In this exercise, we will explore how to deploy Web API in Kubernetes. First of all, create a .NET9 Web API application with the below code:

**Program.cs (Microservice B)**

using Microsoft.OpenApi.Models;

using System.Diagnostics;

using System.Text.Json.Serialization;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers().AddJsonOptions(options =>

{

options.JsonSerializerOptions.ReferenceHandler = ReferenceHandler.Preserve;

});

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen(c =>

{

c.SwaggerDoc("v1", new OpenApiInfo { Title = "Microservice B", Version = "v1" });

});

var app = builder.Build();

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json", "Microservice B"));

app.Lifetime.ApplicationStarted.Register(() =>

{

var url = "https://localhost:7074/Swagger";

try

{

Process.Start(new ProcessStartInfo

{

FileName = url,

UseShellExecute = true

});

}

catch (Exception ex)

{

Console.WriteLine($"Could not open browser: {ex.Message}");

}

});

}

app.UseHttpsRedirection();

app.MapControllers();

app.Run();

**JustBController.cs**

using Microsoft.AspNetCore.Mvc;

namespace MicroserviceB.Controllers;

[Route("api/[controller]")]

[ApiController]

public class JustBController : ControllerBase

{

[HttpPost("process")]

public IActionResult ProcessData([FromBody] string message)

{

Console.WriteLine($"[B] Received: {message}");

return Ok($"[B] Processed: {message}");

}

}

**DockerFile**

# See https://aka.ms/customizecontainer to learn how to customize your debug container and how Visual Studio uses this Dockerfile to build your images for faster debugging.

# This stage is used when running from VS in fast mode (Default for Debug configuration)

FROM mcr.microsoft.com/dotnet/aspnet:9.0 AS base

USER $APP\_UID

WORKDIR /app

EXPOSE 80

# This stage is used to build the service project

FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build

ARG BUILD\_CONFIGURATION=Release

WORKDIR /src

COPY ["MicroserviceB.csproj", "./"]

RUN dotnet restore "./MicroserviceB.csproj"

COPY . .

WORKDIR "/src"

RUN dotnet build "./MicroserviceB.csproj" -c $BUILD\_CONFIGURATION -o /app/build

# This stage is used to publish the service project to be copied to the final stage

FROM build AS publish

ARG BUILD\_CONFIGURATION=Release

RUN dotnet publish "./MicroserviceB.csproj" -c $BUILD\_CONFIGURATION -o /app/publish /p:UseAppHost=false

# This stage is used in production or when running from VS in regular mode (Default when not using the Debug configuration)

FROM base AS final

WORKDIR /app

COPY --from=publish /app/publish .

ENTRYPOINT ["dotnet", "MicroserviceB.dll"]

Dockerfile includes four stages, with some nuances tailored for Visual Studio (VS) integration and optimization. Let’s break down each stage and its significance in detail:

1. **base Stage**

FROM mcr.microsoft.com/dotnet/aspnet:9.0 AS base

USER $APP\_UID

WORKDIR /app

EXPOSE 80

**Purpose**:

* **Runtime Environment Setup**: Prepares the base image for both **debugging** (fast mode in VS) and **production**.
* **Security**: Sets a non-root user via USER $APP\_UID to follow least-privilege principles.
* **Port Configuration**: Exposes port 80 for HTTP traffic.
* **Workdir**: Defines /app as the working directory for the app.

**Significance**:

* Serves as the foundation for the final stage.
* Optimized for VS debugging (e.g., hot reload support).
* Avoids redundancy by reusing this stage across configurations.

1. **build Stage**

FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build

ARG BUILD\_CONFIGURATION=Release

WORKDIR /src

COPY ["MicroserviceB.csproj", "./"]

RUN dotnet restore "./MicroserviceB.csproj"

COPY . .

RUN dotnet build "./MicroserviceB.csproj" -c $BUILD\_CONFIGURATION -o /app/build

**Purpose**:

* **Dependency Restoration**: Restores NuGet packages *before* copying all source code to leverage Docker layer caching.
* **Build Process**: Compiles the app using the specified BUILD\_CONFIGURATION (default: Release).

**Significance**:

* Separates dependency restoration (dotnet restore) from the build (dotnet build) to optimize caching.
* The ARG parameter allows flexibility (e.g., switching between Debug and Release modes).
* Uses the **SDK image** to access build tools.

