figure 5: Output of the docker ps command after ASP.NET Core container runs

Press Ctrl+C on your keyboard: the process ends, and the container ends with it. Open the browser and navigate to **http://localhost**. You would expect to see the ASP.NET Core application response, but you will receive a connection refused error. This error occurs because the ASP.NET Core listener waits for requests on port 80 in the container, but, by default, your container doesn't redirect the external request to its local host. You have to specify this redirection when starting the container, using the **`-p`** options. For example, we can redirect a request from the external port 8080 to the internal port 80 with the execution of the following command.

Code Listing 9: Command to map port when running a Docker container

```
docker run -p 8080:80 d203
```

If you open the browser and navigate to http://localhost:8080, you can see the default template of an ASP.NET MVC Core application.

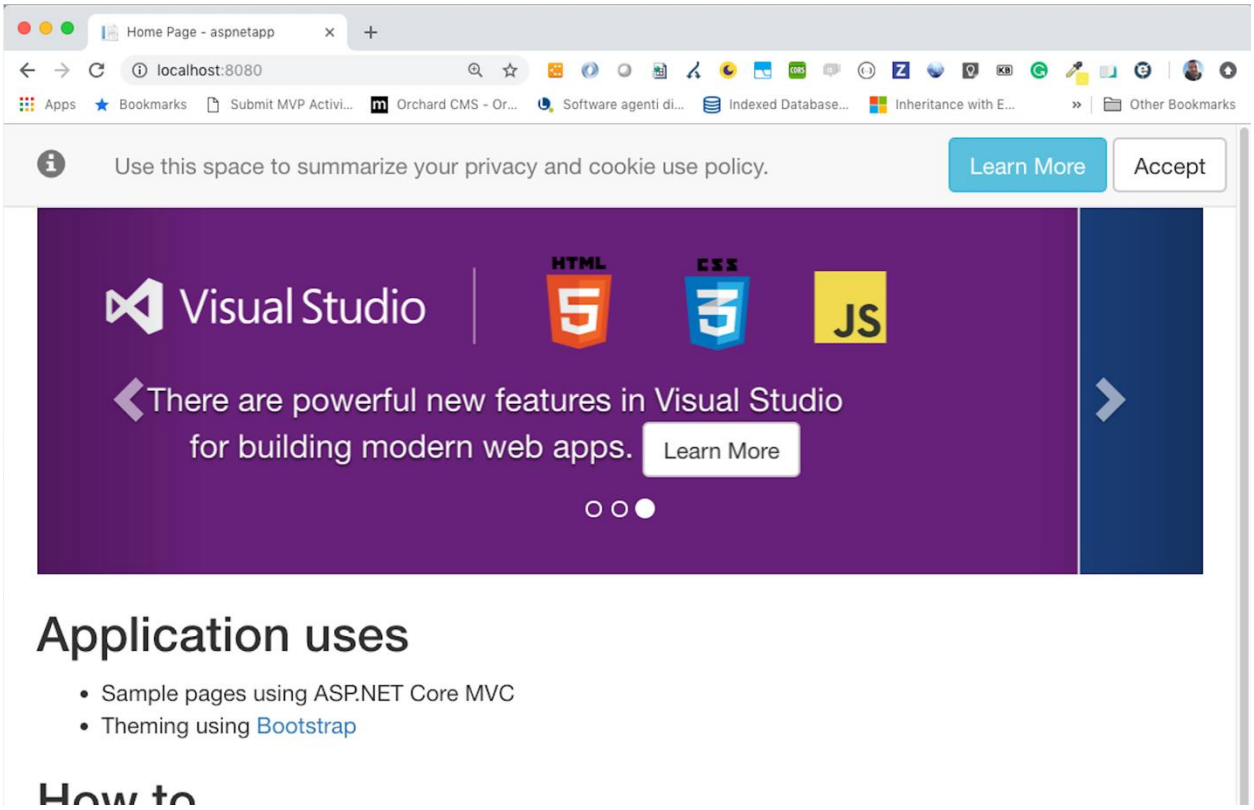

Figure 6: The sample ASP.NET Core application run from a Docker container

By default, the command **`docker run`** creates and executes a new container every time. So, if you have already created the container with a previous run, you can use the following command to start the container.

Code Listing 10: Command to start a Docker container

```
docker start <container-id>
```

You can retrieve the container ID from the container list (**`docker ps -a`**). As an alternative, when you run a container, you can also specify a name that is assigned by default randomly (in the previous example, the name **`condescending_clackwell`** is assigned). To create a container with a name, you can use the options **`--name`**.

Code Listing 11: Command to name a container when it is running

```
docker run -p 8080:80 --name mvcapp d203
```

You can then stop and start the container using the **`mvcapp`** name instead of the container ID.

Code Listing 12: Commands to stop and start a container by its name

```
docker stop mvcapp
```

```
docker start mvcapp
```

The name must be unique; otherwise, the Docker daemon will respond with an error. If you want to create a container without running it, you can use the command **`docker create`** instead of **`docker run`**.

Code Listing 13: Command to create a container without running it

```
docker create -p 8080:80 --name mvcapp2 d203
```

To remove the container, you can use the **`docker rm`** command with the container name or ID.

Code Listing 14: Command to remove a container by its name

```
docker rm mvcapp2
```

Another very useful option for the **`run`** command, when working with a container that doesn't terminate, is **`-d`**.

Code Listing 15: Command to run a container in detached mode

```
docker run -d -p 8080:80 --name mvcapp2 d203
```

With this option, the container is created and is running in the background, printing in the terminal window the ID of the created container so that you can run other commands in the same window. The command **`docker start`** runs the container in background mode by default. If you want the interactive mode, you can use **`docker start -i mvcapp`**.

Chapter 2 Create Your Application with Docker

Develop your ASP.NET Core application using Docker

Now that we've started an existent .NET Core application with Docker, it's time to create our image based on our code. Assuming that you're already familiar with .NET Core basics, we will now concentrate on how to create a development image to execute our application and sync the source code with the container file system.

If you are new to the .NET Core CLI, basically you need to know these commands:

- **`dotnet new`**: Creates a new project from the specified template. If you want to create an MVC application, you can use **`dotnet new mvc`**.
- **`dotnet restore`**: Restores all the NuGet dependencies of our application.
- **`dotnet build`**: Builds our project.
- **`dotnet run`**: Executes our project.
- **`dotnet watch run`**: Runs our project and watches for file changes to rebuild and re-run it.
- **`dotnet publish`**: Creates a deployable build (the .dll) of our project.

You can work on Windows or Mac with your favorite editor/IDE. For our examples, we will use a Mac with Visual Studio Code, but all the instructions are also available for Windows (you will find the corresponding Windows command in parenthesis if it's different from the Mac command).

You can download and install both Visual Studio Code and .NET Core for free from the official websites:

- Visual Studio Code
- .NET Core

The first step is the creation of a new web application using .NET Core. With the .NET CLI (Command Line Interface), we can add the project files of a new web project to a folder using the following commands in the terminal window.

Code Listing 16: Create a new ASP.NET Core application

```
mkdir -p myapp/frontend (for Windows: mkdir myapp\frontend)
cd myapp/frontend
donet new mvc
```

To ensure that everything is working fine, you can execute the **`run`** command.

Code Listing 17: Command to run a .NET Core application

```
dotnet run
```

You will display the following result.

```
Micheles-MacBook-Pro:frontend micheleaponte$ dotnet run
: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[0]
      User profile is available. Using '/Users/micheleaponte/.aspnet/Dat
st.
Hosting environment: Development
Content root path: /Users/micheleaponte/myapp/frontend
Now listening on: https://localhost:5001
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

Figure 7: ASP.NET Core application running output

Perfect! Press Ctrl+C to interrupt the execution and open the project with Visual Studio Code (you can type **`code .`** in the terminal window to open a new instance of the editor from the current folder). To create a Docker image, we need to describe, in a file named **`Dockerfile`**, all the operations that we want to execute to obtain the final result. Now, create a new file in the current folder and name it **`Dockerfile`**, and let's go script the needed operations.

We can now install the Visual Studio Code extension named Docker by Microsoft, which help us to edit the Docker files. This plug-in adds syntax, highlighting, and complete code in all Docker configuration files, including **`Dockerfile`**.

The creation of a new image starts from an existent base image, which has been chosen depending on what you want to do. In our example, we need to compile a .NET Core application, so we need an image that contains the .NET Core SDK. Microsoft provides the image **`microsoft/dotnet`** with multiple versions that we can select by the tag. For our aim, the best image is **`microsoft/dotnet:sdk`**, which contains all the libraries and tools to build a .NET Core application. The first command of our script is the following.

Code Listing 18: Dockerfile FROM instruction

```
FROM microsoft/dotnet:sdk
```

The keyword **`FROM`** instructs the Docker daemon to use the specified image to create a new image. If the image is not present locally, it is downloaded. Now we can execute all the operations we need to copy and build our application. As a preliminary step, we can create a new directory in the new image with the command **`RUN`**.

Code Listing 19: Dockerfile RUN instruction

```
RUN mkdir app
```

You can use **RUN** for all the operations that you want to execute on the file system during the new image creation. You can specify a working directory on the container to execute all forward commands based on a specific directory. In our case, we can use the directory just created.

Code Listing 20: Dockerfile WORKDIR instruction

```
WORKDIR /app
```

From this moment on, all commands are executed from the folder app. Now we can copy our source code under the image, using the **COPY** command.

Code Listing 21: Dockerfile COPY instruction

```
COPY . .
```

The **COPY** command requires us to specify source and target so that, with the specified command, we are copying all the content of the current directory in the app directory of the new image. Now we need to restore all project dependencies in the image.

Code Listing 22: Execute .NET Core dependencies restore from a Dockerfile

```
RUN dotnet restore
```

Finally, when a container is created and running, we want to execute our code. Instead of **RUN**, which executes a command on the image, we need the **CMD** command, which is executed as the container starts.

Code Listing 23: Dockerfile CMD instruction

```
CMD dotnet watch run
```

We are ready to create our image, but we need an optimization first. The **COPY** command copies all the contents of the current folder, including the bin and obj folders. We can exclude these folders using a **Dockerignore** file. If you are familiar with GIT, the concept is the same: exclude some folders from the elaboration. You can then create a file named **.dockerignore** and add the following rows.

Code Listing 24: Dockerignore file content for .NET Core application

```
bin/
obj/
```

Now when we create the image, the **bin** and **obj** folders will be excluded from the copy. It's time to create our image. The command to execute is in Code Listing 25.

Code Listing 25: Command to build a Docker image

```
docker build -t frontend .
```

The result is similar to the following.

```
Micheles-MacBook-Pro:frontend micheleaponte$ docker build -t frontend .
Sending build context to Docker daemon  11.78kB
Step 1/6 : FROM microsoft/dotnet:sdk
 ---> 389d91a8617b
Step 2/6 : RUN mkdir app
 ---> Running in 62004d21c1bc
Removing intermediate container 62004d21c1bc
 ---> a86707506cc2
Step 3/6 : WORKDIR /app
 ---> Running in 05181ff78042
Removing intermediate container 05181ff78042
 ---> f6be862be06d
Step 4/6 : COPY . .
 ---> 089003db2a6c
Step 5/6 : RUN dotnet restore
 ---> Running in a3c6d9175ef5
  Restoring packages for /app/frontend.csproj...
  Generating MSBuild file /app/obj/frontend.csproj.nuget.g.props.
  Generating MSBuild file /app/obj/frontend.csproj.nuget.g.targets.
  Restore completed in 664.83 ms for /app/frontend.csproj.
Removing intermediate container a3c6d9175ef5
 ---> 753618f20b82
Step 6/6 : CMD dotnet watch run
 ---> Running in 4ff5c76b6dc1
Removing intermediate container 4ff5c76b6dc1
 ---> 5beb406143d0
Successfully built 5beb406143d0
Successfully tagged frontend:latest
```

Figure 8: Docker build command output

If you examine the terminal output, you can see that the command creates our image in six steps, one for each **`Dockerfile`** instruction. After each step, you can see the log **`Removing intermediate container`**, which reminds us that an image is built from a base image that creates other layers over it. So, after each **`Dockerfile`** instruction, a new container is constructed, and the instruction is executed on it and immediately removed. You can see the new image executing the command **`docker images`** in Figure 9.

```
REPOSITORY                TAG                 IMAGE ID            CREATED             SIZE
frontend                  latest              5beb406143d0        16 minutes ago      1.74GB
microsoft/dotnet          sdk                 389d91a8617b        2 weeks ago         1.74GB
microsoft/dotnet-samples  aspnetapp           d20323c39641        3 weeks ago         263MB
microsoft/dotnet-samples  latest              9c22d06482c2        3 weeks ago         180MB
```

Figure 9: The new Docker image created

Since we have not specified a tag, **`latest`** is attributed to our image. If now you execute the **`docker run -p 5000:5000 -p 5001:5001 -t frontend`** command, a new container is created and run. The option **`-t`** allocates a pseudo-TTY. In our case, it is useful to enable Ctrl+C to stop the Kestrel listening.

```
Micheles-MacBook-Pro:frontend micheleaponte$ docker run -p 5000:5000 -p 5001:5001 frontend
watch : Polling file watcher is enabled
watch : Started
info: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[0]
      User profile is available. Using '/root/.aspnet/DataProtection-Keys' as key repository; keys
info: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[58]
      Creating key {e6889836-e357-4571-9373-766164e401d7} with creation date 2019-01-05 14:38:59Z,
expiration date 2019-04-05 14:38:59Z.
warn: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[35]
      No XML encryptor configured. Key {e6889836-e357-4571-9373-766164e401d7} may be persisted to s
info: Microsoft.AspNetCore.DataProtection.Repositories.FileSystemXmlRepository[39]
      Writing data to file '/root/.aspnet/DataProtection-Keys/key-e6889836-e357-4571-9373-766164e40
warn: Microsoft.AspNetCore.Server.Kestrel[0]
      Unable to bind to https://localhost:5001 on the IPv6 loopback interface: 'Cannot assign reque
warn: Microsoft.AspNetCore.Server.Kestrel[0]
      Unable to bind to http://localhost:5000 on the IPv6 loopback interface: 'Cannot assign reques
Hosting environment: Development
Content root path: /app
Now listening on: https://localhost:5001
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

Figure 10: Docker container creation from the new image

However, if you open the browser and navigate to **localhost:5000**, the application is not reachable. To understand why this happens, you have to know how Docker networking works. Without entering into this complex topic, you can host the ASP.NET Core application on the 0.0.0.0 IP address, which permits you to receive the requests from any IPv4-host at all. There are several ways to do this. In the **`launchSettings.json`**, located in the **`Properties`** folders, you can change the **`applicationUrl`** property from **`localhost`** to **`0.0.0.0`**.

Code Listing 26: ApplicationUrl configuration from launchSettings.json

```
"frontend": {
      "commandName": "Project",
      "launchBrowser": true,
      "applicationUrl": "https://0.0.0.0:5001;http://0.0.0.0:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
 }
```

As an alternative, you can add the **`.UseUrls()`** configuration method before the **`.UseStartup()`** in the **`CreateWebHostBuilder`** method, located in the **`Program.cs`** file.

Code Listing 27: UseUrls methods in web host creation

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseUrls("http://0.0.0.0:5000;https://0.0.0.0:5001")
                .UseStartup<Startup>();
```

My preferred option is the use of the **`--urls`** option in the **`run`** command, because it moves this fix to the Docker container execution that causes the drawback. Let's modify the last instruction of our **`Dockerfile`** in this way.

Code Listing 28: The option --urls of the command dotnet run

```
CMD dotnet watch run --urls "http://0.0.0.0:5000;https://0.0.0.0:5001"
```

Now, rebuild the image with the same command used previously. As you can see, the build process is faster than the first time because, during the build, Docker caches the operations and reuses the result to improve the performance. If you don't use the cache, it's possible to specify the **`--no-cache`** option. When running the container from this new image, it all works fine.

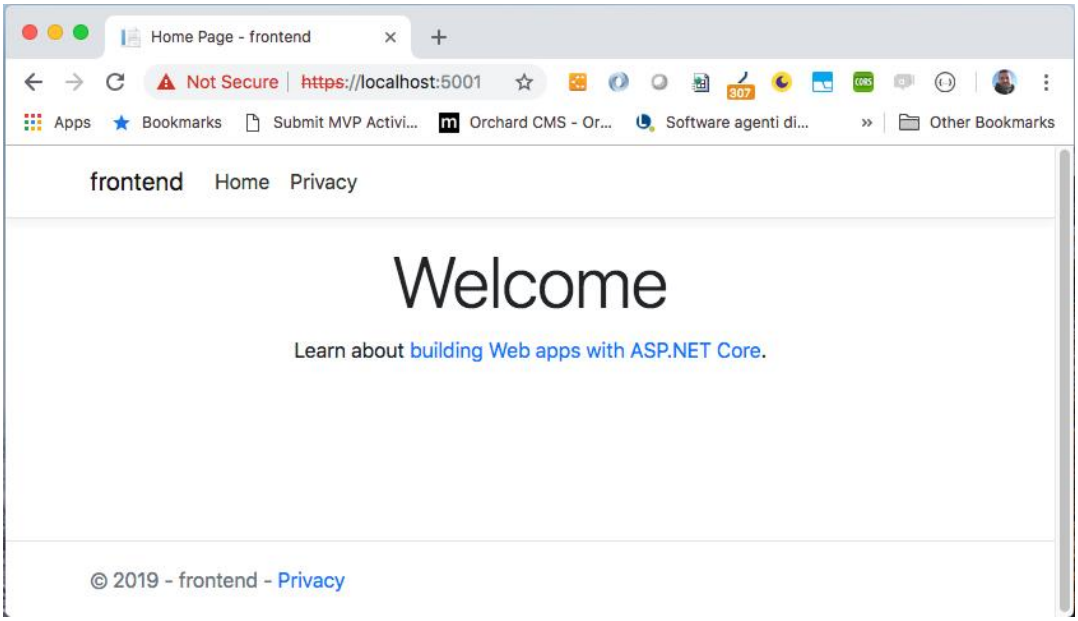

Figure 11: Our web application executed from Docker

We want to use this image to develop our application, so if we edit the source files, we would like for the container to rebuild and re-execute the modified code. The **`watch`** parameter of the **`dotnet watch run`** command does precisely this, but if we execute the code in our container, it watches only the file copied in the container.

To solve this problem, we have to create a Docker volume that permits us to share a folder between the host machine and the container. Not all the files of our project should be shared with the container, such as the **`bin`** and **`obj`** folders. You can specify the **`dotnet`** command to generate these folders in a specific location, but I prefer to move all the files that I want to share with the container into a specified folder. So, in my **`frontend`** project, I create an **`src`** folder and move into it the **`Controllers`**, **`Models`**, **`Views`**, and **`wwwroot`** folders, together with the **`Program.cs`** and **`Startup.cs`** files.

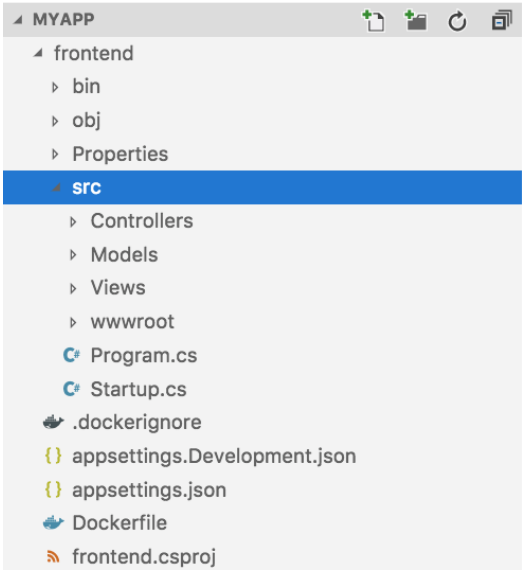

Figure 12: Application reorganization structure

If your ASP.NET Core application serves views and static files, you need to specify that the **`Views`** and **`wwwroot`** folders are now in the **`src`** folder. The fastest way is to change the content root of the application in the **`Program.cs`**.

Code Listing 29: Set a custom content root when creating web host

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
      WebHost.CreateDefaultBuilder(args)
     .UseContentRoot(
          Path.Combine(Directory.GetCurrentDirectory(), "src"))
     .UseStartup<Startup>();
```

This change is not necessary if your application only serves API. Now we are ready to rebuild our development image (execute the **`docker build`** command again) and run the container as follows.

Code Listing 30: Docker container run with volume

```
docker run -p:5000:5000 -p:5001:5001 -v $(pwd)/src:/app/src -t frontend
```

The option **`-v`** creates a shared volume between the local **`src`** folder and the **`src`** folder in the container. The **`$(pwd)`** instruction returns the current path, which we used to create the needed absolute source path. The destination path must be absolute (**`/app/src`**, not just **`/src`**, even if the app is the current working directory). Now if you change something in the local **`src`** folder, the same changes are available in the container, which is very useful during development!

Add containers to your project

Now we are ready to add other containers to our project. For example, we can add a database like SQL Server or MongoDB to save our data, or add a cache service like Redis to improve the performance of the read data operations. We can add advanced search functionality with an Elasticsearch container and decouple communication using a message broker like RabbitMQ. We have endless possibilities, thanks to Docker.

Is this too complicated? Let's take it one step at a time. Suppose we want to add SQL Server to our project. The good news is that SQL Server is also available for Linux, starting from the 2017 edition, so we can use it with both the Windows and Linux containers. We can download the correct image using the following command.

Code Listing 31: SQL Server Docker image pull

```
docker pull mcr.microsoft.com/mssql/server
```

To run this image, we need to specify some parameters to configure the SQL Server instance correctly:

- **ACCEPT_EULA=Y**: Accept the end-user license agreement.
- **SA_PASSWORD=<your_strong_password>**: Specify the password for SA user that must be at least eight characters including uppercase letters, lowercase letters, base-10 digits, and/or non-alphanumeric symbols.
- **MSSQL_PID=<your_product_id | edition_name>**: Specify which edition we want to use; the default value is the Developer edition.

