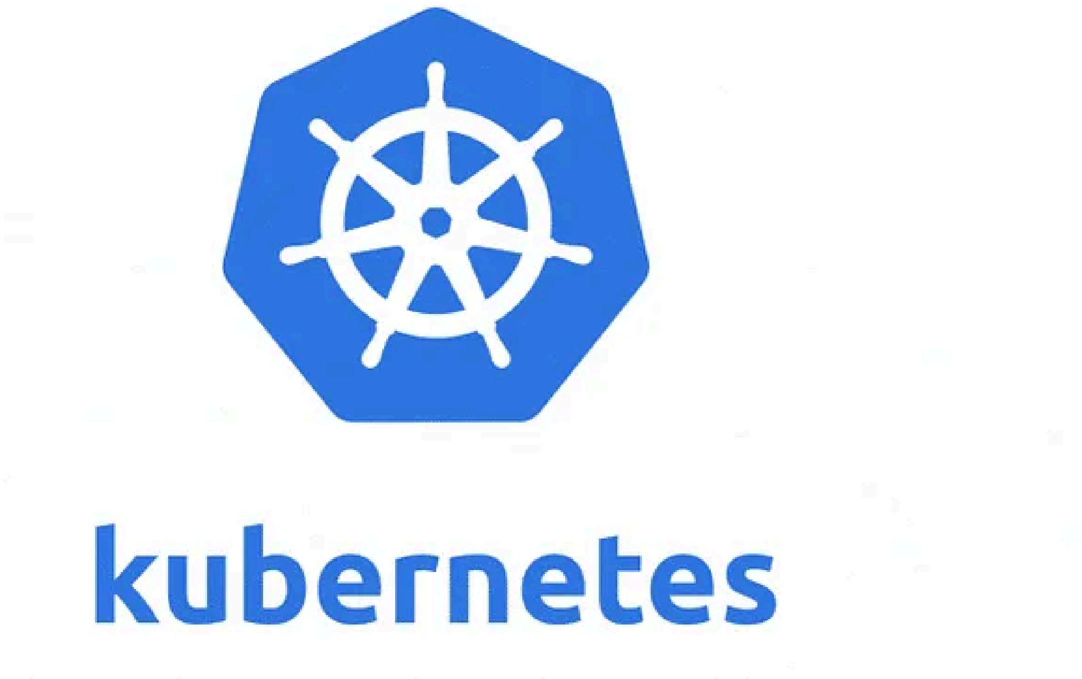
Building an API with .NET Core, Docker and Kubernetes

José Sousa · Follow
8 min read · Jul 13, 2023

Listen Share More



Introduction

In today's modern software development landscape, building scalable and containerized applications is crucial. In this blog post, we'll explore how to build an API using the popular .NET Core framework, containerize it with Docker, and deploy it to a Kubernetes cluster. This combination offers a powerful and flexible environment for developing, packaging, and orchestrating your API.

Prerequisites

Before we dive into the steps, ensure that you have the following tools set up:

- .NET Core SDK
- Docker
- Kubernetes cluster (local or cloud-based)
- Kubernetes CLI (kubectl)