To specify an argument with the **docker run** command, you can use the **-e** option. We can add a custom name to the container to simplify the connection of the application with the database. To do this, we can use the option **--name**, like in the following command.

Code Listing 32: SQL Server Docker container run

```
docker run -e 'ACCEPT_EULA=Y' -e 'SA_PASSWORD=Password_123' -e
'MSSQL_PID=Express' -p 1433:1433 --name sqlserver
mcr.microsoft.com/mssql/server
```

Now we have a SQL Server container named **sqlserver**, which exposes an instance of SQL Server Express Edition on port 1433. We can connect to this container with the user **SA** using the password **Password_123**.

To use SQL Server with the ASP.NET Core application, we need to connect our development container to the **sqlserver** container. To do this, we can use the option **–link**.

Code Listing 33: Front-end Docker container run with the link to the database container

```
docker run -p:5000:5000 -p:5001:5001 -v $(pwd)/src:/app/src -t --link
sqlserver --name frontend  frontend
```

Now you can use **sqlserver** as a server in the connection string block of the **appsettings** configuration file.

Code Listing 34: SQL Server connection string

```
"ConnectionStrings": {
    "MyAppDbContext": "Server=sqlserver; Database=myapp; User=sa;
Password=Password_123; MultipleActiveResultSets=true"
}
```

We can use this connection string with Entity Framework Core and add the following rows to the **ConfigureServices** method of the **Startup** class.

Code Listing 35: Entity Framework Core configuration

```
services.AddDbContext<MyAppDbContext>(o =>
    o.UseSqlServer(Configuration.GetConnectionString("MyAppDbContext"));
```

For our example, we can add a basic **Customer** class to the model, with some properties and their constraints, using data annotations.

Code Listing 36: The Customer class of the application model

```
public class Customer
{
        public int Id { get; set; }
        [Required]
        [StringLength(200)]
        public string Name { get; set; }
        [Required]
        [StringLength(16)]
        public string VAT { get; set; }
        public bool Enabled { get; set; }
}
```

The Entity Framework **DbContext**, in our case the **MyAppDbContext**, is very simple. We use the **OnModelCreating** to seed the database with some sample data.

Code Listing 37: The application DbContext

```
public class MyAppDbContext: DbContext
{
        public MyAppDbContext(DbContextOptions<MyAppDbContext> options)
            : base(options) { }

        public DbSet<Customer> Customers { get; set; }

        protected override void OnModelCreating(
```

```
            ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Customer>().HasData(
                new Customer()
                {
                    Id = 1,
                    Name = "Microsoft",
                    VAT = "IE8256796U",
                    Enabled = true
                },
                new Customer()
                {
                    Id = 2,
                    Name = "Google",
                    VAT = "IE6388047V",
                    Enabled = false
                });
        }
}
```

Yes, it sounds good to have these customers. As you probably already know, the initialization of the database with Entity Framework requires the generation of a migration model, which you can scaffold from the code executed with the following command from the terminal, in the project root.

Code Listing 38: Command to generate Entity Framework initial migration

```
dotnet ef migrations add InitialCreate
```

This command creates a **`Migrations`** folder that you have to move into the **`src`** folder: in this way, it will be copied in the container. Now you need to connect to the bash of the **`frontend`** container using the **`docker exec`** command.

Code Listing 39: Docker command to connect with the container bash shell

```
docker exec -it frontend bash
```

Now you are in the **`frontend`** container linked to the **`sqlserver`** container, so you can generate the database from the migration by using the following command from the **`app`** folder.

Code Listing 40: Entity Framework command to update the target database

```
dotnet ef database update
```

If everything works fine, you'll see the following result.

```
Micheles-MacBook-Pro:frontend micheleaponte$ docker exec -it frontend bash
root@2d5427ec19d4:/app#  dotnet ef database update
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core 2.2.0-rtm-35687 initialized 'MyAppDbContext' using provider 'Micr
h options: None
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (549ms) [Parameters=[], CommandType='Text', CommandTimeout='60']
      CREATE DATABASE [myapp];
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (181ms) [Parameters=[], CommandType='Text', CommandTimeout='60']
      IF SERVERPROPERTY('EngineEdition') <> 5
      BEGIN
          ALTER DATABASE [myapp] SET READ_COMMITTED_SNAPSHOT ON;
      END;
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (8ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      CREATE TABLE [__EFMigrationsHistory] (
          [MigrationId] nvarchar(150) NOT NULL,
          [ProductVersion] nvarchar(32) NOT NULL,
          CONSTRAINT [PK___EFMigrationsHistory] PRIMARY KEY ([MigrationId])
      );
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (3ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT OBJECT_ID(N'[__EFMigrationsHistory]');
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (1ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT [MigrationId], [ProductVersion]
      FROM [__EFMigrationsHistory]
      ORDER BY [MigrationId];
info: Microsoft.EntityFrameworkCore.Migrations[20402]
      Applying migration '20190125085815_InitialCreate'.
Applying migration '20190125085815_InitialCreate'.
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (2ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      CREATE TABLE [Customers] (
          [Id] int NOT NULL IDENTITY,
          [Name] nvarchar(200) NOT NULL,
          [VAT] nvarchar(16) NOT NULL,
          [Enabled] bit NOT NULL,
          CONSTRAINT [PK_Customers] PRIMARY KEY ([Id])
      );
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (27ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      IF EXISTS (SELECT * FROM [sys].[identity_columns] WHERE [name] IN (N'Id', N'Enabled', N
_ID(N'[Customers]'))
          SET IDENTITY_INSERT [Customers] ON;
      INSERT INTO [Customers] ([Id], [Enabled], [Name], [VAT])
      VALUES (1, 1, N'Microsoft', N'IE8256796U');
      IF EXISTS (SELECT * FROM [sys].[identity_columns] WHERE [name] IN (N'Id', N'Enabled', N
_ID(N'[Customers]'))
          SET IDENTITY_INSERT [Customers] OFF;
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (10ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      IF EXISTS (SELECT * FROM [sys].[identity_columns] WHERE [name] IN (N'Id', N'Enabled', N
_ID(N'[Customers]'))
          SET IDENTITY_INSERT [Customers] ON;
      INSERT INTO [Customers] ([Id], [Enabled], [Name], [VAT])
      VALUES (2, 0, N'Google', N'IE6388047V');
      IF EXISTS (SELECT * FROM [sys].[identity_columns] WHERE [name] IN (N'Id', N'Enabled', N
_ID(N'[Customers]'))
          SET IDENTITY_INSERT [Customers] OFF;
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (1ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      INSERT INTO [__EFMigrationsHistory] ([MigrationId], [ProductVersion])
      VALUES (N'20190125085815_InitialCreate', N'2.2.0-rtm-35687');
Done.
root@2d5427ec19d4:/app#
```

Figure 13: Database update command output

Now you only need to add a **`CustomersController`** to our project to test the communication with the database.

Code Listing 41: Application MVC customer's controller

```
public class CustomersController : Controller
{
        private readonly MyAppDbContext context = null;

        public CustomersController(MyAppDbContext context)
        {
            this.context = context;
        }

        public IActionResult Index()
        {
            var customers = this.context.Customers.ToList();
            return View(customers);
        }
}
```

Add an **`Index.cshtml`** to the **`Views/Customers`** folder to show the customer list.

Code Listing 42: Application customer's list view

```
@model IEnumerable<frontend.src.Models.Data.Customer>;
<table class="table">
    <tr>
        <th>Id</th>
        <th>Name</th>
        <th>VAT</th>
        <th>Enabled</th>
    </tr>
    @foreach (var item in Model)
    {
        <tr>
            <td>@item.Id</td>
            <td>@item.Name</td>
            <td>@item.VAT</td>
            <td>@item.Enabled</td>
        </tr>
    }
</table>
```

Finally, this is the result of all our efforts.

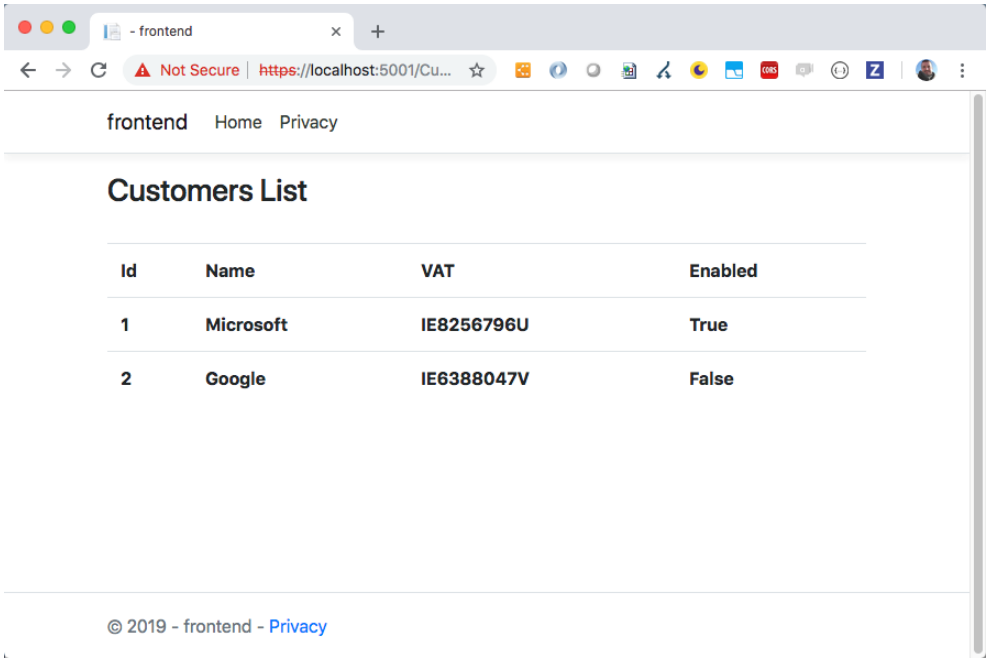

Figure 14: Application customer's list output

It's time to add some optimizations and some considerations to this configuration. The first observation is about the option **`--link`**. It is straightforward and works fine in our scenario, but its use is deprecated because of its limitations. With the **`--link`** option, Docker set up environment variables in the target container to correctly set the IP address and port numbers, so if I need to change something in the linked container (for example, to make an update needed to have a new image version), all my links break. Another big problem with the linking is that links are deleted when you delete the container.

The right solution to connect containers is the creation of a network with the command **`docker network create`**. In our case, we can execute the following command.

Code Listing 43: Docker command to create a network

```
docker network create myapp-network
```

Now we can add all the containers we want to this network simply using the option **`--net`**. In our case, we can run the containers as follows.

Code Listing 44: Docker commands to run containers with an existing network

```
- docker run -e 'ACCEPT_EULA=Y' -e 'SA_PASSWORD=Password_123' -e
'MSSQL_PID=Express' -p 1433:1433 --name sqlserver --net myapp-network
mcr.microsoft.com/mssql/server
- docker run -p:5000:5000 -p:5001:5001 -v $(pwd)/src:/app/src -t --link
sqlserver --name frontend --net myapp-network frontend
```

If you are lazy like me, you probably won't execute the database update command from the container shell every time you run the **frontend** container. The best way to automate this task is to create a startup script and execute it when the container starts, instead of the used CMD.

First, create a file in the **frontend** folder named **startup.sh** and insert in it the two commands to execute as the container starts.

Code Listing 45: Startup script for the front-end entry point

```
dotnet ef database update
dotnet watch run --urls "http://0.0.0.0:5000;https://0.0.0.0:5001"
```

We need to add the executing permission to the script; otherwise, we will receive a permission-denied error.

Code Listing 46: Command to add execution permission to the startup script

```
chmod +x startup.sh
```

You can also add this command in the **Dockerfile** with a **RUN** command if you want, but it is not necessary. Now we can change the **Dockerfile** as follows.

Code Listing 47: Dockerfile with ENTRYPOINT instruction

```
FROM microsoft/dotnet:sdk
RUN mkdir app
WORKDIR /app
COPY . .
ENTRYPOINT ./startup.sh
```

Note that we use the instruction **ENTRYPOINT** instead of **CMD**. You can see the last **CMD** for comparison; the symbol **#** indicates a comment, so the **CMD** row will not be executed. This instruction is the best practice when you want to start a container with a script, and when you want to use a container as an executable. If you rebuild the image (**docker build -t frontend .**) and recreate the container, you can watch the result in your terminal.

There is another important consideration for the SQL Server container. If you connect with the shell (**docker exec -it sqlserver bash**) and navigate to the **var/opt/mssql/data** folder, you can see the databases connected with the SQL Server instance.

```
Micheles-MacBook-Pro:frontend micheleaponte$ docker exec -it sqlserver bash
root@1c55de051bee:/# cd var/opt/mssql/data
root@1c55de051bee:/var/opt/mssql/data# ls
master.mdf  mastlog.ldf  model.mdf  modellog.ldf  msdbdata.mdf  msdblog.ldf  myapp.mdf  myapp_log.ldf
tempdb.mdf  templog.ldf
root@1c55de051bee:/var/opt/mssql/data#
```

Figure 15: SQL Server database files

Obviously, if you delete the container, your data will be lost. For development purposes, this isn't a problem, but you can solve this problem by creating a volume of this folder or attaching external files to the instance. Instead, if you are in production, you must save these files in a separate volume, but we will address this topic when we talk about Kubernetes.

Try to imagine a scenario with more than two containers: executing the commands to run all containers in the right order, with all the necessary parameters, can be very tedious. You also need to remember all this information or read it in the project documentation. The good news is that Docker provides a fantastic tool for running our container ecosystem. The tool is Docker Compose—let's look at how to use it to improve our daily work.

Run your container with Docker Compose

Docker Compose is a tool that can execute containers from a YAML configuration file. YAML is a format designed to be simple to understand by humans, like XML, but with minimum effort on learning syntax.

To understand the simplicity of the YAML format, let's create a file named **`docker-compose.yaml`** in the **`myapp`** folder. The first row of the script declares the version of the Compose configuration file that we want to use; version 3 is the most recent.

Code Listing 48: Docker Compose version

```
version: '3'
```

Now we need to declare the services that compose the ecosystem, in our case only two: the front end and the database. In YAML, each configuration block is delimited through indentation; to define two services, **`frontend`** and **`sqlserver`**, you need to use the following block.

Code Listing 49: Docker Compose services

```
services:
      frontend:
      sqlserver:
```

For each service, we can specify the image or the **`Dockerfile`** path for the creation of the image, the ports to map to the container, and eventually, the **`volumes`** and **`environments`** variables. You can also specify that a service depends on another service, so that Docker Compose can run the containers in the right order. For example, the configuration for our **`frontend`** is the following.

Code Listing 50: Docker Compose front end configuration

```
frontend:
    build: ./frontend
    ports:
      - 5000:5000
```

```
      - 5001:5001
    volumes:
      - ./frontend/src:/app/src
    depends_on:
      - sqlserver
```

We are describing to the composer how to create a service from an image that it has to build from the **Dockerfile** (omitted because it is the standard name) in the folder **./frontend**. When the image is built, it can run the container exposing the ports 5000 and 5001, and create a volume between the **./frontend/src** host folder and the **/app/src** container folder. We specify that this container depends on the service **sqlserver**, so the system has to start the **sqlserver** service first, and the **frontend** afterward.

The **sqlserver** service uses the image **mcr.microsoft.com/mssql/server**, exposes the port 1433, and sets the **environment** variables to configure SQL Server.

Code Listing 51: Docker Compose database configuration

```
sqlserver:
    image: mcr.microsoft.com/mssql/server
    ports:
      - 1433:1433
    environment:
      ACCEPT_EULA: "Y"
      SA_PASSWORD: "Password_123"
      MSSQL_PID: "Express"
```

Now we are ready to compose our application using Docker Compose. From the terminal, in the **myapp** folder, execute the following command.

Code Listing 52: Docker Compose execution command

```
docker-compose up
```

The first time you execute this command, you can observe some preliminary operations that will be cached to improve the performance of the future command execution. The first one is the front-end image creation.

```
Micheles-MacBook-Pro:myapp micheleaponte$ docker-compose up
Creating network "myapp_default" with the default driver
Building frontend
Step 1/5 : FROM microsoft/dotnet:sdk
 ---> 389d91a8617b
Step 2/5 : RUN mkdir app
 ---> Using cache
 ---> 35ea26e26cf3
Step 3/5 : WORKDIR /app
 ---> Using cache
 ---> 381eab21bcd0
Step 4/5 : COPY . .
 ---> d3c32c125399
Step 5/5 : ENTRYPOINT ./startup.sh
 ---> Running in 665a0fbd07e9
Removing intermediate container 665a0fbd07e9
 ---> 528b34acba13
Successfully built 528b34acba13
Successfully tagged myapp_frontend:latest
WARNING: Image for service frontend was built because it did not already exist. To rebuild this image you mus
t use `docker-compose build` or `docker-compose up --build`.
```

Figure 16: Docker Compose execution output

In our case, the image will be tagged as **`myapp_frontend:latest`**. Just after this operation, the composer creates the container **`myapp_sqlserver_1`** and **`myapp_frontend_1`** and attaches the database to the **`frontend`** through the creation of a default network named **`myapp_default`**.

```
Creating myapp_sqlserver_1 ... done
Creating myapp_frontend_1  ... done
Attaching to myapp_sqlserver_1, myapp_frontend_1
```

Figure 17: Docker Compose containers creation

When the **`myapp_sqlserver_1`** container starts up, the entry point of the **`frontend`** is executed, so Entity Framework creates the database and fills the sample data in it. Once this task is complete, the **`frontend`** starts.

```
frontend_1   | Hosting environment: Development
frontend_1   | Content root path: /app/src
frontend_1   | Now listening on: http://0.0.0.0:5000
frontend_1   | Now listening on: https://0.0.0.0:5001
frontend_1   | Application started. Press Ctrl+C to shut down.
```

Figure 18: Docker Compose front-end container execution output

If you navigate to the address https://localhost:5001/Customers, you can see the same result of the manual execution of our application. Very convenient, don't you think? Now, if you want to interrupt the application execution, you have only to press Ctrl+C. The execution interruption doesn't delete the containers, so if you want to restart the application, you can re-execute the **`docker-compose up`** command.

If you are not interested in the interactive mode, you can also execute the command **`docker-compose start`**.

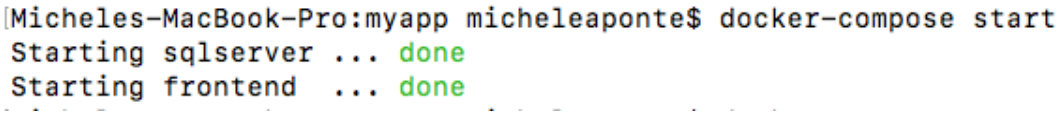

Figure 19: docker-compose start command output

In this case, you can stop the execution with the command **`docker-compose stop`**.

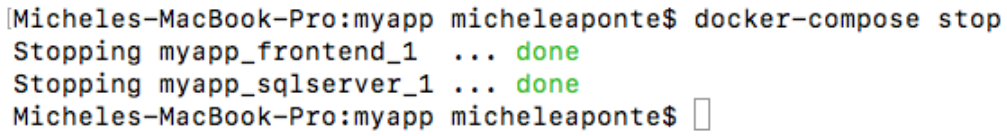

Figure 20: docker-compose stop command output

If you want to delete the created containers (not the images), you can use the command **`docker-compose down`**. In this case, the default network created for the communication between the services will also be deleted.

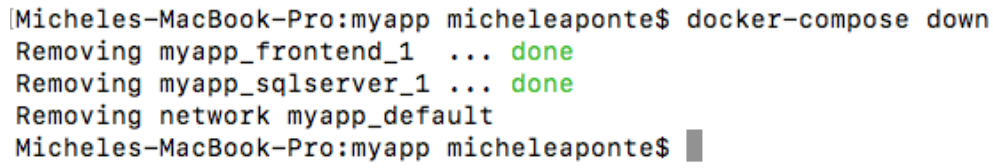

Figure 21: docker-compose down command output

Now that our development environment is configured and convenient to use, let's get ready for the next stage: preparing the images for deployment.

Create the final image for publication

When your code is ready for deployment, you can prepare the image for publication on the target registry. Docker Hub is one example, but you can also create your own registry or use a private registry. Azure, for example, provides the Azure Container Registry service to publish your images.

The final image doesn't run the application as a development image (**`dotnet watch run`**), but we need to publish the application with the command **`dotnet publish`**, probably without debugging support files, and in a specific directory. In our case, the right command is the following.

Code Listing 53: Command to publish a .NET Core application

```
dotnet publish -c Release -o out
```

Therefore, we need to proceed in two steps: publish our application and then execute the publication result. The **`publish`** command creates a DLL library with the name of the project (**`frontend.dll`**), so the right command to execute the **`frontend`** is the following.

Code Listing 54: Command to execute a published .NET Core application

```
dotnet frontend.dll
```

As you might recall, to create the image for development, we started from the image **microsoft/dotnet:sdk**, which contains the whole .NET software development kit needed for the compilation. The execution of a published application doesn't require all the SDK; you only need the ASP.NET Core runtime, available with the image **microsoft/dotnet:aspnetcore-runtime**.

Remember, we need to write a specific **Dockerfile** for the final image. In the **frontend** folder, create a file named **Prod.Dockerfile** and add the following rows.

Code Listing 55: Final image Dockerfile build step

```
#step 1: build
FROM microsoft/dotnet:sdk AS build-stage
RUN mkdir app
WORKDIR /app
```

The instruction **FROM** is the same as the development image, but in this case, we also use the instruction **AS**. This permits us to name this first stage to remind us that this is the step where we build the application: **build-stage**. After the creation of the **app** folder and the set of the working directory, in the development **Dockerfile** we copied our application files on the image. Now we want to do something different.

Code Listing 56: Files copy and dependencies restore of the final image Dockerfile build step

```
COPY frontend.csproj .
RUN dotnet restore
COPY ./src .
```

First, we want to copy only the project file (**frontend.csproj**) that contains the project configuration needed to run the command **dotnet restore**, which downloads all the project dependencies from NuGet. After the restore, we copy the **./src** folder to the **app** folder. This change permits us to effortlessly resolve the problem of the **Views** and **wwwroot** default path of ASP.NET Core. These folders are in the root project folder by default, while we find them in the **src** folder. If you remember, we have solved this problem in the development image by changing the content root when we create the web host builder (Code Listing 57).

When we publish the application, the Views folder is encapsulated in a DLL that for us is named **frontend.Views.dll**. The **wwwroot** folder instead is copied without changes. At runtime, when we execute the compiled application, ASP.NET Core doesn't find our views if they are in a different folder. Moreover, the **wwwroot** folder will be not be copied for the same reason.

Copying the **src** folder in the destination **app** folder solves the problem, but our content root changed and doesn't work because the **src** folder doesn't exist. We can solve this problem with a simple **#if** directive.

Code Listing 58: Content root conditional configuration

```
WebHost.CreateDefaultBuilder(args)
      #if DEBUG
      .UseContentRoot(
           Path.Combine(Directory.GetCurrentDirectory(), "src"))
      #endif
      .UseStartup<Startup>();
```

Returning to our **Dockerfile**, we publish the application with the **Release** configuration in a specific folder named **out**.

Code Listing 59: .NET Core publish command in Dockerfile

```
RUN dotnet publish -c Release -o out
```

The **Dockerfile** syntax permits us to have more than one stage in a single file so that we can add the following statements to the **Prod.Dockerfile**.

Code Listing 59: Final image Dockerfile run step

```
#step 2: run
FROM microsoft/dotnet:aspnetcore-runtime
WORKDIR /app
COPY --from=build-stage /app/out .
ENTRYPOINT dotnet frontend.dll
```

Starting from the **aspnetcore-runtime** image, we set the existing folder **/app** as the working directory and copy from the previous stage the content of the **/app/out** (the publish artifacts). Now we are ready to set the entry point of the new image.

Perfect! We are now ready to create our final image. From the terminal window, go to the **frontend** folder and execute the following command.

Code Listing 60: Docker final image build command

```
docker build -t frontend:v1 -f Prod.Dockerfile .
```

We have tagged our image with the name **frontend:v1** to avoid applying the latest version. Moreover, our **Dockerfile** has a nonstandard name, so we need to specify the **-f** parameter with the correct **Dockerfile** name. The result is the following.

```
Micheles-MacBook-Pro:frontend micheleaponte$ docker build -t frontend:v1 -f Prod.Dockerfile .
Sending build context to Docker daemon  3.958MB
Step 1/11 : FROM microsoft/dotnet:sdk AS build-stage
 ---> 389d91a8617b
Step 2/11 : RUN mkdir app
 ---> Running in 1c0f242b67fa
Removing intermediate container 1c0f242b67fa
 ---> d17bdf220236
Step 3/11 : WORKDIR /app
 ---> Running in d22c066f6554
Removing intermediate container d22c066f6554
 ---> 6388a582a573
Step 4/11 : COPY frontend.csproj .
 ---> f1e20b13b929
Step 5/11 : RUN dotnet restore
 ---> Running in bf1d5a26c4c1
  Restoring packages for /app/frontend.csproj...
  Generating MSBuild file /app/obj/frontend.csproj.nuget.g.props.
  Generating MSBuild file /app/obj/frontend.csproj.nuget.g.targets.
  Restore completed in 582.79 ms for /app/frontend.csproj.
Removing intermediate container bf1d5a26c4c1
 ---> 8fe23f7fdd39
Step 6/11 : COPY ./src .
 ---> a62d7eba4db2
Step 7/11 : RUN dotnet publish -c Release -o out
 ---> Running in 4b16b6804189
Microsoft (R) Build Engine version 15.9.20+g88f5fadfbe for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

  Restore completed in 36.73 ms for /app/frontend.csproj.
  frontend -> /app/bin/Release/netcoreapp2.2/frontend.dll
  frontend -> /app/bin/Release/netcoreapp2.2/frontend.Views.dll
  frontend -> /app/out/
Removing intermediate container 4b16b6804189
 ---> 31fe1254dbc7
Step 8/11 : FROM microsoft/dotnet:aspnetcore-runtime
 ---> 47e847c04838
Step 9/11 : WORKDIR /app
 ---> Running in 075c047dbcbe
Removing intermediate container 075c047dbcbe
 ---> 8c982bb8c075
Step 10/11 : COPY --from=build-stage /app/out .
 ---> 573b5007ee53
Step 11/11 : ENTRYPOINT dotnet frontend.dll
 ---> Running in 36239bad3f8d
Removing intermediate container 36239bad3f8d
 ---> 743c35a2e0a7
Successfully built 743c35a2e0a7
Successfully tagged frontend:v1
Micheles-MacBook-Pro:frontend micheleaponte$
```

Figure 22: Docker final image build command output

You can run the image with the following command.

Code Listing 61: Command to run a container from the final image

```
docker run -it -p 80:80 --name frontend frontend:v1
```

However, the use of **`docker-compose`** is more convenient. So let's duplicate the **`docker-compose.yml`** previously created, rename it in **`docker-compose.prod.yml`**, and change the **`frontend`** service as follows:

Code Listing 62: Docker Compose frontend service for the final image

```
frontend:
  build:
    context: ./frontend
    dockerfile: Prod.Dockerfile
```

```
  ports:
    - 80:80
  depends_on:
    - sqlserver
```

This example shows us that the **`build`** setting is not the path of the Dockerfile, but the execution context of the build. So, if the Dockerfile name is not standard, you need to make the execution context (**`./frontend`**) and the Dockerfile name (**`Prod.Dockerfile`**) explicit. In this case, the port is 80, and we do not need a volume. We can start our container with the following command.

Code Listing 63: Production Docker Compose execution

```
docker-compose -f docker-compose.prod.yml up
```

Now, if you open the browser and navigate to the http://localhost address, you can see that everything works fine. But if you navigate to the customer's controller, you'll see the following error.

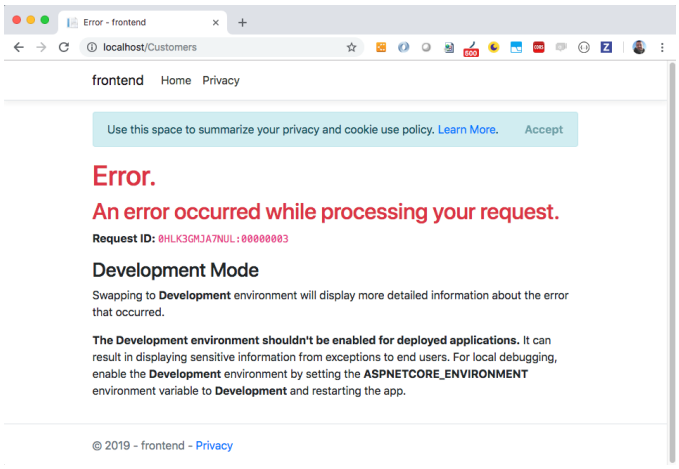

Figure 23: Database access error with the production configuration

This error occurs because we do not execute the database update from our migration scripts, but the question is: should we permit, in the production environment, the update of the database by a migration launched from the **`frontend`**? The answer depends on your update strategy. In a real deployment pipeline, manual or automatic, you probably have a backup/snapshot task to restore the last version in case of problems with the new release. The best way is to have a separate task of the deployment pipeline that launches a script, manually or automatically (it also depends on the target stage of your deployment), that updates the database to a new version.

There are also other problems when executing the database update from an application container. For example, if your **`frontend`** is deployed in an environment that can automatically scale depending on the traffic (the cloud), each instance created will try to update the database. The Microsoft official documentation reports as follows:

Database migration should be done as part of deployment, and in a controlled way. Production database migration approaches include:

- Using migrations to create SQL scripts and using the SQL scripts in deployment.
- Running the **`dotnet ef`** database update from a controlled environment.

EF Core uses the **`_MigrationsHistory`** table to see if any migrations need to run. If the DB is up to date, no migration is run.

You can find the documentation here.

If you want to create a SQL script starting from the migration, you can execute the following command from the terminal (in the **`frontend`** folder).

Code Listing 64: SQL script generation from the Entity Framework migration

```
dotnet ef migrations script --idempotent --output "script.sql"
```

The **`idempotent`** parameter generates a script that can be used on a database at any migration. The SQL Server image contains a tool named **`sqlcmd`** that allows us to execute the SQL command from the command line. If the database does not exist, you can create it with the following command

Code Listing 65: Database creation with the docker exec command

```
docker exec -it myapp_sqlserver_1 /opt/mssql-tools/bin/sqlcmd -S
localhost -U SA -P 'Password_123' -Q 'CREATE DATABASE [myapp]'
```

The parameter **`-S`** specifies the server, which in this case is **`localhost`** because we execute the command from the SQL Server container. The parameters **`-U`** and **`-P`** set the username and password to connect with the database. The parameter **`-Q`** sets the query that you want to execute.

To run the created script, we need to copy it in the container with the **`docker cp`** command.

Code Listing 66: SQL script copy to the container with the docker cp command

```
docker cp script.sql myapp_sqlserver_1:/script.sql
```

To execute the script, you can use the **`sqlcmd`** tool with the option **`-d`** to indicate the database (**`myapp`** in our case), and the script name with the parameter **`-i`**, as follows.

Code Listing 67: SQL script execution in the container with the docker exec command

```
docker exec -it myapp_sqlserver_1 /opt/mssql-tools/bin/sqlcmd -S
localhost -U SA -P 'Password_123' -d myapp -i script.sql
```

Another way to do the same thing is to create a custom image from the SQL Server image with the script and the commands to execute it, but again, this is not the best practice for the production environment.

Speaking of the production environment, you need to make the main settings of your container configurable, like the database connection string or the execution environment.

The base template of ASP.NET Core configures only the development, staging, or production environments, but you can add all the stages you want. This value is stored in the environment variable named **`ASPNETCORE_ENVIRONMENT`**, so you can set any value you want. The good news is that, when publishing an ASP.NET Core application with the Release configuration, the **`ASPNETCORE_ENVIRONMENT`** is set to **`Production`**. With Docker, you can change this value by using the **`-e`** parameter with **`docker run`**, or with the **`block`** environment of the **`docker-compose`** syntax. In the end, this configuration doesn't impact the final image build.

The database connection string is particularly interesting because it contains information that you probably would like to be able to change. You can make it configurable with the creation of an environment variable that contains the connection string value. We need to add the following row in **`Dockerfile`** and **`Prod.Dockerfile`**.

Code Listing 68: Docker environment variable creation for the SQL Server connection string

```
ENV SQLSERVER_CONNECTIONSTRING = 'Server=sqlserver; Database=myapp;
User=sa; Password=Password_123; MultipleActiveResultSets=true;'
```

The instruction **`ENV`** declares an environment variable (in our case, the **`SQLSERVER_CONNECTIONSTRING`** variable) and its default value. To use it, we only need to read the connection string from the environment variables in the **`Configure`** method of the **`Startup.cs`** file, and set it to the **`MyAppDbContext`** configuration.

Code Listing 69: Retrieve connection string from the environment variable

```
var connectionString = Environment
      .GetEnvironmentVariable("SQLSERVER_CONNECTIONSTRING");
services.AddDbContext<MyAppDbContext>(
      o => o.UseSqlServer(connectionString));
```

From this moment on, we are able to change the connection string when executing a container from the **`frontend`** image.

Ok, it's time to publish our image. First of all, we have to decide where to publish the container. For our example, we can use Docker Hub, which requires an account (it's free); if you have downloaded the Docker Tools from the official site, you already have one.

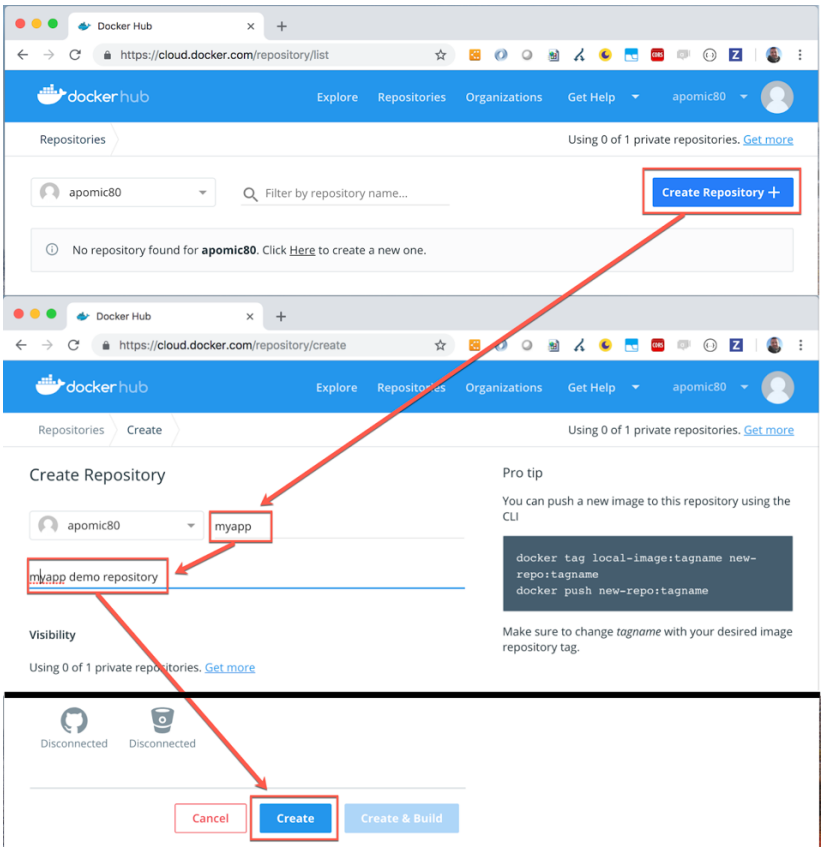

Figure 24: Docker Hub Repository creation

So, go to the Docker Hub and log in with your credentials. On the **Repositories** page, click **Create a Repository**, choose a name for the repository (**`myapp`**, in our case), and optionally add a description. Click **Save** to continue (Figure 24).

Your repository is now ready. From the terminal, log in to the Docker Hub with your credentials, using the **`docker login`** command.

```
Micheles-MBP:frontend micheleaponte$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't
have a Docker ID, head over to https://hub.docker.com to create one.
Username: apomic80
Password:
Login Succeeded
Micheles-MBP:frontend micheleaponte$
```

Figure 25: Docker Hub login command

Before publishing, we need to rename our image to meet the registry requirements. Execute the following command from the terminal.

Code Listing 69: Docker command to change image tag

```
docker tag frontend:v1 apomic80/myapp:frontend-v1
```

The **`docker tag`** permits us to rename the image tag from **`frontend:v1`**, created in the last build, to **`<account_username>/<repository_name>:<tag>`** (in our case, **`apomic80/myapp:frontend-v1`**). Now we can publish the image with the **`docker push`** command.

```
Micheles-MBP:frontend micheleaponte$ docker push apomic80/myapp:frontend-v1
The push refers to repository [docker.io/apomic80/myapp]
4314ec90e8bf: Pushed
f760a30dab91: Pushed
8a8de9be5a63: Mounted from microsoft/dotnet
f876f4ce37fd: Mounted from microsoft/dotnet
f0296b566559: Mounted from microsoft/dotnet
3c816b4ead84: Mounted from microsoft/dotnet
frontend-v1: digest: sha256:f851ba0d3c64190aaf837dd526f3cfe49f5c7dd44021deb837ae
0b3da9af5bf1 size: 1580
Micheles-MBP:frontend micheleaponte$
```

Figure 26: Docker push image execution output

From now on, we can deploy a **`frontend`** container everywhere!

Chapter 3 Deploy Your Application on Kubernetes

The journey to Kubernetes begins

There are several reasons to choose Kubernetes as a container orchestrator, but you can appreciate its power and simplicity only when you start to work with it. Naturally, you need to understand its basic components and how they work together.

Kubernetes, often shorted to k8s (for the eight characters between the k and the s in the alphabet), was created by Google in a private project named Borg and donated to the open-source community in 2014. It is written in Go/Golang, and its name is a Greek word that means helmsman, the person who steers a ship.

Kubernetes is designed to manage and scale your containers based on your configuration. To do this, it offers a cluster architecture to simplify the communication among available resources. Today, it's the de facto standard, adopted and supported by the major cloud providers because it is platform-agnostic and well-documented.

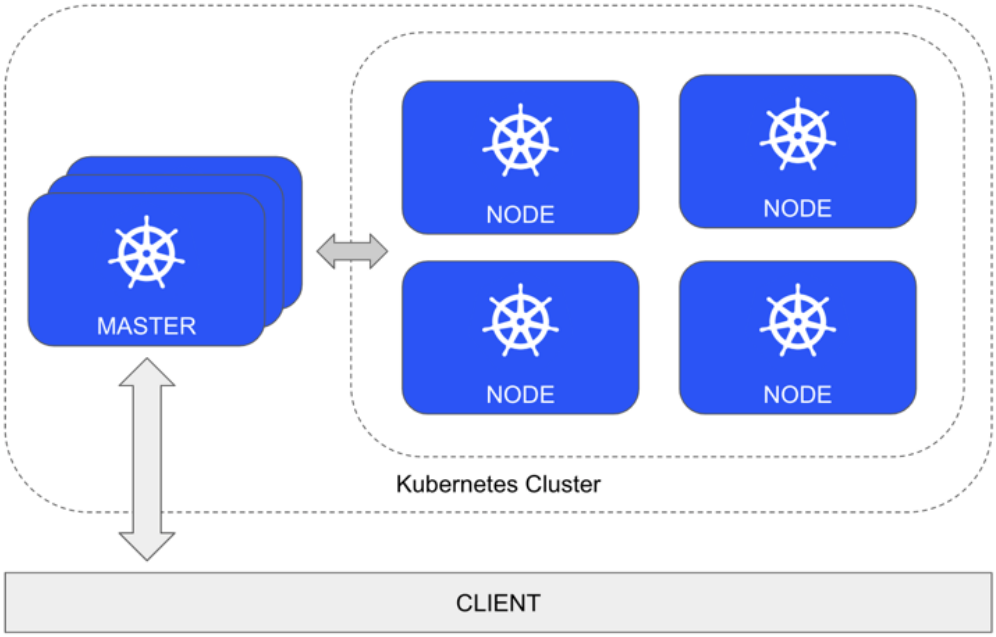

Figure 27: Kubernetes cluster components

As with any cluster, Kubernetes is composed of one or more master nodes, called masters, and one or more slave nodes, called nodes. You can have more masters for high-availability requirements. So if one master dies or is unavailable, the other master can guarantee that the system continues to work.

In the master, no user containers can be executed—only the services needed to control the nodes can be executed, ensuring that the system as a whole respects the configuration.

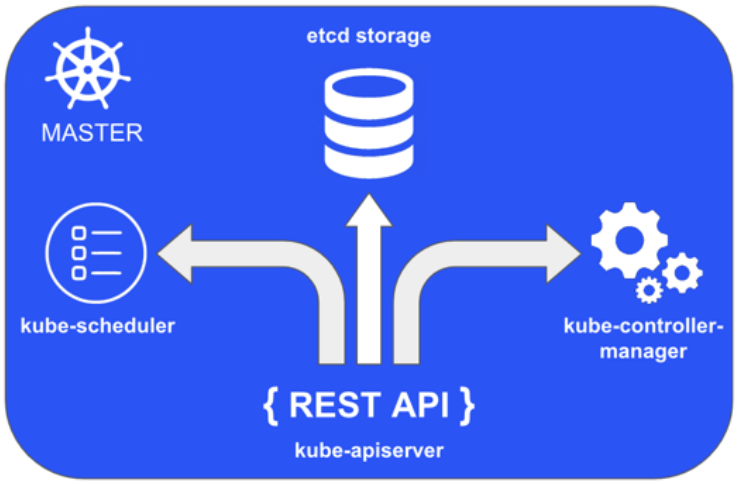

Figure 28: Kubernetes master

The master exposes a REST API server that consumes JSON data via manifest files to allow the Kubernetes configuration. This component is named **`kube-apiserver`**. The cluster configuration and state are saved on a distributed, consistent, and watchable persistent storage, accessible only through the REST API. The storage uses **`etcd`**, an open-source key/value store designed for small amounts of data that can be kept in memory. This store is the heart of Kubernetes, so you should back it up.

The master node also contains some controller services to control specific items like the nodes, namespaces, and endpoints. This component is named **`kube-controller-manager`**, and its primary goal is to watch for changes in the system and react to maintain the configured state. The last master component is named **`kube-scheduler`**, and it has the task of assigning work to the nodes and watching the API server for the creation of new pods. A pod is the minimum working unit in Kubernetes and contains one or more containers. We will examine them in depth shortly.

Figure 29: Kubernetes node

A Kubernetes node, previously called a “minion,” is the worker machine where the containers are executed. A node contains an agent named **`kubelet`** that communicates with the master to run operations on the node and reports back the node status. This agent registers the machine as a Kubernetes node, and after the registration, the node becomes a resource of the cluster. So, the node does not decide which activities should be done, because only the master knows the state of the whole system and can make that decision. The agent exposes an endpoint on the 10255 port, where it is possible to query the node for the configuration, status, and pods.

A node also contains the container engine to perform operations like pull images and start and stop the containers. The container engine is pluggable, so you can use any container engine you want. Usually, the container engine is Docker, but you can change it for any container specification implementation. For example, you can use **`rkt`**, the Kubernetes pod-native container runtime.

The Kubernetes node networking is managed from **`kube-proxy`**, the node component that allows IP addressed to the pod and balances the traffic across all pods in a service.

Before going deep into how Kubernetes works, let's create an environment in which we can experiment with its components. We have some choices, depending on which operating system you use, how much memory and CPU you have, and how lazy you are.

We can distinguish two main possibilities: the creation of a local environment or the configuration of a Kubernetes cluster in one of the available cloud providers.