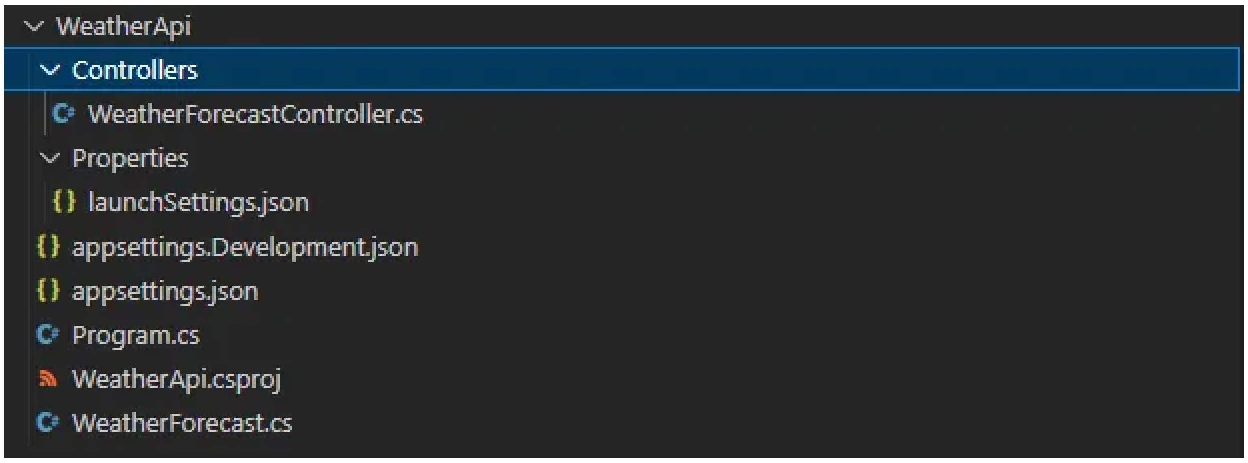
Step 1: Create the API using .NET Core

To get started, follow these steps:

1. Open your preferred integrated development environment (IDE) or a command-line interface (CLI).
2. Create a new .NET Core Web API project by running the following command:

```
dotnet new webapi -n WeatherApi
```

You should end up with a file structure simillar to this:



```
WeatherApi
  Controllers
    WeatherForecastController.cs
  Properties
    launchSettings.json
  appsettings.Development.json
  appsettings.json
  Program.cs
  WeatherApi.csproj
  WeatherForecast.cs
```

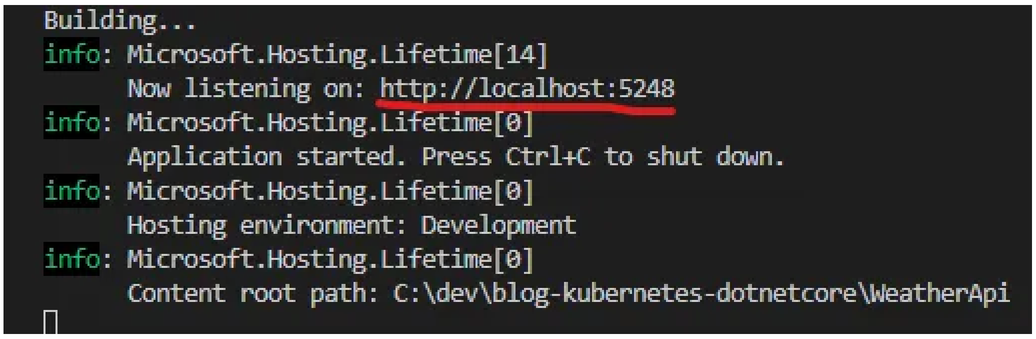
- The `Controllers` directory contains the controller files responsible for handling API requests and defining endpoints.
- The `WeatherApi.csproj` file is the project file that manages the project's dependencies and configuration.
- The `Program.cs` file is the entry point for the application.

- The `Startup.cs` file configures the application and services.

We can now build and run our web api project:

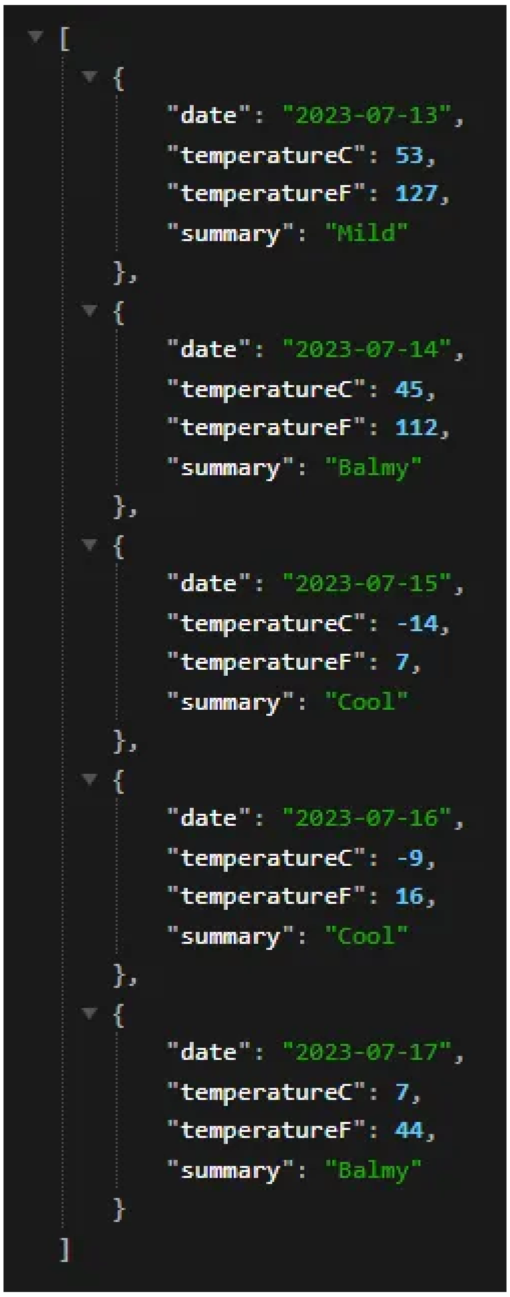
```
dotnet build
dotnet run
```

You should have an output simillar to this:



```
Building...
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5248
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\dev\blog-kubernetes-dotnetcore\WeatherApi
```

Access it using the URL listed above http://localhost:5248/weatherforecast



```
[
  {
    "date": "2023-07-13",
    "temperatureC": 53,
    "temperatureF": 127,
    "summary": "Mild"
  },
  {
    "date": "2023-07-14",
    "temperatureC": 45,
    "temperatureF": 112,
    "summary": "Balmy"
  },
  {
    "date": "2023-07-15",
    "temperatureC": -14,
    "temperatureF": 7,
    "summary": "Cool"
  },
  {
    "date": "2023-07-16",
    "temperatureC": -9,
    "temperatureF": 16,
    "summary": "Cool"
  },
  {
    "date": "2023-07-17",
    "temperatureC": 7,
    "temperatureF": 44,
    "summary": "Balmy"
  }
]
```

It is as simple as this! You have successfully created a basic API using .NET Core.

Step 2: Containerize the API with Docker

To containerize your API with Docker, follow these steps:

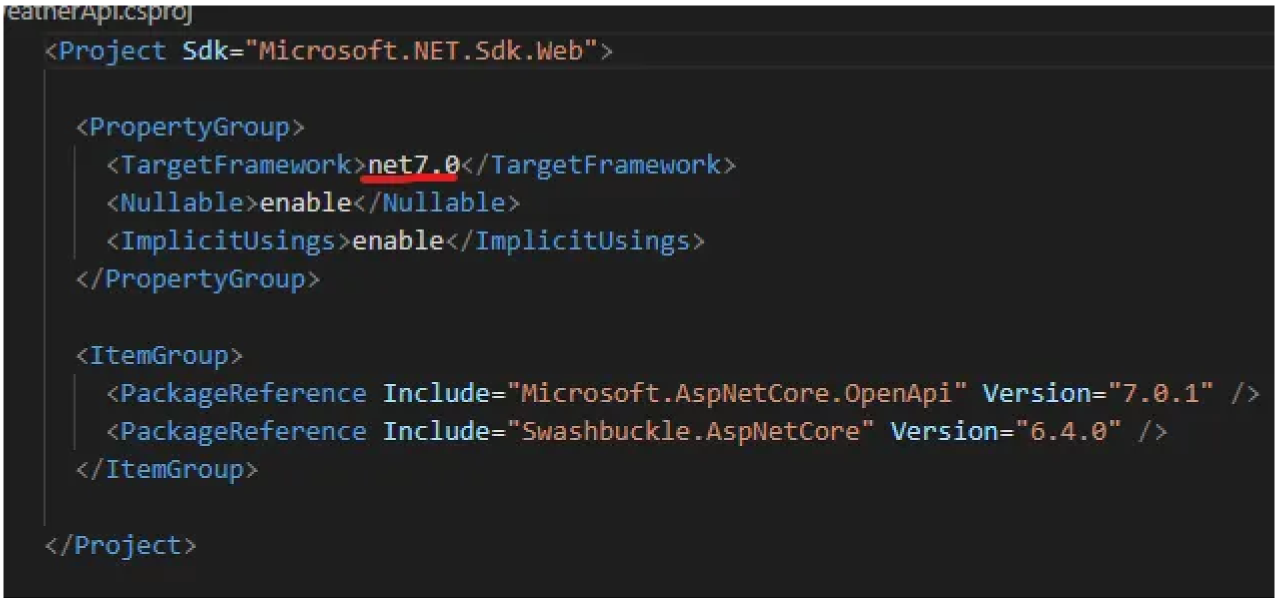
1. Create a file named `Dockerfile` in the root directory of your API project.

2. Open the `Dockerfile` and add the following code:

```
# Set the base image
FROM mcr.microsoft.com/dotnet/sdk:7.0 AS build-env

# Set the working directory
WORKDIR /app
```

```
# Copy csproj and restore as distinct layers
COPY *.csproj ./
RUN dotnet restore

# Copy everything else and build the API
COPY . ./
RUN dotnet publish -c Release -o out

# Build the runtime image
FROM mcr.microsoft.com/dotnet/aspnet:7.0
WORKDIR /app
COPY --from=build-env /app/out .

# Expose the API port
EXPOSE 80

# Set the entry point for the API
ENTRYPOINT ["dotnet", "WeatherApi.dll"]
```

***Note:** Ensure that the .NET SDK used in the **Dockerfile** matches the SDK version of you project. In my case I'm using .NET 7, double check the .csproj file.*



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

  <PropertyGroup>
    <TargetFramework>net7.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.OpenApi" Version="7.0.1" />
    <PackageReference Include="Swashbuckle.AspNetCore" Version="6.4.0" />
  </ItemGroup>

</Project>
```

Let's go through the Dockerfile step-by-step:

- The `FROM` statement sets the base image as the .NET SDK image, which allows us to build and publish the API.
- The `WORKDIR` instruction sets the working directory within the container to `/app`.
- The `COPY` command copies the `.csproj` file into the container's `/app` directory.

- The `RUN dotnet restore` command restores the NuGet packages for the project.
- The second `COPY` command copies the rest of the project files into the container's `/app` directory.
- The `RUN dotnet publish` command builds the API in the `Release` configuration and outputs the published files into the `out` folder.
- The second `FROM` statement sets the base image as the .NET runtime image, which is smaller and more lightweight than the SDK image.
- The `WORKDIR` instruction sets the working directory to `/app` again.
- The `COPY --from=build-env` command copies the published files from the `build-env` stage (the previous stage) into the container's `/app` directory.
- The `EXPOSE` instruction exposes port 80, which is the default port for the API.
- Finally, the `ENTRYPOINT` command specifies the command to run when the container starts. In this case, it runs the `MyApi.dll` file with the `dotnet` command.

1. Save the `Dockerfile`.
2. Open a command-line interface (CLI) or terminal in the same directory as the `Dockerfile`.
3. Build the Docker image using the following command:

```
docker build -t weatherapi-image .
```

This command builds the Docker image based on the `Dockerfile` in the current directory and tags it as `weatherapi-image`.

Once the image is built, you can run a container based on that image using the following command:

```
docker run -d -p 8080:80 --name weatherapi-container weatherapi-image
```

This command starts a container named `weatherapi-container` based on the `weatherapi-image`, mapping port 8080 on the host to port 80 within the container. The API will be accessible at `http://localhost:8080/weatherforecast`.

That's it! You have successfully containerized your API using Docker. You can now deploy and run the API in any environment that supports Docker.

Feel free to modify the `Dockerfile` according to your specific requirements, such as adding environment variables or configuring additional settings.

Step 3: Deploy to Kubernetes

To deploy your API to Kubernetes, follow these steps:

Set up a Kubernetes cluster: Depending on your requirements, you can set up a local Kubernetes cluster using Minikube or a cloud-based cluster using services like Azure Kubernetes Service (AKS) or Amazon Elastic Kubernetes Service (EKS). I'll be using **Minikube** for this example, you can follow this doc: https://minikube.sigs.k8s.io/docs/start/
Please note that installing and starting your local minikube Kubernetes environment or the first time may take some time and a few machine restarts. I've also had several issues starting minikube in windows and using the following command fixed the problem:

```
minikube start --hyperv-use-external-switch --driver=docker --docker-env=local
```

This is a breakdown of the previous command:

- `minikube start`: This command starts Minikube and sets up a local Kubernetes cluster.
- `--hyperv-use-external-switch`: This flag is specific to the Hyper-V driver. It instructs Minikube to use an external virtual switch for networking instead of

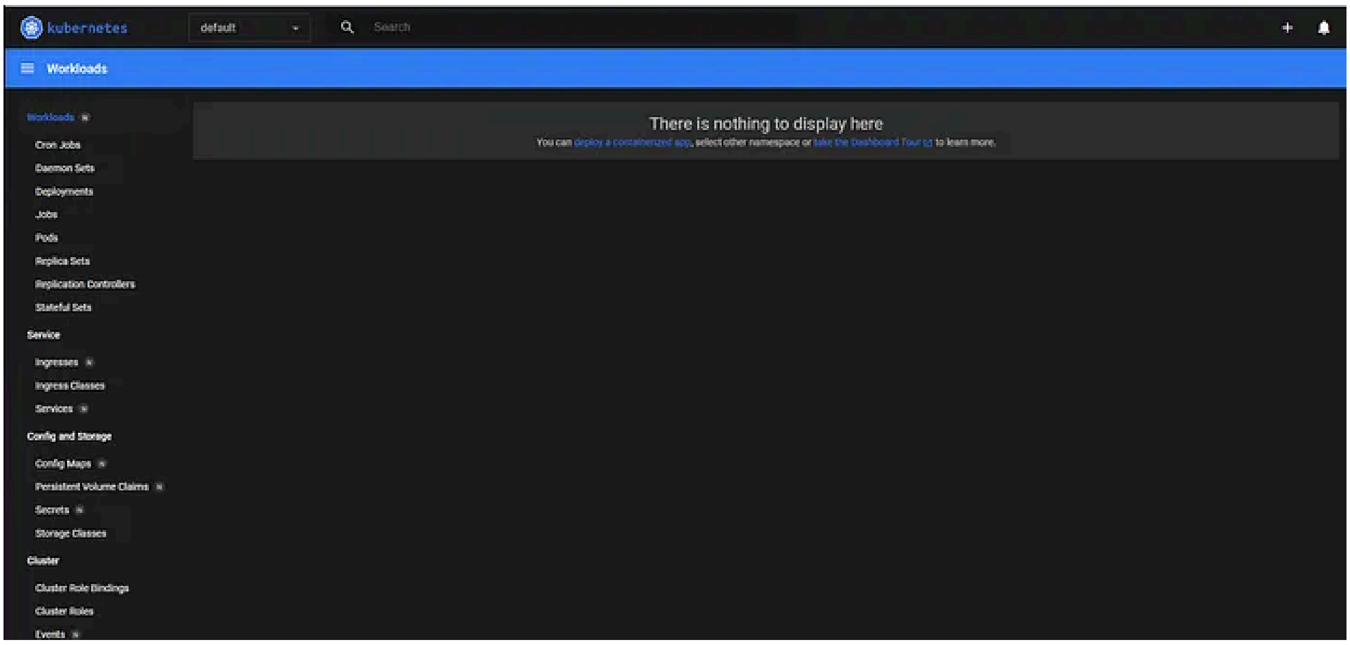
creating an internal virtual switch. This can be helpful if you want to connect your cluster to the network or access it from other machines.

- `--driver=docker` : This flag specifies the driver to be used by Minikube. In this case, it sets the driver to Docker, indicating that Minikube should use the Docker driver for managing the Kubernetes cluster.
- `--docker-env=local` : This flag is used to configure Minikube to use the local Docker environment. By setting it to `local` , Minikube will utilize the Docker daemon on your local machine rather than spinning up a separate virtual machine for Docker.

When you execute this command, Minikube will start and create a single-node Kubernetes cluster using the Docker driver and the local Docker environment.

Now we should be able to open the Kubernetes dashboard:

```
minikube dashboard
```



There is nothing to display here
You can deploy a containerized app, select other namespace or take the Dashboard Tour to learn more.

Lets now deploy our weather API to Kubernetes :)

Create a deployment YAML file: In your project directory, create a new file named `deployment.yaml` (or any other desired name) to define the deployment configuration. Open the file in a text editor and add the following content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: weatherapi-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: weatherapi
  template:
    metadata:
      labels:
        app: weatherapi
    spec:
      containers:
      - name: weatherapi-container
        image: weatherapi-image
        imagePullPolicy: Never
        ports:
        - containerPort: 80
```

- The `metadata` section specifies the name of the deployment.
- The `spec.replicas` field indicates the desired number of replicas (instances) of the API to be running.
- The `spec.selector` field specifies the label selector for the deployment.
- The `spec.template` section defines the configuration for the pods created by the deployment.
- The `spec.template.metadata.labels` field sets the label for the pods.
- The `spec.template.spec.containers` section specifies the container configuration.
- The `name` field sets the name of the container.
- The `image` field references the Docker image you built earlier (`weatherapi-image`).

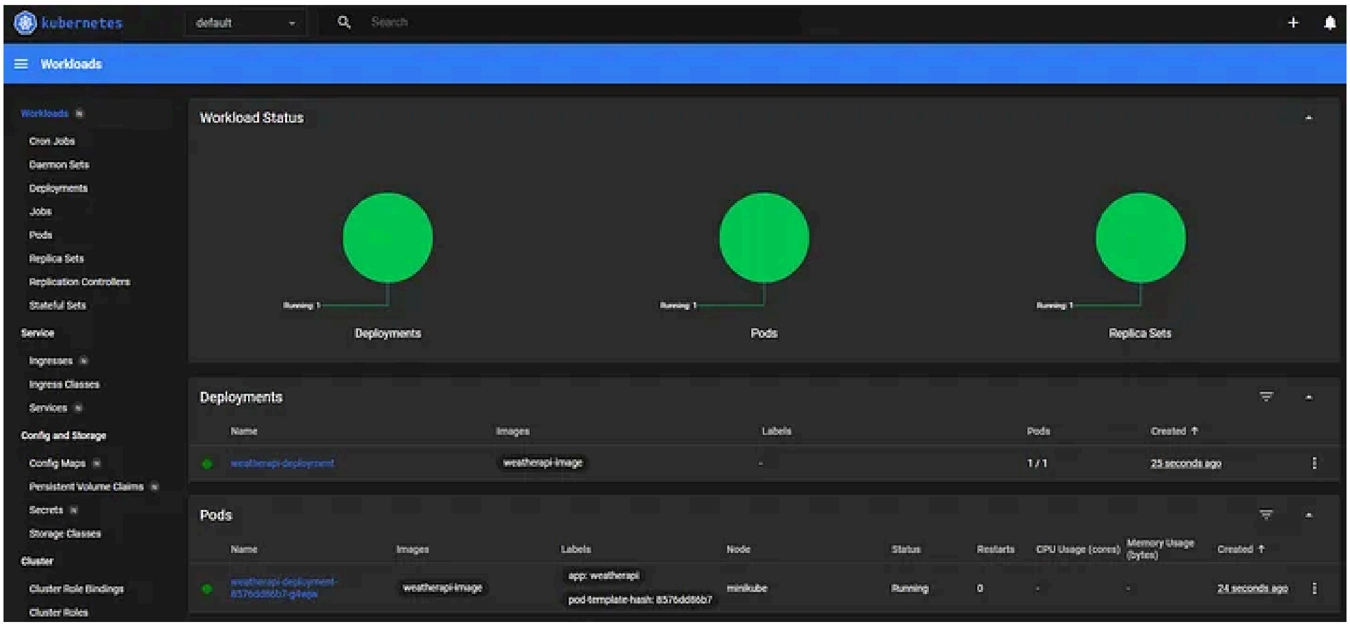
- The `imagePullPolicy: Never` ensures that Kubernetes does not attempt to pull the image from a remote registry.
- The `ports` section specifies the container port to expose (in this case, port 80).