Microsoft with Azure, Google with Google Cloud Platform, and Amazon with AWS provide services to create a Kubernetes cluster as a managed service, so you need only to configure your needs and pay for it. A production environment is the best choice, but first, you could explore the Kubernetes features in a local environment.

For the creation of a local environment, you have three options:

- Install a Kubernetes cluster locally using virtual machines.
- Install Minikube.
- Enable Kubernetes in Docker Desktop.

The first option requires experience with Linux administration and much patience, because creating a Kubernetes cluster (with only one master and one node) can be a very frustrating experience.

Minikube was born to solve this kind of problem. It is a tool that installs and runs a single-node cluster inside a virtual machine, designed for development purposes. You can find the instructions for the installation on the official documentation.

As you probably predicted, we will opt for the third option, which is perfect if, like me, you are lazy and you already have the Docker tools installed on your machine. From the Docker tray icon, choose the **Settings** menu item, select the **Kubernetes** tab in the **Settings** dialog, select the **Enable Kubernetes** check box, and click **Apply**.

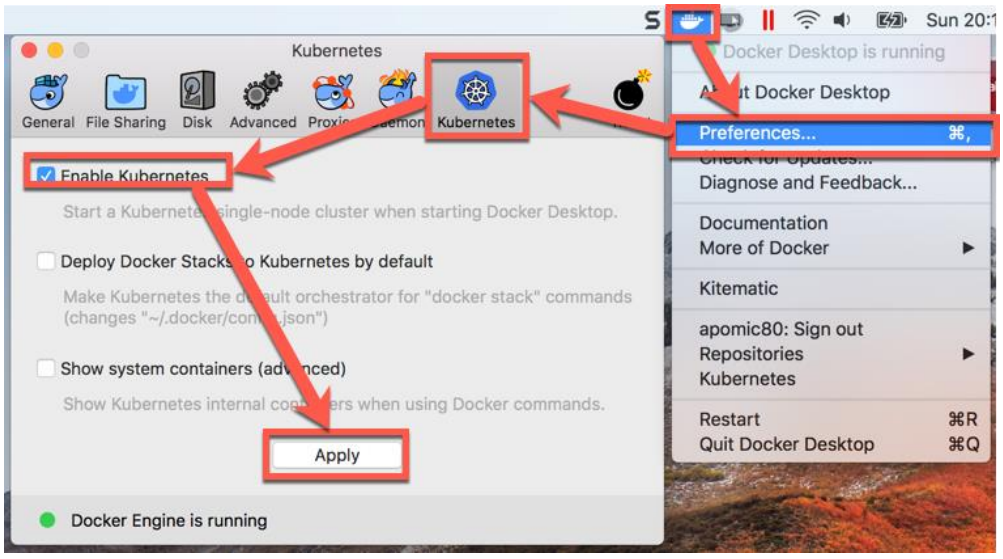

Figure 30: Enabling Kubernetes in Docker for Desktop

The installation requires several minutes, depending on your connection and the power of your machine. The good news is that all the operations are automatic, so you only need to wait for the success message.

When the installation ends, you can test that it works from the terminal by executing the following command.

Code Listing 70: Command to show information about the Kubernetes cluster

```
kubectl cluster-info
```

The **`kubectl`** is a command line tool installed with Kubernetes that allows us to interact with a cluster. If the command executes correctly, you can see the cluster information.

```
Micheles-MBP:~ micheleaponte$ kubectl cluster-info
Kubernetes master is running at https://localhost:6443
KubeDNS is running at https://localhost:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
Micheles-MBP:~ micheleaponte$
```

Figure 31: Kubernetes information command output

When you interact with Kubernetes, you can adopt an imperative or declarative approach. With the imperative approach, you impose a command on the Kubernetes cluster that will execute it and return a result. With the declarative approach, you can describe your desired state for the cluster and let Kubernetes do all the necessary work to achieve the status indicated.

To understand the difference between the two approaches and gain the necessary knowledge to publish our application in the local cluster, we need to introduce and analyze some Kubernetes concepts. Let the games begin!

Deploy your images in Kubernetes

Suppose we want to deploy our **frontend** in Kubernetes. To do this, we need to create a Kubernetes resource known as the *pod*. A pod is the minimum deployable unit in the cluster, so in Kubernetes, we talk about pods instead of containers.

A pod can contain one or more container—usually only one, but if you have more containers that are strongly dependent, you can deploy them together in the same pod. They will scale together and share the same IP address. You can create a pod from the command line using the **kubectl run** command, specifying a name for it, the image to use, and the port where the container responds.

Code Listing 71: Command to create a Kubernetes pod

```
kubectl run frontend --image=apomic80/myapp:frontend-v1 --port 80 --
restart=Never
```

The parameter **restart** indicates that if the pod fails or dies, it will never be recreated. If you run the command **kubectl get pods**, you can see the created pods and their status.

```
Micheles-MBP:~ micheleaponte$ kubectl get pods
NAME       READY     STATUS    RESTARTS   AGE
frontend   1/1       Running   0          1m
Micheles-MBP:~ micheleaponte$
```

Figure 32: Kubernetes pod creation output

To analyze the created pod in depth, you can use the command **kubectl describe pod frontend**, which shows you all the available information about the pod and the containers contained within it.

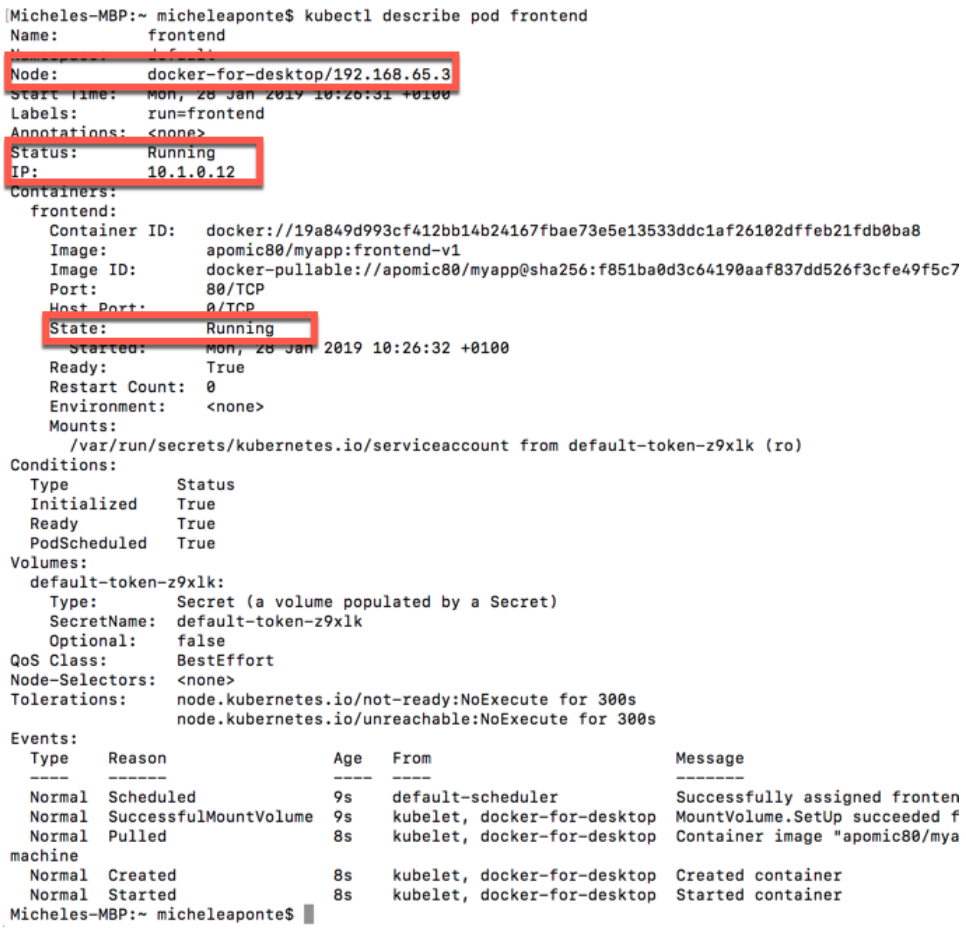

Figure 33: Kubernetes pod description output

In the image, you can see a status label for the pod and a status label for the container. You already know the possible container states, but the pod has different values. It follows a life cycle that starts with the creation command, which brings its status to the **`Pending`** value, during which the containers in it are loaded. If there are errors creating the containers, the pod will assume the **`Failed`** state; otherwise, it will assume the state **`Running`**. **`Running`** means that all containers are created and at least one is starting, restarting, or running. When all the containers in the pod are terminated successfully and will not be restarted, the pod will assume the **`Succeeded`** status.

To delete the pod, you can execute the following command.

Code Listing 72: Command to delete a Kubernetes pod

```
kubectl delete pod frontend
```

It is essential to understand that all the operations are not instantaneous—they are sent to the master, which validates them and sends the right requests to the nodes interested by the operation. So, if you execute the command **`kubectl get pods`**, just after the **`delete`** command, you will probably see the pod in the list in the **`Terminating`** status.

We have created a pod directly from the command line, but the best way to do so is to use a YAML configuration file that will be sent to the cluster. As for the **kubectl** tool, the YAML also requires a lot of practice to memorize the syntax. Let's create a **pod.yml** file with the following script to create the **frontend** pod.

Code Listing 73: YAML description to create a Kubernetes pod

```
apiVersion: v1
kind: pod
metadata:
  name: frontend-pod
  labels:
    app: myapp
    zone: prod
    version: v1
  spec:
    containers:
      - name: myapp-frontend
        image: apomic80/myapp:frontend-v1
        ports:
          - containerPort: 80
```

The first line indicates the version of the syntax used from the script, which is needed by the cluster to correctly understand the configuration. The second row defines the kind of item that we want to create, a **Pod**. After these initial rows, the script describes the item **metadata**. Here there are three blocks: **name**, which indicates the pod name; **labels**, which adds some custom labels to the pod in the **<key>:<value>** format; and **spec**, the item specification, which in the case of the pod is the definition of the containers in it. To create this pod, you can execute the following command.

Code Listing 74: Command to create a Kubernetes pod from a configuration file

```
kubectl create -f pod.yml
```

Instead of using **kubectl** tools to monitor the Kubernetes cluster, we can use a simple web user interface that graphically shows the cluster state. It is not enabled by default; it's basically a web portal deployed as a pod, so we have to create it from a YAML configuration file.

Code Listing 75: Command to create all the Kubernetes components to run the Dashboard

```
kubectl create -f
https://raw.githubusercontent.com/kubernetes/dashboard/master/aio/deploy/re
commended/kubernetes-dashboard.yaml
```

To access the application locally, we need to enable the Kubernetes proxy with the following command that starts a proxy server.

Code Listing 76: Command to start Kubernetes proxy server

```
kubectl proxy
```

Now you can open the Kubernetes Dashboard from here.

The first time you open the Dashboard, you need to authenticate with a token. Your token is kept in a special content, named **secret**, designed to be safe, so we need to locate it using the following command.

Code Listing 77: Command to show all Kubernetes secrets

```
kubectl -n kube-system get secret
```

This command shows all the secrets of the cluster. We need to retrieve the name of the secret that has a name starting with **deployment-controller-token**.

```
Micheles-MBP:kubernetes micheleaponte$ kubectl -n kube-system get secret
NAME                                     TYPE                                  DATA      AGE
attachdetach-controller-token-2fzlg      kubernetes.io/service-account-token   3         15h
bootstrap-signer-token-cdn4n             kubernetes.io/service-account-token   3         15h
bootstrap-token-i1rb5g                   bootstrap.kubernetes.io/token         7         15h
certificate-controller-token-bwwdr       kubernetes.io/service-account-token   3         15h
clusterrole-aggregation-controller-token-k987r   kubernetes.io/service-account-token   3         15h
cronjob-controller-token-8sdgr           kubernetes.io/service-account-token   3         15h
daemon-set-controller-token-r4njq        kubernetes.io/service-account-token   3         15h
default-token-w78xb                      kubernetes.io/service-account-token   3         15h
deployment-controller-token-rkdxq        kubernetes.io/service-account-token   3         15h
disruption-controller-token-dgbrn        kubernetes.io/service-account-token   3         15h
endpoint-controller-token-lc7qt          kubernetes.io/service-account-token   3         15h
generic-garbage-collector-token-dnqdz    kubernetes.io/service-account-token   3         15h
horizontal-pod-autoscaler-token-27cmc    kubernetes.io/service-account-token   3         15h
```

Figure 34: Kubernetes secrets list output

The name for ours is **deployment-controller-token-rkdxq**, so we can request the token using the following command.

Code Listing 79: Command to show the specified secret details

```
kubectl -n kube-system describe secret deployment-controller-token-rkdxq
```

```
Micheles-MBP:kubernetes micheleaponte$ kubectl -n kube-system describe secret deployment-controller-token-rkdxq
Name:         deployment-controller-token-rkdxq
Namespace:    kube-system
Labels:       <none>
Annotations:  kubernetes.io/service-account.name=deployment-controller
              kubernetes.io/service-account.uid=2fb638c0-2268-11e9-9883-025000000001

Type:  kubernetes.io/service-account-token

Data
====
ca.crt:     1025 bytes
namespace:  11 bytes
token:      eyJhbGciOiJSUzI1NiIsImtpZCI6Ii9.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Ii1wia3V1ZXJuZXRlcy5pby9zZXJ2aWNlYW
Njb3VudC9uYW1lc3BhY2UiOiJrdWJlLXN5c3RlbSIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY291bnQvc2VjcmV0Lm5hbWUiOiJkZXBsb3ltZW50LWNvbnRyb
2xsZXItdG9rZW4tcmtkeHEiLCJrdWJlcm5ldGVzLmlvL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UtYWNjb3VudC5uYW1lIjoiZGVwbG95bWVudC1jb250cm9sbGVy
Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZXJ2aWNlLWFjY291bnQudWlkIjoiMmZiNjM4YzAtMjI2OC0xMWU5LTk4ODMtMDI1MDAwMDAwMDAxIiw
ic3ViIjoic3lzdGVtOnNlcnZpY2VhY2NvdW50Omt1YmUtc3lzdGVtOmRlcGxveW1lbnQtY29udHJvbGxlciJ9.DTYiBMYshz9n7FgcefrmNNrVTGn_8jOP6vcdx
ZjzS9riKAr-DAOCwzp2Zpu6nQBOa2JtgcKkbPA5x3V0lDygAq9oshTzJHlHkhJdoen9QBWhuFnEpoLgP1x7VtMM-KC57LDV3Xd4jMIiKajiL7z3pw46A5wWob3r
EcUBPne6wxPnk1zL1PLbO4rPFlGhOepogcR4usNQ8975VdX3uYjHiHqtFu_odHiYM-wfSPtzUrK39Li1sPtp_-U6H7u6QcbU-HMdGqB5dk_OkIq-e5jkij4zqm3
DUbFVUMTgjJlpjd65pcGDlsiVVquKLoBJ8Zx3N9PBTfAIojLFXmkkvF50qQ
Micheles-MBP:kubernetes micheleaponte$
```

Figure 35: Kubernetes secrets detail output

Copy the token and paste it in the login window. Finally, we can see the Kubernetes Dashboard.

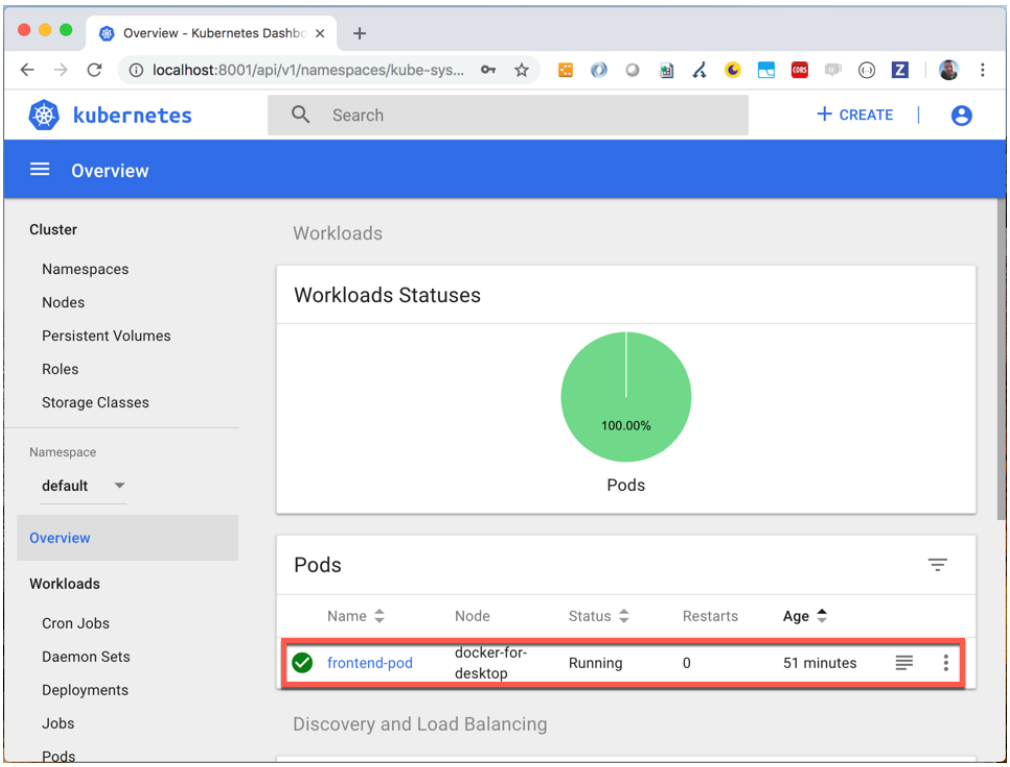

Figure 36: Kubernetes Dashboard

As you can see, there is only one instance of the pod, but Kubernetes is designed to scale, so we can use another item kind to declare that we want, for example, always five instances of the pod. To achieve this behavior, we need to use a **`ReplicationController`**, which is a master controller that replicates the specified pods, balances them on all the nodes, and reacts if some pod dies to restore the desired state.

Let's create a file named **`rc.yml`** and write the following script.

Code Listing 78: Kubernetes ReplicationController configuration script

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: myapp-rc
  spec:
      replicas: 5
      selector:
        app: myapp
      template:
        metadata:
        labels:
          app: myapp
          zone: prod
          version: v1
        spec:
          containers:
            - name: myapp-frontend
              image: apomic80/myapp:frontend-v1
              ports:
              - containerPort: 80
```

As you can see, the script version is the same as the previous one, but the **kind** value is now **ReplicationController**. In the **metadata** block, we specify the name of the replication controllers that will be created, and in the **specification** block, we indicate the number of replicas for the pod. We also need to specify a selector label to indicate which pod will be replicated, and a template with the information about the pod. The pod definition is the same as the previous script, so if you don't remove the previous pod, the replication controller will create only four pods.

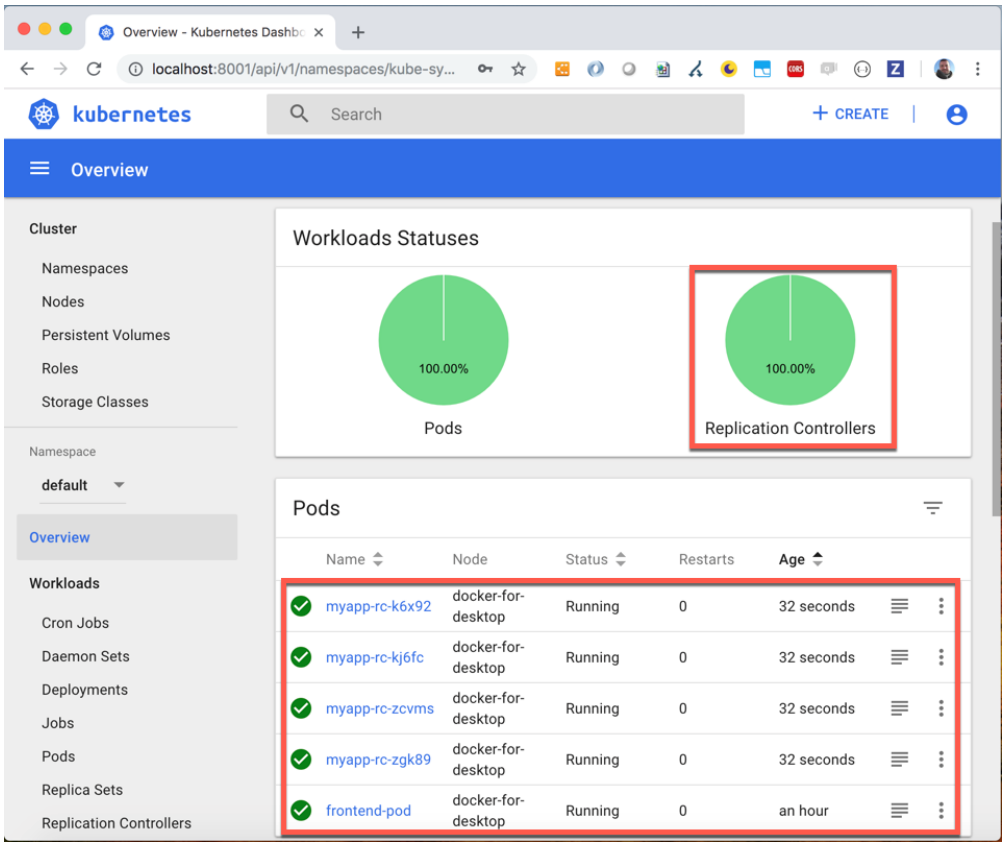

Figure 37: Kubernetes replication controller in Dashboard

Try to delete the **`frontend`** pod (**`kubectl delete pod frontend-pod`**) and observe how the replication controller will add a new pod to maintain the desired state of five replicas.

Expose your application with Kubernetes

Now that we've deployed our application, we want to access it. If you execute the **`kubectl describe`** command for the created pods (**`kubectl describe <pod-name>`**, see Figure 33), you can see that each of them is running on a specific IP, internal to the node. You can also see the Node IP (**`192.168.65.3`**), but you can't use it directly because the forwarding rule to send the requests from the node IP to the pod IP is missing. We need to expose the pod, and Kubernetes provides the services to do that.

A service has the task of exposing pods to a client or other pods, with an IP (or a DNS) and a port. We can expose a specified pod or a group of pods (our replicas). If we can expose the pods that compose the created replication controller (**`myapp-rc`**) to the client, we can run the **`kubectl expose`** command from the terminal as follows.

Code Listing 79: Command to expose the replication controller

```
kubectl expose rc myapp-rc --name=myapp-rc-service --type=NodePort
```

The **--name** option sets a name for the service. The **--type** option sets the service type, and can be one of the following four values:

- **ClusterIP**: The default value that exposes the pods internal to the cluster.
- **NodePort**: Exposes the pods as a static port on each node that contains the pods.
- **LoadBalancer**: Exposes the pods externally using a load balancer of a cloud provider.
- **ExternalName**: Available from version 1.7 of **kube-dns** to expose the pods through the contents of the **externalName** field.

We used the **NodePort** type to make the service available on a port, which we can retrieve with the described command (**kubectl describe service myapp-rc-service**).

```
Micheles-MacBook-Pro:kubernetes micheleaponte$ kubectl describe service myapp-rc-service
Name:                     myapp-rc-service
Namespace:                default
Labels:                   app=myapp
                          version=v1
                          zone=prod
Annotations:              <none>
Selector:                 app=myapp
Type:                     NodePort
IP:                       10.111.118.173
LoadBalancer Ingress:     localhost
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
NodePort:                 <unset>  31521/TCP
Endpoints:                10.1.0.28:80,10.1.0.31:80,10.1.0.32:80 + 2 more...
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
Micheles-MacBook-Pro:kubernetes micheleaponte$
```

Figure 38: Output of the created service description

Now we can reach our application at the address http://localhost:31521 because we have a local cluster.

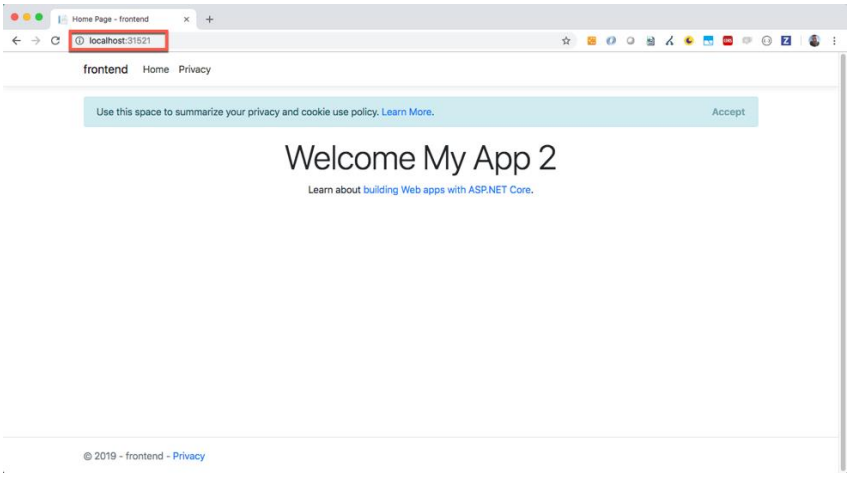

Figure 39: The application exposed from the Kubernetes cluster

The **NodePort** type automatically creates a **ClusterIP** service used by the **NodePort** service to route the requests to the pods.

We can create a service in a declarative way using a YAML file. We create a file named **service.yml** with the following script.

Code Listing 80: Service YAML description

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-svc
spec:
  selector:
    app: myapp
  type: NodePort
  ports:
  - port: 80
    nodePort: 30001
```

The property **kind** now is **Service**, and we set its name in the **metadata** block as usual. The **spec** block sets three things: the **selector**, to indicate which pods will be exposed; the service type, **NodePort** in this case; and the **ports** mapping, in this case, the HTTP port 80 and **NodePort 30001** of the cluster. Now you can delete the previous service (**kubectl delete service myapp-rc-service**) and create the service using the following command from the terminal.

Code Listing 81: Command to create service from a YAML script

```
kubectl create -f service.yml
```

Now the application is available at the address http://localhost:30001.

Let's focus for a moment on the selector. What is the **app: myapp** value? The answer is interesting: it is one of the labels of the pods. Labels in Kubernetes are not only a way to add some information to elements, but are also used to connect them. In this case, we are connecting the service with the pods, and we tell the system to select all pods that have the label **myapp** for the service. As you can see in Code Listing 81, the pods have three labels: **app**, with the value **myapp**; **zone**, with the value **prod**; and **version**, with the value **v1**. To select the pods, you can also use all the labels in the selector, opening a cool scenario.

Imagine deploying others pods, individually or with a replication controller, using the same labels, except for the version, for which you use the value **v2**. This is a typical scenario of new version deployment. If we have a new image with the version **v2** of our application, the replication controller will be as follows.

Code Listing 82: The v2 replication controller YAML script

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: myapp-rc2
spec:
  replicas: 5
  selector:
```

```
    app: myapp
    zone: prod
    version: v2
  template:
    metadata:
      labels:
        app: myapp
        zone: prod
        version: v2
    spec:
      containers:
        - name: myapp-frontend
          image: apomic80/myapp:frontend-v2
          ports:
            - containerPort: 80
```

In this case, you can change the value of the label version in the service to make the new pods available to the client with the second version of the application!

With the same mechanism, you can roll back to the **v1** version or use the **zone** label to pass from a test stage to the production stage. The useful aspect is that the labels are entirely customizable and you can create an unlimited number of them, so you can invent any mechanism you need.

Deploy your application with the deployments

In the real world, the deployment and maintenance of an application can be more complicated than the scenario we have seen with the service and the replication controllers. The labels are a powerful tool for managing Kubernetes elements, but we need a more powerful mechanism that permits us to manage updates and rollbacks in more detail.

For example, say we need to update our application with a rolling update strategy that releases new pods more gradually, one pod at a time. We need to roll back the application to a specific version of the deploy history. We can obtain these behaviors and others in Kubernetes using the deployment.

Do you remember the deployment of our first pod? You can see it in Code Listing 73. We added the **`--restart`** option with the **`Never`** value to create a simple pod, because without the **`--restart`** option, the system would have created a deployment. Try to execute the following command.

Code Listing 83: Command to create a simple deployment

```
kubectl run frontend --image=apomic80/myapp:frontend-v1 --port 80
```

In Figure 40 you can see the output of the command, which shows the creation of a deployment named **`frontend`** and the creation of a pod with a name assigned from the system.

```
Micheles-MacBook-Pro:kubernetes micheleaponte$ kubectl run frontend --image=apomic80/myapp:
frontend-v1 --port 80
deployment.apps "frontend" created
Micheles-MacBook-Pro:kubernetes micheleaponte$ kubectl get pods
NAME                        READY     STATUS    RESTARTS   AGE
frontend-6b74865b77-n9www   1/1       Running   0          8s
Micheles-MacBook-Pro:kubernetes micheleaponte$ kubectl get deployment
NAME       DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
frontend   1         1         1            1           13s
Micheles-MacBook-Pro:kubernetes micheleaponte$
```

Figure 40: Simple deployment creation output

You can see the deployment created with the **`kubectl get deployment frontend`** command. The output, again in Figure 40, shows the primary information about the deployment: the name, the desired replica of the pod, the current pods, the up-to-date pods, and the available pods.

These values suggest the power of a deployment: it manages the desired state of the application for us and reacts if something happens to alter this state.

A deployment has a large number of parameters, so the best way to create it is a YAML file with the description of our desired state. Let's create a file named **`deployment.yml`** with the following script:

Code Listing 84: Deployment YAML script

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
        zone: prod
        version: v1
    spec:
      containers:
        - name: myapp-frontend
          image:  apomic80/myapp:frontend-v1
          ports:
            - containerPort: 80
```

In this simple deployment, the structure is similar to the replication controller, but the selector is more explicit about the label mechanism to match the pods. After the deletion of the previous deploy with the **`kubectl delete deployment frontend`** command (just to prevent confusion), you can execute the script with the following command:

Code Listing 85: Command to create a deployment from the script

```
kubectl create -f deployment.yml
```

The system will create the deployment named **myapp-deployment**, which you can analyze using the **kubectl describe deployment myapp-deployment** command.

In Figure 41, you can see all the information about the deployment; the most interesting points are highlighted in red blocks. The first block shows the replicas and the update strategy, which by default uses the value **RollingUpdate**. You can also view the update strategy parameters that are specific for each strategy type. The second block introduces another Kubernetes element: **ReplicaSet**.

A ReplicaSet is a tool to manage the pod replicas and update strategies, and it is more flexible than the replication controller. It provides set-based labels selection support instead of the equality-based labels support of the **ReplicationController**. With equality-based labels support, you can match labels only with an "is equal" or "is not equal" assertion. With a set-based labels selection, you can express more powerful assertions, like **in**, **not in**, or **exists**.

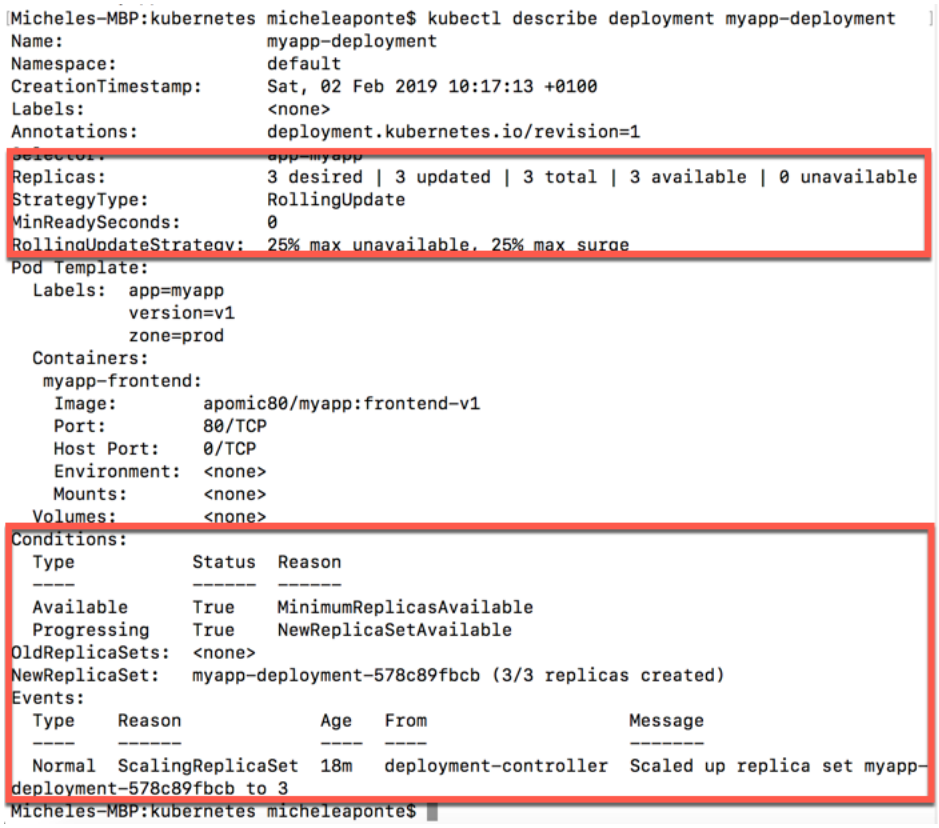

Figure 41: Output of the deployment describe command

For example, with a replication controller, you can match a group of pods that has a label **zone** with the value **prod** or **test**, but not both of them. With a ReplicaSet, you can match all the pods with the prod label in **prod** and **test**, using the **matchExpressions** property of the selector.

Code Listing 86: ReplicaSet selector with matchExpressions value

```
selector:
  matchLabels:
    app: myapp
  matchExpressions:
    - {key: zone, operator: In, values: [prod, test]}
```

A deployment automatically creates a ReplicaSet to manage the replicas and the update strategies configured.

Let's focus on the update strategies. Kubernetes offers two possible values that we can use to set the desired behavior to update our application:

- **`Recreate`**: Removes the previous version and loads the new.
- **`RollingUpdate`**: The default value; moves to a new version gradually based on configured parameters.

The value **`Recreate`** is useful in the development stage because it destroys the old version of the pods and creates the new one, interrupting the service. In a production scenario, interrupting the service is not acceptable, so the **`RollingUpdate`** value, with its parameters, can help us to update the application by degree.

With the **`RollingUpdate`** value, you can use the following parameters:

- **`maxUnavailable`**: The number of pods that can be unavailable during the update.
- **`maxSurge`**: The number of pods that can exceed the replicas requested.
- **`minReadySeconds`**: The number of seconds to wait before the next pod's creation.

The parameters **`maxSurge`** and **`maxUnavailable`** can be a number or a percentage; the **`minReadySecond`** can be only a number. When the update starts, Kubernetes creates a second **`ReplicaSet`**, where it deploys the new pods and removes the old one from the original **`ReplicaSet`**, following the parameter values. See, for example, the following script.

Code Listing 89: RollingUpdate parameters

```
spec:
  replicas: 10
  selector:
    matchLabels:
      app: myapp
    strategy:
      type: RollingUpdate
      minReadySeconds: 5
      rollingUpdate:
        maxSurge: 1
        maxUnavailable: 1
```

In this sample, Kubernetes will create a new **ReplicaSet** and deploy one pod at a time, waiting five seconds for each pod. During the update, only one pod can be unavailable, and the maximum number of pods is 11: ten for the replica value, and one for the **maxSurge** value. You can not specify the **minReadySecond** parameter, but if your container in the pod is not fast, you can make your application unavailable for a while.

If we modify Code Listing 87 (our deployment.yml file) with these values, we can apply the change using the following command.

Code Listing 87: Deployment apply command

```
kubectl apply -f deployment.yml --record
```

The **--record** parameter permits us to track all the changes to the deployment, which we will use to analyze the history and rollback to a previous version of the application. The operation requires some time to complete, but you can see the progress with the following command.

Code Listing 88: Deployment rollout status command

```
kubectl rollout status deployment myapp-deployment
```

You can see the result of this command in Figure 42.

```
Micheles-MacBook-Pro:kubernetes micheleaponte$ kubectl rollout status deployment myapp-d
eployment
Waiting for rollout to finish: 3 of 10 updated replicas are available...
Waiting for rollout to finish: 3 of 10 updated replicas are available...
Waiting for rollout to finish: 3 of 10 updated replicas are available...
Waiting for rollout to finish: 3 of 10 updated replicas are available...
Waiting for rollout to finish: 3 of 10 updated replicas are available...
Waiting for rollout to finish: 3 of 10 updated replicas are available...
Waiting for rollout to finish: 3 of 10 updated replicas are available...
Waiting for rollout to finish: 6 of 10 updated replicas are available...
Waiting for rollout to finish: 8 of 10 updated replicas are available...
deployment "myapp-deployment" successfully rolled out
```

Figure 42: Deployment rollout status output

If you used the **--record** parameter in the **apply** command, you can see the history of your deployment with the following command.

Code Listing 89: Deployment rollout history command

```
kubectl rollout history deployment myapp-deployment
```

Try to change one of the labels and re-execute the command in Code Listing 91; this way, we will have two rows in the history. The output of the **history** command shows the revisions and the cause of the change.

```
Micheles-MacBook-Pro:kubernetes micheleaponte$ kubectl rollout history deployment myapp-
deployment
deployments "myapp-deployment"
REVISION  CHANGE-CAUSE
1         kubectl apply --filename=deployment.yml --record=true
2         kubectl apply --filename=deployment.yml --record=true
```

Figure 43: Deployment history output

If we want to return to a specific revision, for example, revision 1, we can use the following command.

Code Listing 90: Deployment undo command

```
kubectl rollout undo deployment myapp-deployment --to-revision=1
```

Chapter 4 Deploy the Database

Create a deployment for SQL Server

Now that we have learned how to deploy a container in Kubernetes, we are ready to move the database used for our front end. To do this, you have to create a specific deployment script in a separate file or in the same file of the front-end deployment. You can choose the best solution for you based on your requirements. In the case of SQL Server, a specific script is probably the best way for most scenarios, because you deploy a standard image provided by Microsoft instead of your custom image.

We have to separate the deployment of our database in the development or testing stages and the deployment in the production stage because the update strategy could be different. For example, in the development stage, we use sample data that can be lost without problems. I can recreate both the pods and the data for each update. But in the production environment, I could recreate the pods, but I cannot lose data. We can use volumes to solve the same problems we deal with when using Docker and containers. In the next paragraph, we will see that Kubernetes provides a similar tool, but now we focus on a preliminary version of the script to deploy the database and see how we can enable connections from the front end using Kubernetes Services.

Let's create a **`db-deployment.yml`** file and insert the following script.

Code Listing 91: Database deployment script

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-db-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: myapp-db
  minReadySeconds: 5
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: myapp-db
    spec:
      containers:
        - name: myapp-database
          image: mcr.microsoft.com/mssql/server
          ports:
            - containerPort: 1433
```

```
          env:
            - name: "ACCEPT_EULA"
              value: "Y"
            - name: "SA_PASSWORD"
              value: "Password_123"
            - name: "MSSQL_PID"
              value: "Express"
```

It is very similar to the **frontend** script, but we use the **env** block to specify the environment variables that configure the SQL Server instance. This time the password is visible, but in the next paragraph, we will see a more secure mode of managing this value.

Now we need a service to expose the database, but for security reasons, we can expose it only inside the cluster. We don't use a **NodePort** Service, but a **ClusterIP** Service, and we set the correct labels to select the database pod.

Code Listing 92: Service script to expose the database

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-db-service
spec:
  selector:
    app: myapp-db
  ports:
    - protocol: TCP
      port: 1433
      targetPort: 1433
  type: ClusterIP
```

You can put the YAML script in the same file of the deployment because they are related, but you can also put it in another file if you want.

Now we are ready to create the new deployment using the **kubectl create** command.

Code Listing 93: Command to create the database deployment

```
kubectl create -f db-deployment.yml
```

We need to change the **frontend** deployment to configure the connection string in the environment variable created for the image.

Code Listing 94: Frontend deployment script update to add environment variables

```
    spec:
      containers:
        - name: myapp-frontend
```

```
          image: apomic80/myapp:frontend-v1
          ports:
            - containerPort: 80
          env:
            - name: ASPNETCORE_ENVIRONMENT
              value: DEVELOPMENT
            - name: SQLSERVER_CONNECTIONSTRING
              value:
"Server=localhost:30002;Database=myapp;User=sa;Password=Password_123;Multip
leActiveResultSets=true;"
```

We can now apply the changes using the **`kubectl apply`** command.

Code Listing 95: Frontend deployment configured with environment variables

```
kubectl apply -f deployment.yml
```

If we navigate to the Customers page of our front end, we can see the result shown in Figure 44.

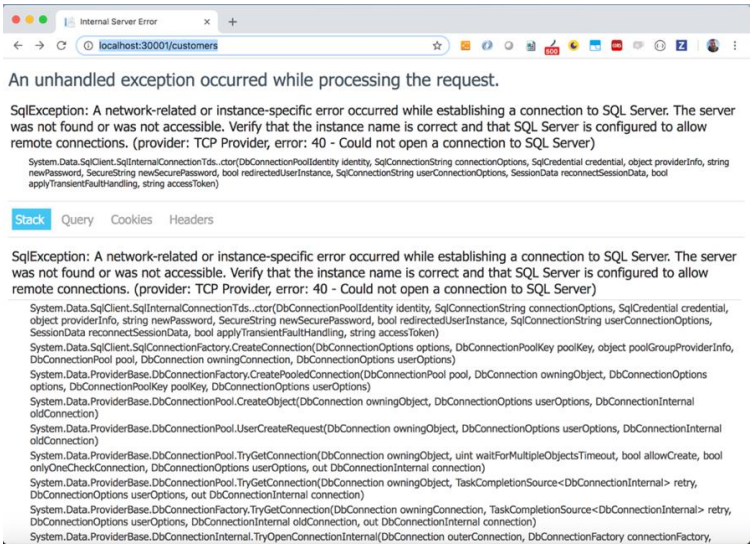

Figure 44: Customers page database error

We received an error because our database does not exist and we decided not to provide the database creation with the **`frontend`** image. In this scenario, the reasoning of this choice is clear: we have more than one replica and the script would have been executed by each of them.

This is a good opportunity to see another command of the **`kubectl`** tool that permits us to execute commands in a container guest of a pod. The command is **`kubectl exec`** and we can use it to connect us to the database container in the deployed pod (retrieve the pod name with the **`kubectl get pod`** command), as follows.

Code Listing 99: Command to connect to the bash of the container in the database pod

```
kubectl exec -it myapp-db-deployment-789b766d79-kxvcq -- /bin/bash
```

If you have more than one container in the target pod, you also need to specify the target container with the option **`--container`**. From the internal bash of the container, we can use the **`sqlcmd`** tool to create the database as follows.

Code Listing 96: Database creation command

```
/opt/mssql-tools/bin/sqlcmd -S localhost -U SA -P 'Password_123' -Q
'CREATE DATABASE [myapp]'
```

From another terminal window, we can copy the SQL script file created in Code Listing 65 to the container with the **`kubectl cp`** command, which is the equivalent of the **`docker cp`** command, as follows.

Code Listing 97: Command to copy the SQL script file to the container in the database pod

```
kubectl cp script.sql myapp-db-deployment-789b766d79-kxvcq:script.sql
```

Now we can update the database and launch the SQL script copied in the container, with the following command.

Code Listing 98: Command to execute a SQL script in the database container

```
/opt/mssql-tools/bin/sqlcmd -S localhost -U SA -P 'Password_123' -d myapp
-i script.sql
```

Returning to the browser and refreshing the page, we can see the correct result (Figure 45).