Apply the deployment to Kubernetes: In your command-line interface, run the following command to apply the deployment YAML file:

```
minikube kubectl -- apply -f deployment.yaml
```

This command deploys the weather API to the Kubernetes cluster based on the configuration specified in the `deployment.yaml` file.

After checking your dashboard you should now be have to see the weather api successfully deployed!



Deployments

Name	Images	Labels	Pods	Created ↑
weatherapi-deployment	weatherapi-image	-	1 / 1	25 seconds ago

Pods

Name	Images	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Created ↑
weatherapi-deployment-8576dd86b7-g4wgw	weatherapi-image	app: weatherapi pod-template-hash: 8576dd86b7	minikube	Running	0	-	-	24 seconds ago

```
minikube kubectl -- get deployments

NAME                    READY   UP-TO-DATE   AVAILABLE   AGE
weatherapi-deployment   1/1     1            1           115s
```

But wait… by default, the API is only accessible within the Kubernetes cluster. To expose it externally, you must create a service. Create a new file named `service.yaml` (or any desired name) in your project directory and add the following content:

```
apiVersion: v1
kind: Service
metadata:
  name: weatherapi-service
spec:
  selector:
    app: weatherapi
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 30080    # Specify a static port for NodePort
  type: NodePort        # Change the service type to NodePort
```

- The `metadata` section sets the name of the service.
- The `spec.selector` field matches the label specified in the deployment YAML.
- The `ports` section specifies the port configuration for the service.
- `nodePort`: You can expose your service on a static port on each worker node in the cluster.

Apply the service to Kubernetes: Run the following command to apply the service YAML file:

```
minikube kubectl -- apply -f service.yaml

service/weatherapi-service created
```

To access the API externally, you can retrieve the external IP address of the service using the following command:

```
 minikube kubectl -- get services

NAME                  TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        A
kubernetes            ClusterIP   10.96.0.1        <none>        443/TCP        2
weatherapi-service    NodePort    10.103.175.76    <none>        80:30080/TCP   1
```

Look for the port number in the `PORT(S)` column associated with your `weatherapi-service`. The NodePort will be in the range of 30000-32767 (or as specified in the `nodePort` field). Access your API using the IP address of any worker node in your cluster and the NodePort. For example: `http://<worker-node-ip>:<node-port>/weatherforecast`.

The `worker-node-ip` can be obtained using the command:

```
minikube ip
192.168.49.2
```

Or you can simply run the command:

```
minikube service weatherapi-service

|-----------|---------------------|-------------|---------------------------|
| NAMESPACE |        NAME         | TARGET PORT |            URL            |
|-----------|---------------------|-------------|---------------------------|
| default   | weatherapi-service  |          80 | http://192.168.49.2:30080 |
|-----------|---------------------|-------------|---------------------------|
* Starting tunnel for service weatherapi-service.
|-----------|---------------------|-------------|------------------------|
| NAMESPACE |        NAME         | TARGET PORT |          URL           |
|-----------|---------------------|-------------|------------------------|
| default   | weatherapi-service  |             | http://127.0.0.1:56790 |
|-----------|---------------------|-------------|------------------------|
* Opening service default/weatherapi-service in default browser...
! Because you are using a Docker driver on windows, the terminal needs to be op
```

And you should be able to access your Pod at http://127.0.0.1:56790/weatherforecast

That's it! You have successfully deployed your API to Kubernetes.

Let me know if you have any further questions!

Kubernetes Microservices Software Development Software Engineering API

Follow

Written by José Sousa

124 Followers

Senior Software Engineer @Farfetch

More from José Sousa

José Sousa