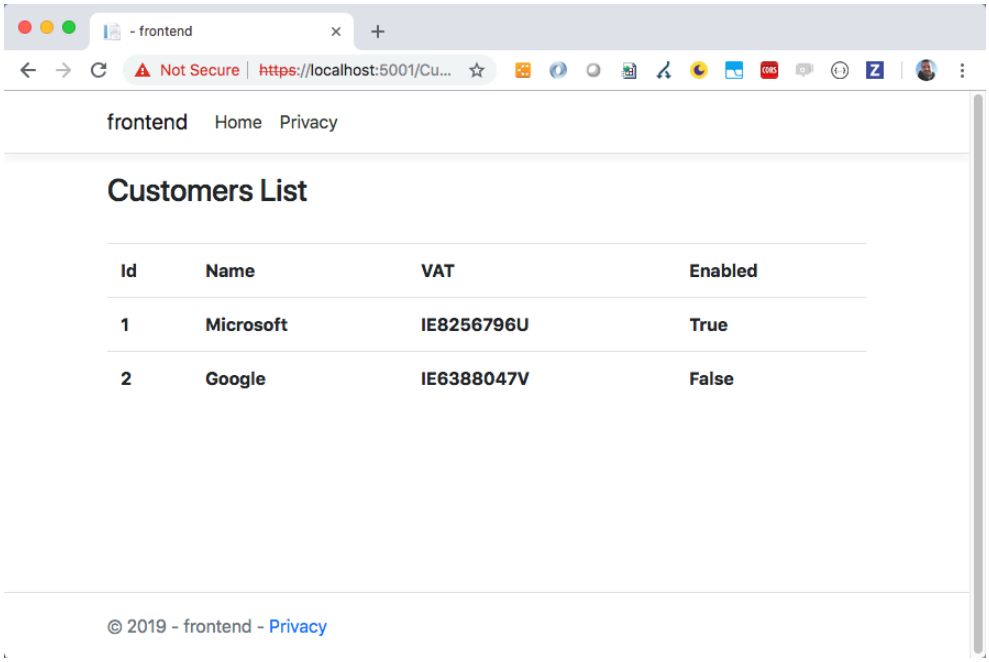

Figure 45: The customer page shown correctly from Kubernetes

If we try to delete and recreate the database, manually or with the **`kubectl apply`** command, we will lose our data. The images update (Microsoft releases an update of its images periodically to solve security problems and provide a new version of the products) is only one of the reasons to recreate the database, so we need to save the data in another place—and back it up! We also need to store the sensitive data in a secure way, like a connection string that contains a password, to ensure that only authorized users have access to it. Let's see how we can do that.

Kubernetes volumes and secrets

We have already seen how Docker can help us share data with containers using volumes. Kubernetes provides a similar mechanism with the same name, but we are talking about something that also needs to be managed in the production environment, so it is essential to provide more advanced storage solutions.

As you can imagine, the volumes are strictly related to the environment's configuration, so we can choose the available volumes' type depending on the infrastructure environment. The main cloud providers, for example, supply different types of volume for Kubernetes, like **`awsElasticBlockStore`** in AWS or **`AzureDisk`** and **`AzureFile`** in Azure.

Some kinds of volumes can be provisioned dynamically, while others cannot. This means that if dynamic provisioning is not supported in your case and you cannot create storage volumes on demand, then a cluster administrator has to create a volume for you. To abstract the concept of storage from its usage, Kubernetes provides the concept of *persistent volumes*. The name immediately communicates the idea of something that withstands the test of time. We can create two types of resources in Kubernetes to implement this concept:

- **PersistentVolume**: A resource created by a cluster administrator with specific characteristics and a lifecycle independent from your applications.
- **PersistentVolumeClaim**: A storage request by the user based on their needs, like specific access modes or sizes.

In this way, the user doesn't need to know the available persistent volumes in the cluster, but can simply say what is needed, and the cluster will provide the correct storage option.

Let's create our first **PersistentVolume**. We create a file named **volume.yml** and insert the following script.

Code Listing 99: HostPath PersistentVolume creation script

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-volume
  labels:
    type: local
spec:
  storageClassName: hostpath
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /mnt/data
```

With this script, we will create a **PersistentVolume** named **local-volume** with a label named **type** and the value **local**. Kubernetes will create a volume in the **/mnt/data** path of the cluster node, in read and write mode by a single Node, and with a size of 10 gigabytes.

The **storage** class is used to indicate the service level or other characteristics like the backup policies, and its purpose is to enable a user to claim the storage based on need. The **hostpath** value is a way to indicate that we want to use storage based on a cluster node path. This is a typical testing and development configuration; you don't use it in a production environment.

We can create the volume with the usual **kubectl create** command (for us, **kubectl create -f volume.yml**) and list the available volumes with the following commands.

Code Listing 100: Commands to list the persistent volumes

```
kubectl get persistentvolumes
kubectl get persistentvolume
kubectl get pv
```

You can see the results of the command in Figure 46.

```
Micheles-MacBook-Pro:kubernetes micheleaponte$ kubectl get pv
NAME           CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM     STORAGECLASS   REASON   AGE
local-volume   10Gi       RWO            Retain           Available             manual                  1h
```

Figure 46: Persistent volumes list

As you can see, the volume has the status **`Available`** and no claims, so we can claim it with a new script placed in a file named **`volume-claim.yml`**.

Code Listing 101: PersistentVolumeClaim script

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: local-volume-claim
spec:
  storageClassName: hostpath
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

With the creation of the resource (using the **`kubectl create -f volume-claim.yml`** command), we request three gigabytes of storage in read and write mode with the class **`hostpath`**. You can see the persistent volume claims with the following command.

Code Listing 102: Commands to list persistent volume claims

```
kubectl get persistentvolumeclaims
kubectl get persistentvolumeclaim
kubectl get pvc
```

By executing the **`kubectl get pv`** command again, we can also see the claim of the volume and its state changed to **`Bound`**.

```
Micheles-MacBook-Pro:kubernetes micheleaponte$ kubectl get pv
NAME           CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM                     STORAGECLASS   REASON   AGE
local-volume   10Gi       RWO            Retain           Bound    default/local-volume-claim   manual                  1h
Micheles-MacBook-Pro:kubernetes micheleaponte$ kubectl get pvc
NAME                 STATUS   VOLUME         CAPACITY   ACCESS MODES   STORAGECLASS   AGE
local-volume-claim   Bound    local-volume   10Gi       RWO            manual         4m
```

Figure 47: Persistent volume claim request info result

When the status of the claim is **`Bound`**, you can use it in your pods. In particular, we want to use this storage to save the SQL Server database, located in the **`/var/opt/mssql`** container path. To do this, return to the **`db-deployment.yml`** file and change the **`spec`** section of the script as follows.

Code Listing 103: Volume mounts in SqlServer deployment

```
    spec:
      containers:
        - name: myapp-database
          image: mcr.microsoft.com/mssql/server
          ports:
            - containerPort: 1433
          env:
            - name: "ACCEPT_EULA"
              value: "Y"
            - name: "SA_PASSWORD"
              value: "Password_1234"
            - name: "MSSQL_PID"
              value: "Express"
          volumeMounts:
            - name: mssqldb
              mountPath: /var/opt/mssql
      volumes:
      - name: mssqldb
        persistentVolumeClaim:
          claimName: local-volume-claim
```

As you can see, the container configuration is changed. In the new section, **volumeMounts**, we indicate that the path **/var/opt/mssql** will be mounted on a volume named **mssqldb**. The second change is about the section volumes, where we use a persistent volume claim to map a persistent volume with the container. In this case, we use the **local-volume-claim**, previously created, in a logical volume named **mssqldb**. Thanks to the equality of the volume mount and the volume name, we obtain the expected result: the container database will be created in the volume instead of in the container.

With the same mechanism, we can also map the database backup folder, or a folder for the images uploaded by users using the front end, solving the problem of storing data without the fear of losing data. In the next chapter, with the administration of a Kubernetes cluster on Azure, we will see some advanced features of volumes that we cannot analyze now because of the limitations of the local environment.

Another problem that we can solve now is the storage of sensitive data in a Kubernetes cluster, like the password to access our database. To do this, Kubernetes provides a tool known as a *secret*, which we already glimpsed when we talked about the token to access the Kubernetes Dashboard.

A secret is a container for a small amount of sensitive data, which our pods can access in a secure way with a file placed at runtime in a volume or using an environment variable. Kubernetes shares this information with the pod, but it doesn't worry about the content of the data, so it doesn't provide strong encryption. Secrets are key/value pairs where both key and value are strings, so we can encrypt the value in any way we want. Typically, a secret uses a Base64 encoding, because the result string does not contain reserved characters and we can store small files and images in a string in this way.

To create a secret for our database password, we can use the following command.

Code Listing 104: Command to create a secret from a literal

```
kubectl create secret generic mssql --from-
literal=SA_PASSWORD="Password_123"
```

Now we only need to change the **SA_PASSWORD** environment variable as follows.

Code Listing 105: Passing secret in an environment variable

```
env:
    - name: "ACCEPT_EULA"
      value: "Y"
    - name: "SA_PASSWORD"
      valueFrom:
          secretKeyRef:
              name: mssql
              key: SA_PASSWORD
```

We can use the same command for the connection string in the **frontend** or change the code to use the same **Secret** when composing the connection string.

If you can store the password using the Base64 encoding, you can retrieve the corresponding value of the **Password_123** string with the following command.

Code Listing 106: Command to encode a string in Base64

```
echo -n Password_123 | base64
```

The result is **UGFzc3dvcmRfMTIz**, which you can use in the command of Code Listing 109. You can also create a secret with a YAML file.

Code Listing 107: YAML script to create a secret

```
apiVersion: v1
kind: Secret
metadata:
  name: mssql
type: Opaque
data:
  SA_PASSWORD: UGFzc3dvcmRfMTIz
```

To get the created secrets, you can use the command **kubectl get secrets**, which shows the following output.

```
Micheles-MacBook-Pro:kubernetes micheleaponte$ kubectl get secrets
NAME                  TYPE                                  DATA      AGE
default-token-z9xlk   kubernetes.io/service-account-token   3         19d
mssql                 Opaque                                1         21m
```

Figure 48: Secrets list from terminal

As you can see in Figure 48, the value of the secrets is not shown for security reasons. To show the value of the **`mssql`** secret, you can use the following command.

Code Listing 108: Command to show a secret value

```
kubectl get secret mssql -o yaml
```

You can see the result of the command in the Figure 49.

```
Micheles-MacBook-Pro:kubernetes micheleaponte$ kubectl get secret mssql -o yaml
apiVersion: v1
data:
  SA_PASSWORD: UGFzc3dvcmRfMTIz
kind: Secret
metadata:
  creationTimestamp: 2019-02-16T16:12:56Z
  name: mssql
  namespace: default
  resourceVersion: "406840"
  selfLink: /api/v1/namespaces/default/secrets/mssql
  uid: b8aac17f-3205-11e9-8ff6-025000000001
type: Opaque
```

Figure 49: Output of the command to show secret value

To decode the value, you can use the following command.

Code Listing 109: Command to decode a Base64 string

```
echo UGFzc3dvcmRfMTIz | base64 --decode
```

If you want to access a secret from a volume mounted with the pod, you need to specify a volume from a secret, as follows.

Code Listing 110: Script to mount volume from secret

```
        volumeMounts:
          - name: mssqldb
            mountPath: /var/opt/mssql
          - name: secret-volume
            mountPath: /etc/secretVolume
      volumes:
      - name: mssqldb
```

```
        persistentVolumeClaim:
          claimName: local-volume-claim
      - name: secret-volume
        secret:
          secretName: mssql
```

In Code Listing 115, we mounted the secret **mssql** in the container path **/etc/secretVolume**, so if you go into the container, you can see the folder **secretVolume** in the **etc** system folder, where the system created the **SA_PASSWORD** file that contains the password **Password_123** (Figure 50).

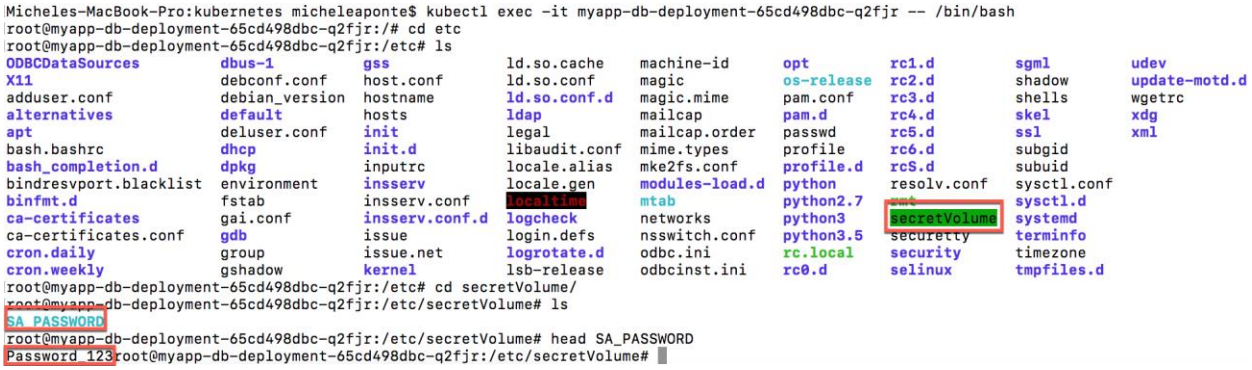

Figure 50: The secret mounted as volume in the container

This mechanism is particularly interesting when you place files in secrets using the option **--from-file** with the **create secret** command, as follows.

Code Listing 111: Command to create a secret from a file

```
kubectl create secret generic mssql --from-file=./<filename>
```

Namespaces and resource quotas

When you install a new Kubernetes cluster, the system creates a default namespace to include all items under a logical name. If you try to create two items with the same name, the system responds with an error. You won't have this problem if you work on small projects or with a small team, but you can create custom namespaces to separate your projects and limit the resources available for each of them.

You can obtain the available namespaces in your cluster with the following command.

Code Listing 112: Command to obtain the available namespaces

```
kubectl get namespaces
```

You can see the result of this command in Figure 51.

```
Micheles-MBP:~ micheleaponte$ kubectl get namespaces
NAME          STATUS    AGE
default       Active    27d
docker        Active    27d
kube-public   Active    27d
kube-system   Active    27d
Micheles-MBP:~ micheleaponte$
```

Figure 51: Available namespaces in Kubernetes

As you can see, there are also some system namespaces, used by Kubernetes to manage system objects. If you want to create a new namespace for the **myapp** application, you can use the following command.

Code Listing 113: Command to create a new namespace

```
kubectl create namespace myapp
```

From this moment, if you want to create Kubernetes objects like pods or deployments, you need to add the **--namespace=myapp** in all **kubectl create** commands. If you are lazy like me, you can change the current context for your commands with the following command.

Code Listing 114: Command to set current namespace

```
kubectl config set-context $(kubectl config current-context) –
namespace=myapp
```

This command sets your current context namespace property to the value **myapp**; the **$(kubectl config current-context)** instruction gets the cluster name (**docker-for-desktop** in our case). You can see the result of your configuration with the command **kubectl config view**, which shows you the current configuration.

```
Micheles-MBP:~ micheleaponte$ kubectl config view
apiVersion: v1
clusters:
- cluster:
    insecure-skip-tls-verify: true
    server: https://localhost:6443
  name: docker-for-desktop-cluster
contexts:
- context:
    cluster: docker-for-desktop-cluster
    namespace: myapp
    user: docker-for-desktop
  name: docker-for-desktop
current-context: docker-for-desktop
kind: Config
preferences: {}
users:
- name: docker-for-desktop
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
Micheles-MBP:~ micheleaponte$
```

Figure 52: Current configuration context

A namespace is like a virtual cluster over a physical Kubernetes one, but the best practice is to use multiple namespaces only if you need them. This practice is important because multiple namespaces can complicate the management of your environment. A good reason to use multiple namespaces is to limit resource quota for the cluster objects like CPU memory. You can easily create a **`ResourceQuota`** with a simple file named **`resourcequota.yml`** and insert this script.

Code Listing 115: Sample resource quota script

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: myapp-quota
spec:
  hard:
    limits.cpu: 1
    limits.memory: 2Gi
```

You can create the resource quota with the following command.

Code Listing 116: Command to create a resource quota

```
kubectl create -f resourcequota.yml
```

As you can see, the option **--namespace** is not specified, so this command is applied to the namespace of the current context (**myapp** in our case). From this moment on, all the pods in the namespace can use only one CPU and a maximum of two gigabytes of memory.

The namespaces also influence the internal networking of the cluster. Kubernetes provides an internal DNS that is automatically configured to help us translate internal IP addresses into logical names. When you create a service, the system allocates an IP address, but a DNS name is also created, composed of the service name with the namespace and the constant **svc.cluster.local**.

Code Listing 117: DNS name rule

```
<myservicename>.<mynamespace>.svc.cluster.local
```

If you create a service in the namespace **myapp** and name it **frontend**, the DNS name will be **frontend.myapp.svc.cluster.local**. Very simple and effective.

Horizontal Pod Autoscaler

Another very cool feature of Kubernetes is the auto scale. You can scale the number of pod instances in your deployment based on system metrics like the CPU percentage usage. This is a typical horizontal scaling technique and it is very useful if the traffic to your application is not constant over time.

If you want to scale the pod instances of the deployment named **myapp-deployment** from 1 to 5 when the use of the CPU is over 50 percent, you can use the following command.

Code Listing 118: Command to enable auto scale of a deployment

```
kubectl autoscale deployment myapp-deployment --cpu-percent=50 --min=1 --
max=5
```

This command configures the Horizontal Pod Autoscaler (HPA), which you can monitor with the following command.

Code Listing 119: Command to show Horizontal Pod Autoscaler settings

```
kubectl get hpa
```

The HPA is a controller that checks the resources defined in a configuration (in our case, the CPU percentage) in a control loop. By default, the loop runs every 15 seconds, but you can control it with the **--horizontal-pod-autoscaler-sync-period** setting.

When the resource metrics match the HPA definition, the controller reacts to implement the desired state, creating or deleting pods. There are various metrics that you can use, and from the 1.6 version, Kubernetes permits the definition of multiple and custom metrics.

Unfortunately, the metrics are not present in the local cluster, as you can see from the **`kubectl describe hpa`** command.

```
Micheles-MBP:kubernetes micheleaponte$ kubectl get hpa
NAME               REFERENCE                     TARGETS         MINPODS   MAXPODS   REPLICAS   AGE
myapp-deployment   Deployment/myapp-deployment   <unknown>/50%   1         5         3          1m
Micheles-MBP:kubernetes micheleaponte$ kubectl describe hpa
Name:                                                  myapp-deployment
Namespace:                                             default
Labels:                                                <none>
Annotations:                                           <none>
CreationTimestamp:                                     Sun, 24 Feb 2019 10:24:30 +0100
Reference:                                             Deployment/myapp-deployment
Metrics:                                               ( current / target )
  resource cpu on pods  (as a percentage of request):  <unknown> / 50%
Min replicas:                                          1
Max replicas:                                          5
Conditions:
  Type           Status  Reason                   Message
  ----           ------  ------                   -------
  AbleToScale    True    SucceededGetScale        the HPA controller was able to get the target's cu
rrent scale
  ScalingActive  False   FailedGetResourceMetric  the HPA was unable to compute the replica count: u
nable to get metrics for resource cpu: unable to fetch metrics from resource metrics API: the server
 could not find the requested resource (get pods.metrics.k8s.io)
Events:
  Type     Reason                         Age                 From                       Message
  ----     ------                         ----                ----                       -------
  Warning  FailedComputeMetricsReplicas   21m (x13 over 27m)  horizontal-pod-autoscaler  failed to ge
t cpu utilization: unable to get metrics for resource cpu: unable to fetch metrics from resource met
rics API: the server could not find the requested resource (get pods.metrics.k8s.io)
  Warning  FailedGetResourceMetric        2m (x51 over 27m)   horizontal-pod-autoscaler  unable to ge
t metrics for resource cpu: unable to fetch metrics from resource metrics API: the server could not
find the requested resource (get pods.metrics.k8s.io)
```

Figure 53: Command to describe the HPA status

You can try this feature with **`minikube`**, adding a specific plugin to enable metrics, or by using a cluster in the cloud like Azure Kubernetes Services.

Chapter 5 Production Time

Azure Kubernetes Service

Installing, managing, and updating a Kubernetes cluster can be very difficult if you don't have experience with the administration of a Linux server. The best solution is probably to entrust these tasks to some provider, and the good news is that you can choose from many solutions. The main cloud providers provide a Kubernetes solution as a service, and Microsoft's Azure Kubernetes Service (AKS) is one of them.

The Kubernetes cluster management is free in Azure, you will pay only for the nodes (virtual machines) and the public, reserved, and balanced IP that you need. If you don't have an Azure account, you can register for a free account here and receive a $200 credit for the first 30 days, and 25 free services forever. A credit card will be required only to check your identity.

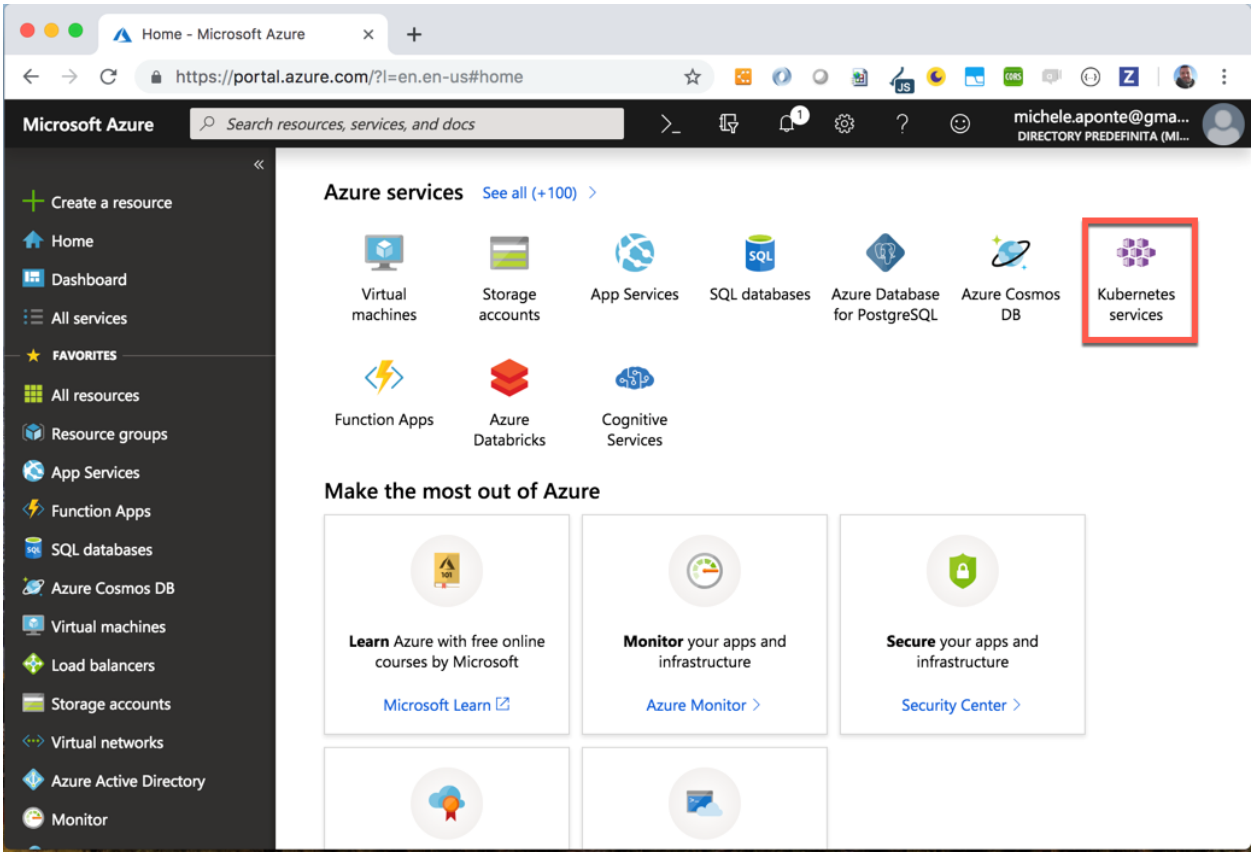

Figure 54: Azure Portal after the registration of a free account

In Figure 54 you can see the Azure Portal after the free registration. You can create a new Kubernetes service by selecting the service from the list, searching for it in the search box, or clicking **Create a resource**.

The creation wizard requires the main information for the cluster, starting with the subscription (you can have more than one subscription in Azure) and the resource group, a logical container to group Azure services. It's a good idea to have a specific resource group for our cluster and the services connected to it to simplify the management and the monitoring of tasks. As shown in Figure 55, select the link **Create New** to create a new resource group with the name **Kubernetes_Cluster**.

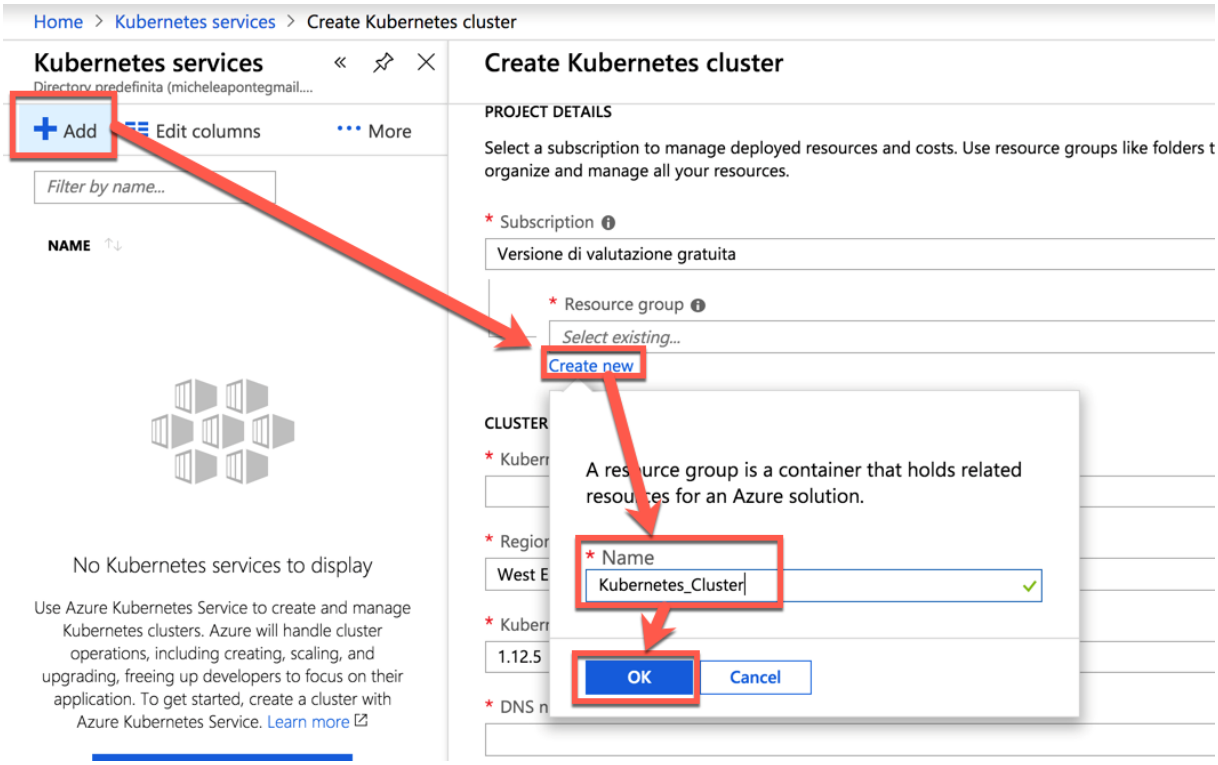

Figure 55: Azure Kubernetes Service wizard

Choose a name for your cluster that is unique in your resource group, and a DNS name prefix that is unique in Azure, because you will use it to connect to your cluster. We can use **myappk8s** for both our values. You can choose the Azure region where the cluster will be created; choose a region near your user to improve the network performance. If you have any requirement on different versions of Kubernetes, you can select it in the selection box. If you don't have any constraints, choose the latest version.

The last information to insert in the first step of the cluster creation wizard is the nodes information. This choice influences the billing, so we can start with the cheaper size of virtual machine for the nodes. Click the **Change size** link (Figure 56) to change the size of the virtual machines for your nodes to **B2s**.

The next steps are:

1. **Authentication**: Manage the resources attached to the cluster.
2. **Networking**: Customize the networking options for the cluster.
3. **Monitoring**: Enable metrics and monitoring options for the cluster nodes.
4. **Tags**: Categorize resource with custom name/value pairs.
5. **Review + create**: Review and analyze your choices before the cluster creation.

If you don't have any particular constraints, you can leave the default values for these steps and click **Review + create**.

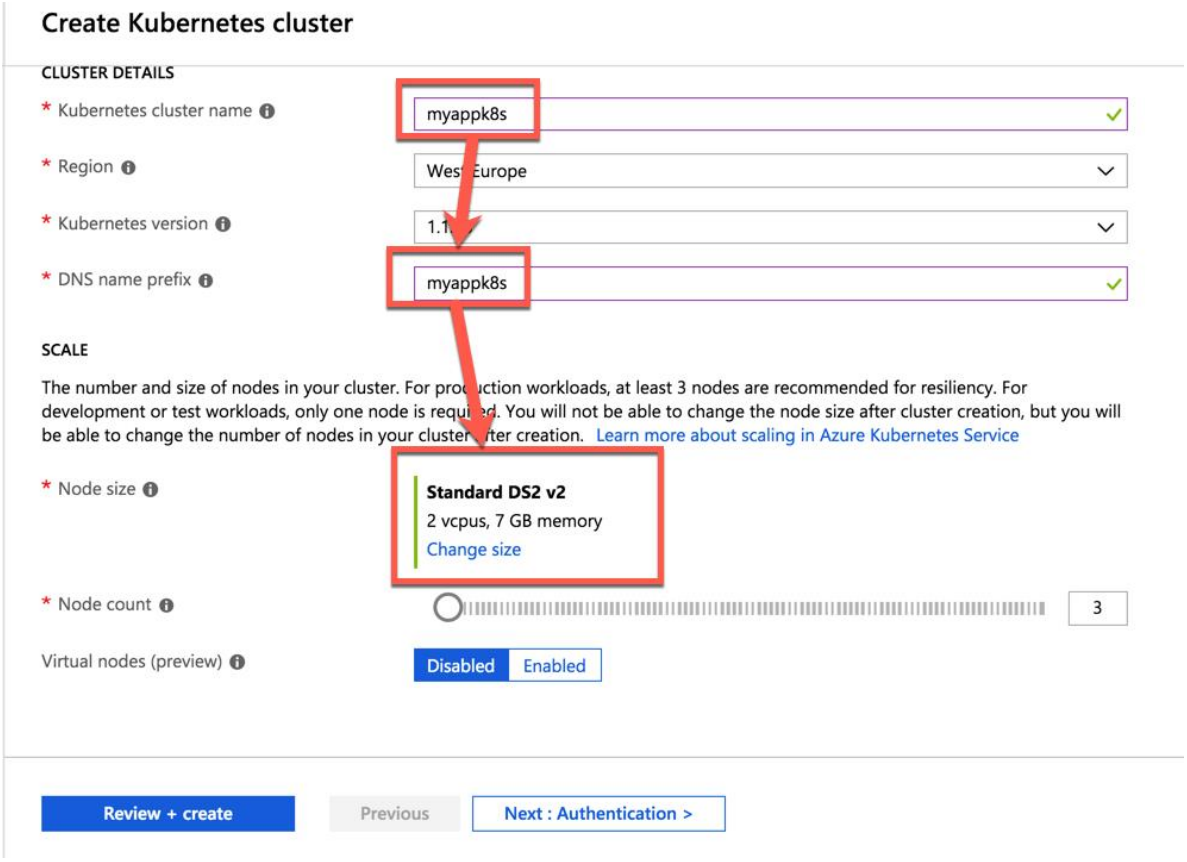

Figure 56: First step of AKS creation wizard

The cluster creation requires several minutes. Many Azure objects will be configured and connected to enable the service. If all operations end without errors, a notification alerts you on the operation success.

It's time to connect our **`kubectl`** to the Azure Kubernetes Service. To do this, we need to install the Azure Command Line Interface (Azure CLI) by Homebrew (if you are on MacOS) or download and execute the Windows installation package (if you are on Windows).

Code Listing 120: Command to install Azure CLI on Mac with Homebrew

```
brew install azure-cli
```

After the installation of the CLI, you can execute commands with **`az`** from the terminal. The first command to execute is **`login`**, as follows.

Code Listing 121: Command to login Azure CLI to Azure subscription

```
az login
```

The command opens the browser to permit you to use the Azure account credentials to complete the operation, and return to the terminal with the operation result.

```
Micheles-MBP:kubernetes micheleaponte$ az login
Note, we have launched a browser for you to login. For old experience with device code, use "az logi
n --use-device-code"
You have logged in. Now let us find all the subscriptions to which you have access...
[
  {
    "cloudName": "AzureCloud",
    "id": "72035500-3969-4b9c-b6fb-74b6183c2ee6",
    "isDefault": true,
    "name": "Versione di valutazione gratuita",
    "state": "Enabled",
    "tenantId": "208fd105-3219-4d7c-a7c7-56e84cdbaf39",
    "user": {
      "name": "michele.aponte@gmail.com",
      "type": "user"
    }
  }
]
Micheles-MBP:kubernetes micheleaponte$
```

Figure 57: The result of the Azure CLI login command

Now that you are connected with the Azure subscription, you can use the **`aks get-credentials`** command to update the Kubernetes current context.

Code Listing 122: Command to merge AKS credential with the current Kubernetes context

```
az aks get-credentials --resource-group Kubernetes_Cluster --name
myappk8s
```

You can see the result in Figure 58.

```
Micheles-MBP:kubernetes micheleaponte$ az aks get-credentials --resource-group Kubernetes_Cluster --
name myappk8s
Merged "myappk8s" as current context in /Users/micheleaponte/.kube/config
Micheles-MBP:kubernetes micheleaponte$
```

Figure 58: The aks get-credential command result

After the command execution, the **`kubectl`** tool is connected to the AKS cluster. You can return to your local environment by using the Docker tool, as shown in Figure 59.

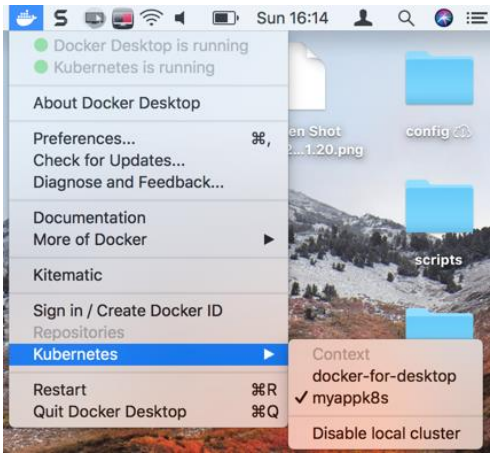

Figure 59: Docker tool to change the target Kubernetes cluster

If you try to execute the **`kubectl get nodes`** command, the result is similar to Figure 60, which shows the two nodes of the Azure Kubernetes Service cluster.

```
Micheles-MBP:kubernetes micheleaponte$ kubectl get nodes
NAME                       STATUS    ROLES     AGE       VERSION
aks-agentpool-26198934-0   Ready     agent     1h        v1.12.5
aks-agentpool-26198934-1   Ready     agent     1h        v1.12.5
Micheles-MBP:kubernetes micheleaponte$
```

Figure 60: The Azure Kubernetes Service Nodes from the terminal command

It's time to deploy our application on Azure! You just need to execute the same command used for local deployment: **`kubectl create -f deployment.yml`**. After a few seconds, you can execute the **`kubectl get pods`**, with the following result.

```
Micheles-MBP:kubernetes micheleaponte$ kubectl get pods
NAME                                 READY     STATUS    RESTARTS   AGE
myapp-deployment-68bf66f79f-8p9cg    1/1       Running   0          21m
myapp-deployment-68bf66f79f-bln5c    1/1       Running   0          21m
myapp-deployment-68bf66f79f-xnxdx    1/1       Running   0          21m
Micheles-MBP:kubernetes micheleaponte$
```

Figure 61: The front-end pod deployed on Azure Kubernetes Service

To expose the application, we need a service. We can use a **`LoadBalancer`** service type. Create a copy of **`service.yml`**, rename it **`service-aks.yml`**, and modify the script as follows.

Code Listing 123: LoadBalancer service to expose front-end application from Azure

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-svc
spec:
  selector:
    app: myapp
```

```
  type: LoadBalancer
  ports:
  - port: 80
```

Execute the script with the **`kubectl create -f service-aks.yml`** command and execute the **`kubectl get service myapp-svc --watch`** command. The **`--watch`** option permits us to monitor the command results until the public IP is assigned from Azure.

```
Micheles-MBP:kubernetes micheleaponte$ kubectl create -f service-aks.yml
service "myapp-svc" created
Micheles-MBP:kubernetes micheleaponte$ kubectl get service myapp-svc --watch
NAME        TYPE           CLUSTER-IP   EXTERNAL-IP    PORT(S)        AGE
myapp-svc   LoadBalancer   10.0.11.34   <pending>      80:30155/TCP   6s
myapp-svc   LoadBalancer   10.0.11.34   51.137.106.191   80:30155/TCP   52s
```

Figure 62: Command to create service on Azure and waiting for public IP assignment

As you can see in Figure 62, after 52 seconds Azure assigns an external IP to the service, which we can use to navigate to the application.

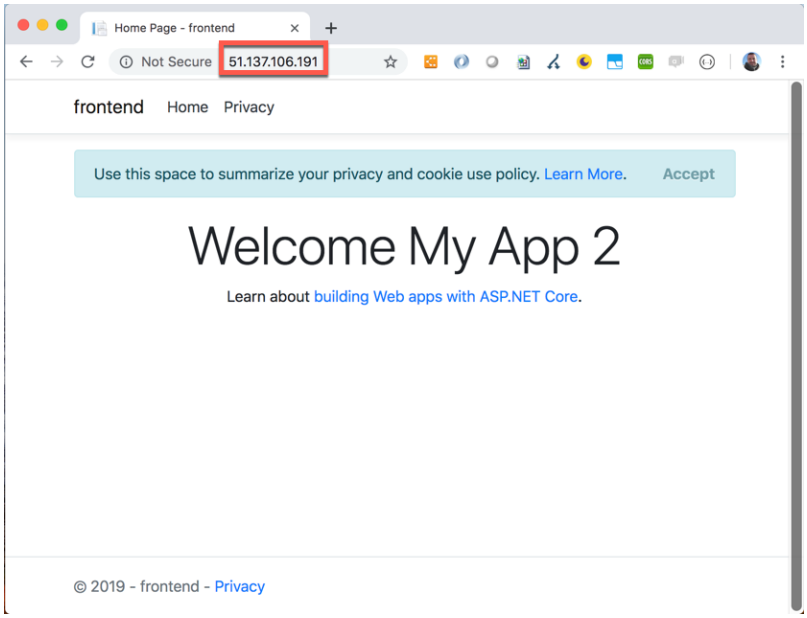

Figure 63: The front-end application exposed from Azure

Now, the database needs a volume to store the physical file with the data. In Azure, you have specific services to store data. Azure disks is one of them, and to use it with Kubernetes, we need to create a **`StorageClass`**. Let's create a **`storageclass-aks.yml`** and insert the following script.

Code Listing 124: Script to create a StorageClass for Azure disks

```
kind: StorageClass
apiVersion: storage.k8s.io/v1beta1
```

```
metadata:
     name: azure-disk
provisioner: kubernetes.io/azure-disk
parameters:
  storageaccounttype: Standard_LRS
  kind: Managed
```

After the execution of this script (with the **`kubectl create -f storageclass-aks.yml`** command), you can use the name **`azure-disk`** to create a persistent volume claim for your volumes. In this case, we do not need to create a persistent volume, because Azure Disks supports the dynamic provisioning. Create a file named **`volume-claim-aks.yml`** with the following script.

Code Listing 125: Script to create a PersistentVolumeClaim for Azure Disk

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: azure-volume-claim
spec:
  storageClassName: azure-disk
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 8Gi
```

Remember to create a secret for the database password (**`kubectl create secret generic mssql --from-literal=SA_PASSWORD="Password_123"`**) and modify the database deployment as follows.

Code Listing 126: Database deployment script changes to set the Azure volume claim

```
spec:
      containers:
        - name: myapp-database
          image: mcr.microsoft.com/mssql/server
          ports:
            - containerPort: 1433
          env:
            - name: "ACCEPT_EULA"
              value: "Y"
            - name: "SA_PASSWORD"
              valueFrom:
                  secretKeyRef:
                      name: mssql
                      key: SA_PASSWORD
            - name: "MSSQL_PID"
```

```
            value: "Express"
          volumeMounts:
            - name: mssqldb
              mountPath: /var/opt/mssql
      volumes:
      - name: mssqldb
        persistentVolumeClaim:
          claimName: azure-volume-claim
```

We are ready to create the deployment (**`kubectl create -f db-deployment-aks.yml`**) and wait for the pod creation. In the meantime, we can create a **`LoadBalancer`** service to access the database and initialize it with the SQL script created with the front end. Create a new file named **`db-service.yml`** with the following script.

Code Listing 127: Script to create a LoadBalancer service for the database

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-db-svc
spec:
  selector:
    app: myapp-db
  ports:
    - protocol: TCP
      port: 1433
      targetPort: 1433
  type: LoadBalancer
```

After the execution of this script (**`kubectl create -f db-service.yml`**), the system allocates an external IP that we can use with our preferred database management tool. You can use **`sqlcmd`** to execute the script, but if you want a free management tool that's also available on MacOS, you can download and install Azure Data Studio.

In my case, Azure provides me with the IP **`23.97.160.75`**, which I can use as server for the connection. As you can see in Figure 65, Azure Data Studio is very similar to Visual Studio Code, so you should be familiar with it.

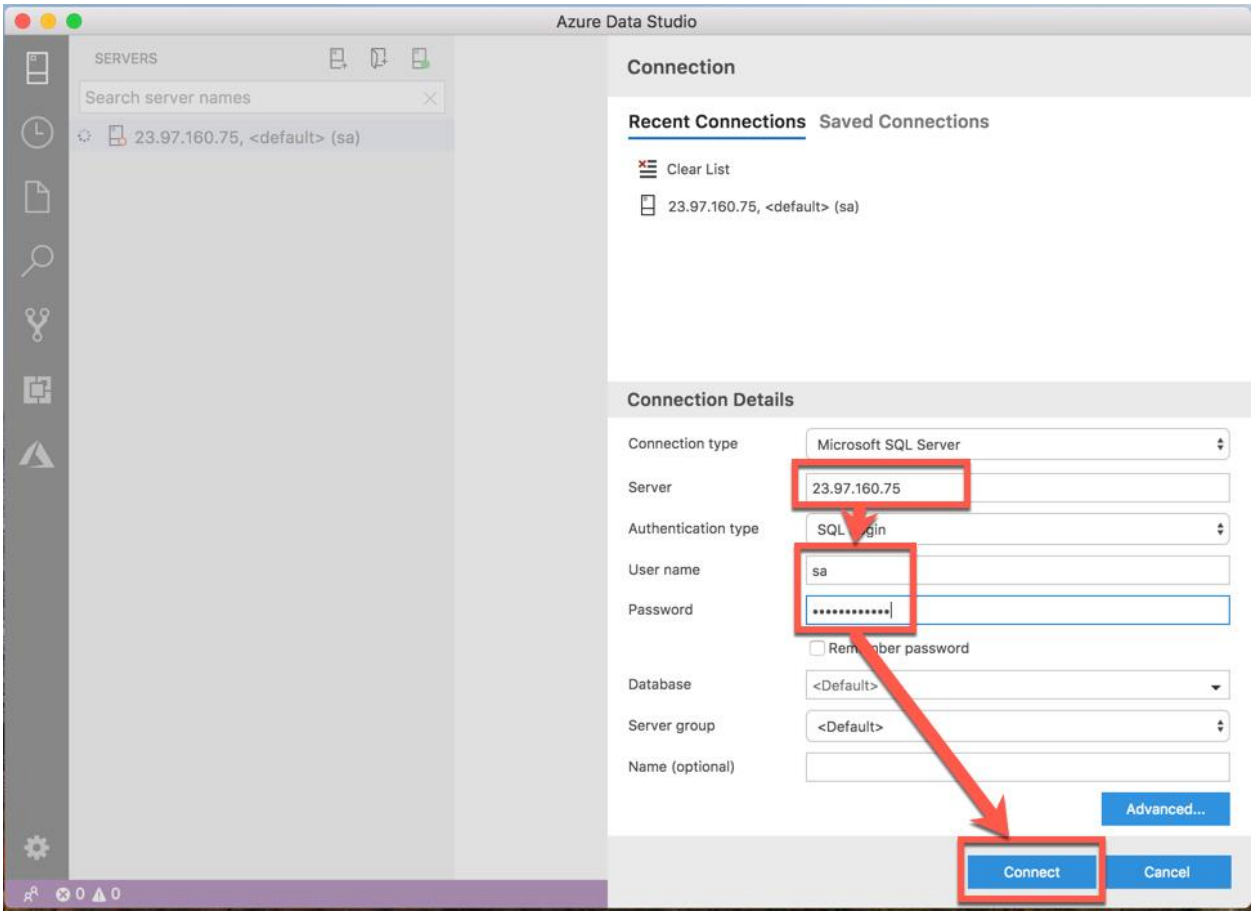

Figure 64: Azure Data Studio connection form

Create a new query to execute the following script.

Code Listing 128: SQL command to create the myapp database

```
CREATE DATABASE myapp
```

We can now open the file **`script.sql`** and execute it on the **`myapp`** database to create the schema and fill the tables with the initial data. You can see the Azure Data Studio dashboard with the connection info and the **`myapp`** database created in Figure 65.

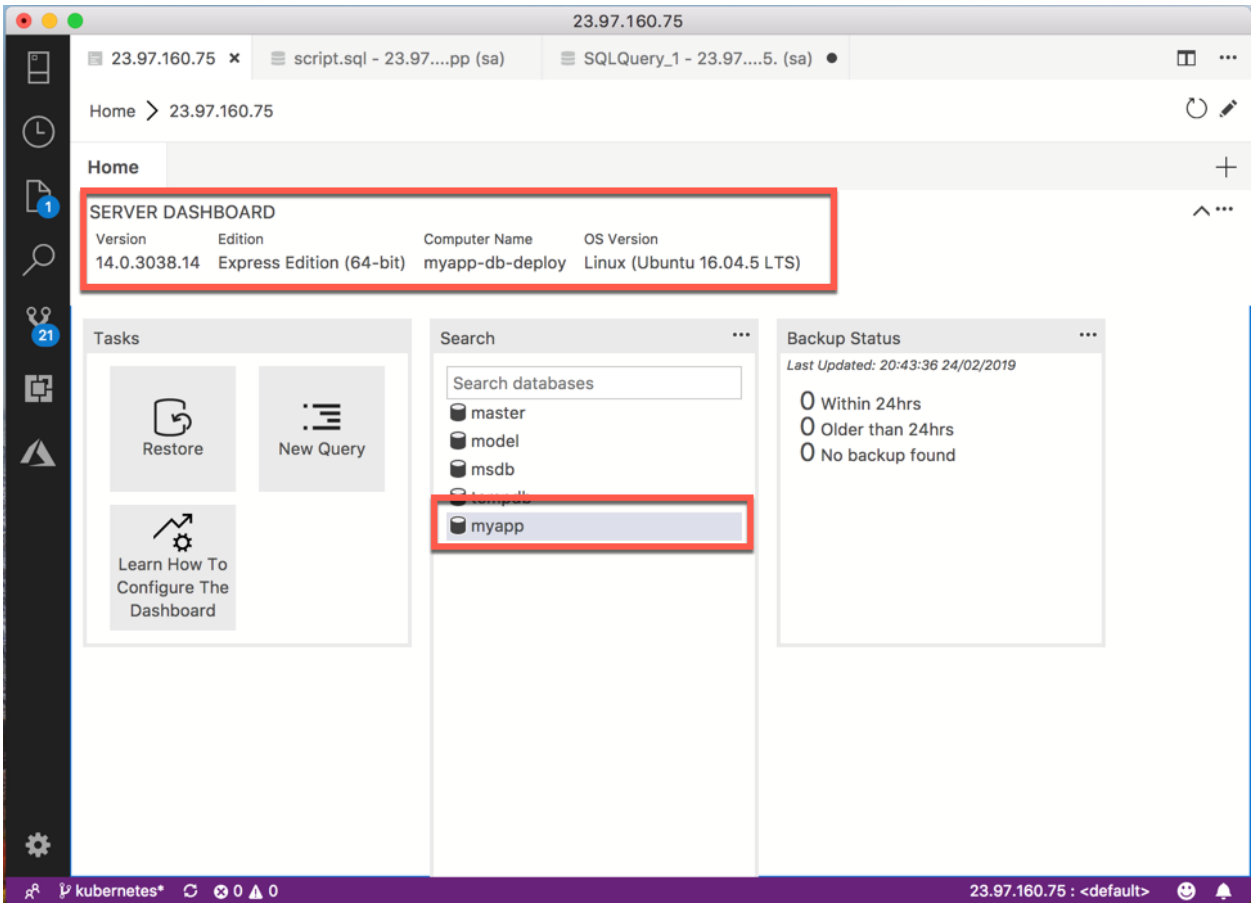

Figure 65: Azure Data Studio dashboard with the connection info and the created database

After the database creation and configuration, you can delete the **`LoadBalancer`** service and create a **`ClusterIP`** service, accessible only from within the cluster. You can recreate the **`LoadBalancer`** service when you need to access to the database for management purposes. If the service is named **`myapp-db-svc`**, the internal DNS name will be **`myapp-db-svc.default.svc.cluster.local`**. You can use it to change the connection string in your **`deployment.yml`**.

Code Listing 129: Front-end deployment connection string

```
    env:
    - name: SQLSERVER_CONNECTIONSTRING
      value: "Server=myapp-db-
svc.default.svc.cluster.local;Database=myapp;User=sa;Password=Password_123;
MultipleActiveResultSets=true;"
```

Apply the change with the **`kubectl apply -f deployment.yaml --record`** command and navigate to the customer page to see the result of your work: the application is running from the Azure Kubernetes Service cluster!

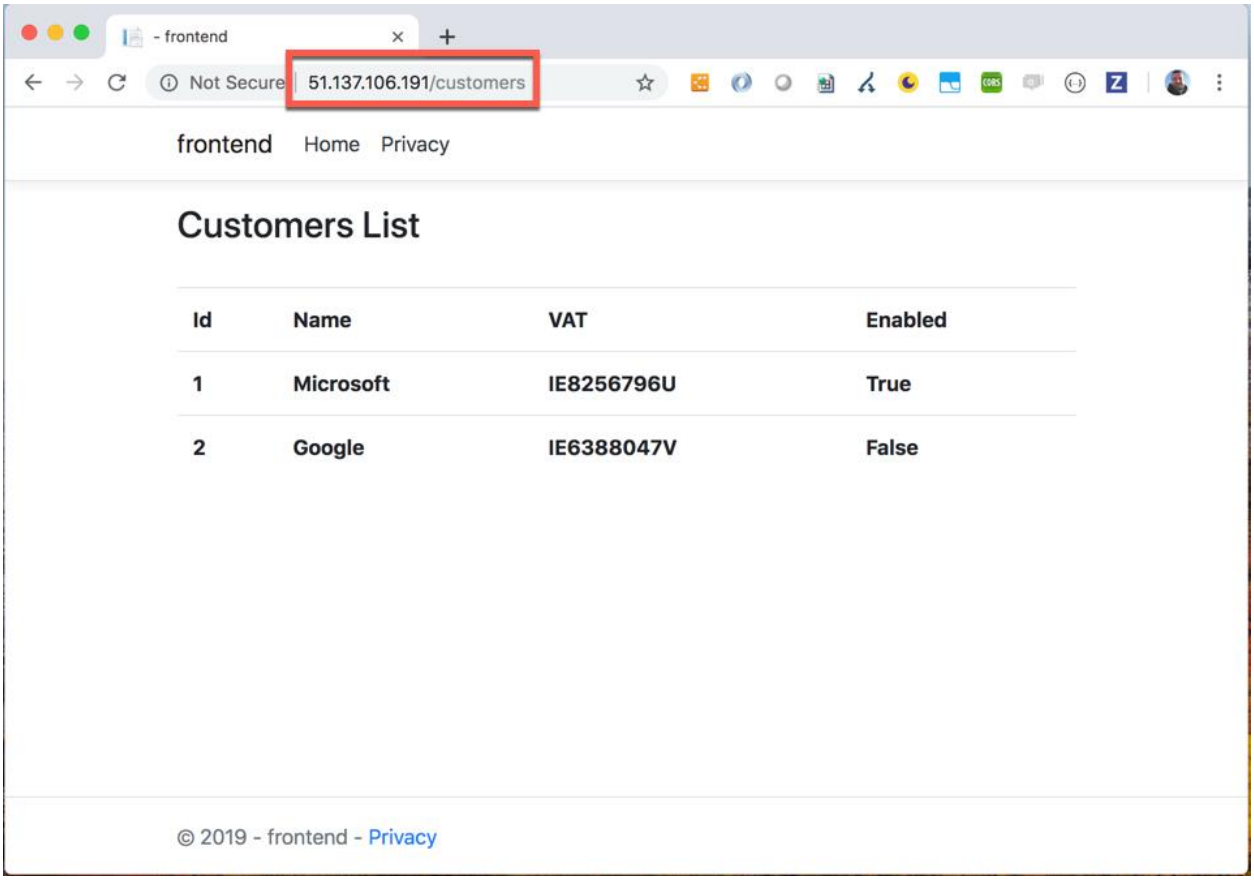

Figure 66: The customer page of the application running from Azure Kubernetes Service

Monitoring a Kubernetes cluster

You can also use the Kubernetes Dashboard with the Azure cluster, but this tool is not the best for managing a cluster in production. Azure provides three default monitoring tools:

- **Insight**: Provides many metrics and details on the cluster.
- **Metrics**: Creates new graphics on available metrics in a graphical environment. It helps to select the parameters and export the results in Excel.
- **Logs**: A query builder to analyze the cluster logs.

If you are on Azure, these tools are very useful both for a first look and an in-depth analysis of the cluster state. The creation of a new chart in the Metrics section is in preview, but you can try it—it has a very simple user experience.

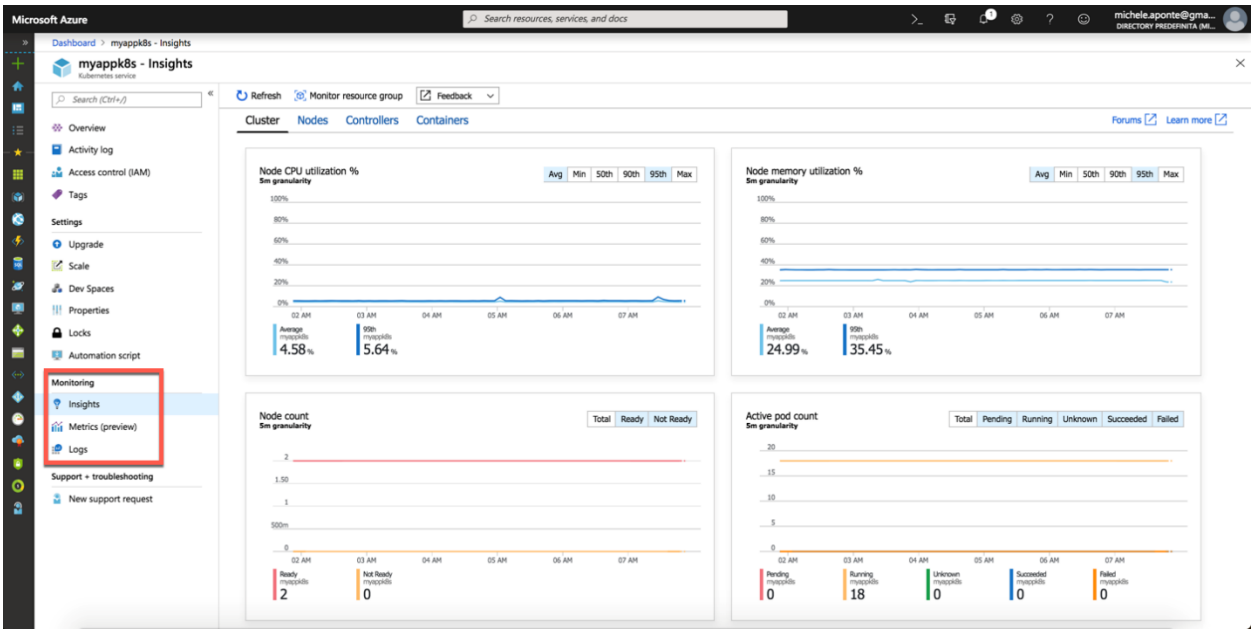

Figure 67: Azure Kubernetes Service monitoring tools

Azure provides tools for monitoring all services and you can access them from the main menu by selecting the **Monitoring** choice. The most well-known service for monitoring in Azure is Application Insights, which you can integrate into your code to provide any information that will be collected and analyzed for you by a machine learning system.

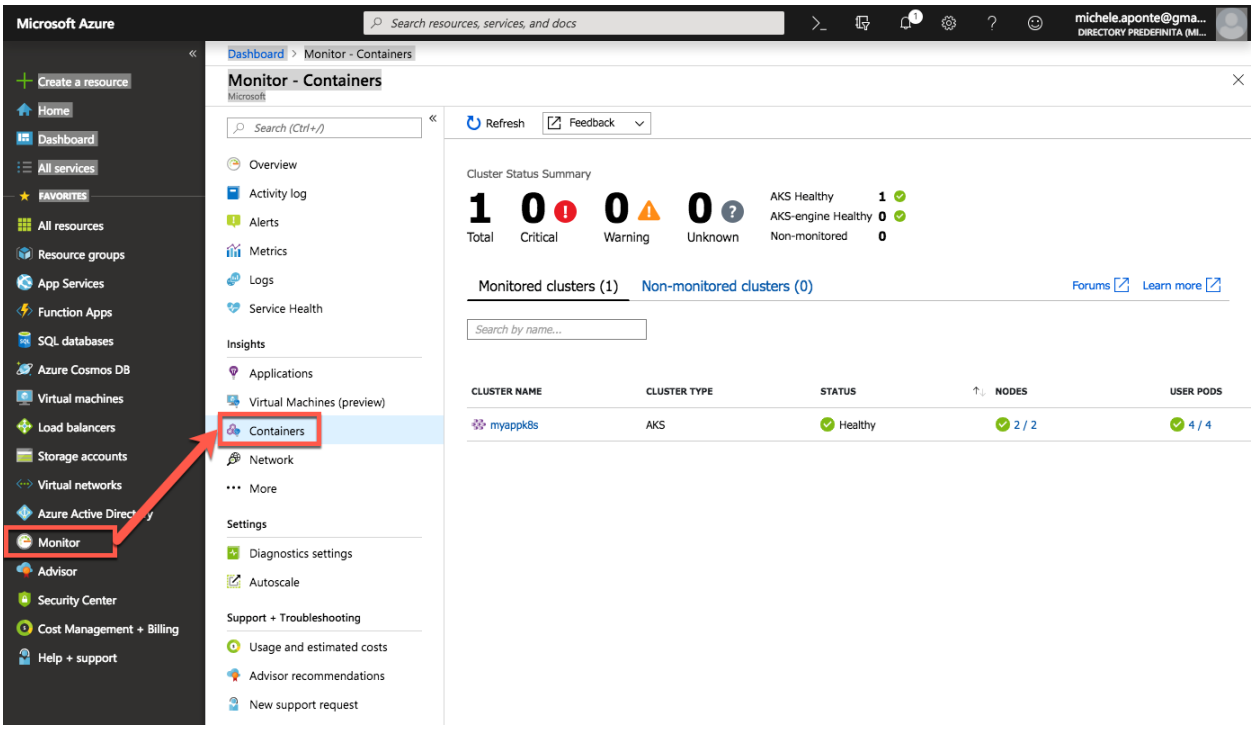

Figure 68: Monitor containers from Azure portal

If you have more than one cluster, you can see the monitoring values of your containers in a single page that shows you an overall view of the status of the container (Figure 68). A very cool feature of Azure's monitoring tools is the possibility to create an alert based on particular situations. From the Alert section, you can create your rule with a simple wizard (Figure 69).

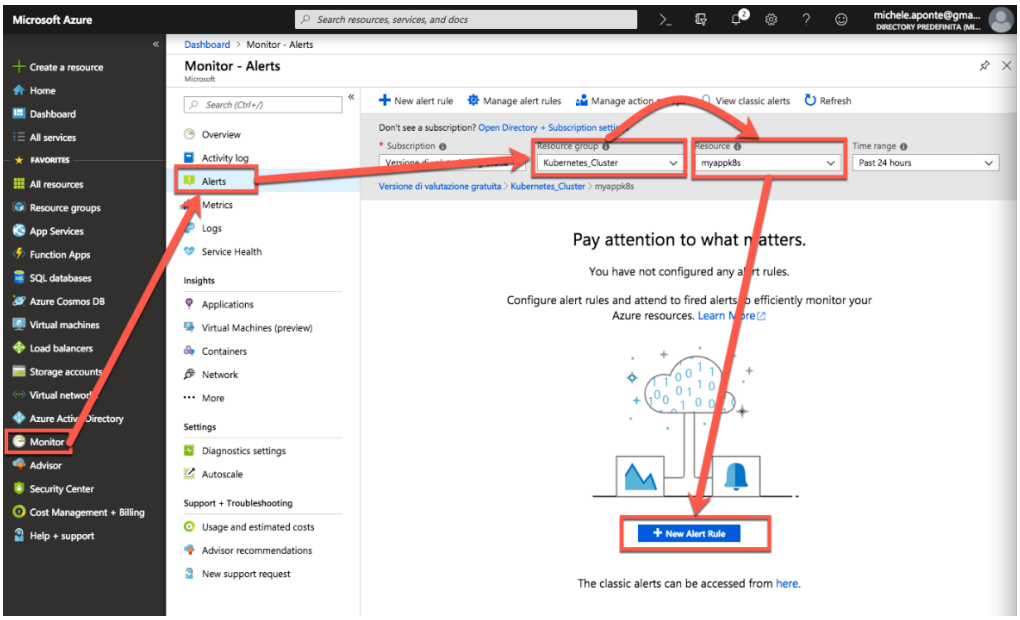

Figure 69: Monitoring alert from Azure portal

You can choose the resource to monitor, the condition that generates the alert, the action group to act when the condition is true (like send an email or text message), a name, and a description of the alert. You can choose the conditions from many options, and aggregate information by configurable period and frequency.

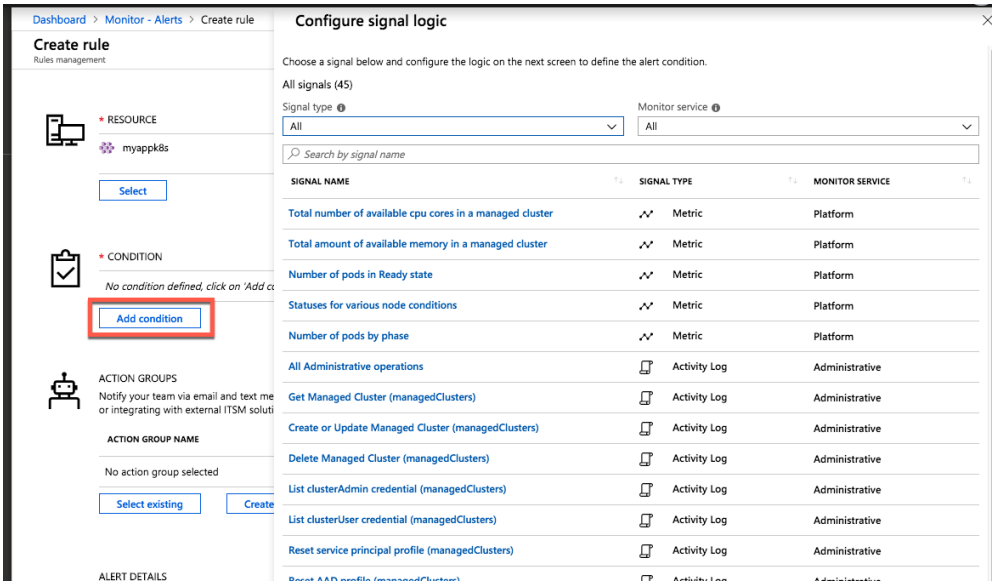

Figure 70: Azure monitoring Alert wizard

If you are not on the Azure platform, there are a lot of advanced tools available (both free and paid) that can help you to have the same control of the cluster monitoring. The most well-known is **Prometheus**, which you can use with AKS if you have hybrid monitoring requirements. It is open source, and you can find rich documentation about its configuration.

What next?

Kubernetes is the de facto standard for container orchestration. You can use it everywhere, and it is receiving a lot of improvements thanks to the contribution of the main cloud providers that supply it as a service.

In this book, we have seen Docker containers and Kubernetes from the developer point of view, but to manage a production environment, there are a lot of tools and objects to explore from a system administrator point of view.

I hope that this book has helped you begin your journey with Docker and Kubernetes, because these tools are the present and the future of software development. I suggest that you study more in depth and put what you learn into practice—it's the only way to work with these tools and have fun while doing it!

Happy coding!