1. **publish Stage**

FROM build AS publish

ARG BUILD\_CONFIGURATION=Release

RUN dotnet publish "./MicroserviceB.csproj" -c $BUILD\_CONFIGURATION -o /app/publish /p:UseAppHost=false

**Purpose**:

* **Publishing**: Generates a self-contained deployment bundle.
* **Optimization**: The flag /p:UseAppHost=false skips generating an OS-specific executable (reduces size, as containers use dotnet CLI).

**Significance**:

* Isolates the publish step from the build to minimize the final image size.
* Prepares the app for deployment by including only runtime assets.

1. **final Stage**

FROM base AS final

WORKDIR /app

COPY --from=publish /app/publish .

ENTRYPOINT ["dotnet", "MicroserviceB.dll"]

**Purpose**:

* **Production-Ready Image**: Uses the lightweight aspnet runtime image from the base stage.
* **Artifact Copy**: Pulls the published output from the publish stage.
* **Entrypoint**: Defines how to launch the app (executes the compiled DLL).

**Significance**:

* Final image excludes SDK and build tools, reducing size and attack surface.
* Inherits security settings (non-root user) and port configuration from base.

**Folder Structure:**

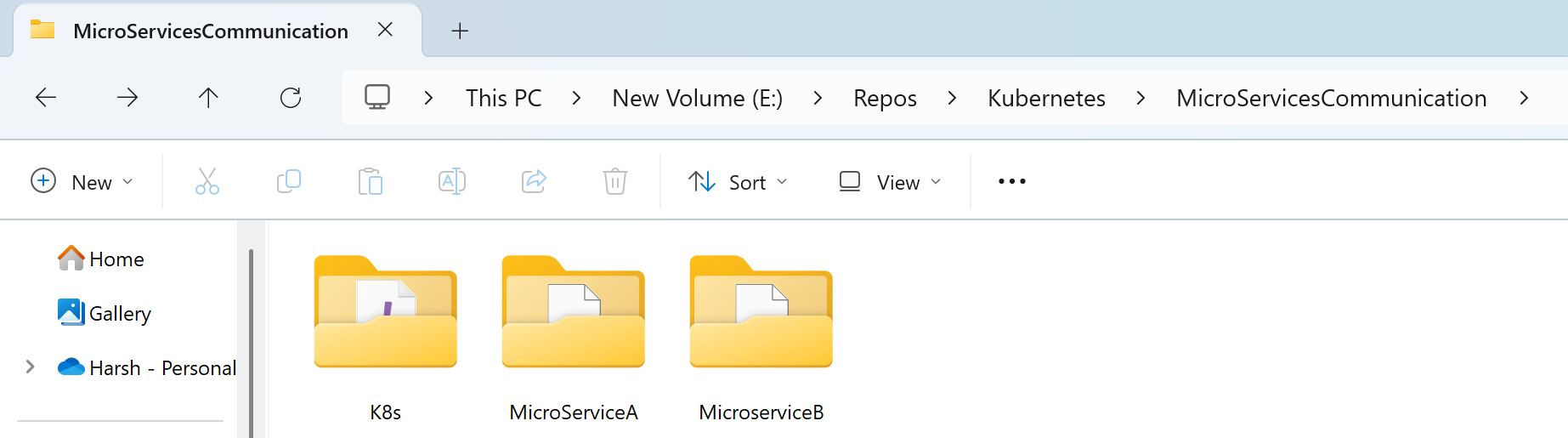
E:\Repos\Kubernetes\MicroServicesCommunication\MicroserviceB\MicroserviceB

The above folder contains the DockerFile and .csproj file which are required to build the docker image.

To run Kubernetes on local, you need to install the below tools:

1. Minikube
2. Kubectl
3. VirtualBox

Also, you need to create a deployment.yaml file that will deploy the services on your local. Here is the folder structure:



Command to build Docker Image:

docker build -t microservicea:latest .

Let's break down the command docker build -t microservicea:latest . and explain each component in detail:

1. **docker**

* **Role:** The Docker CLI (Command Line Interface) tool used to interact with the Docker daemon (background service).
* **Purpose:** Executes Docker commands like building images, running containers, etc.

1. **build**

* **Role:** The Docker subcommand to build a new image from a Dockerfile and a "build context" (files/directories used in the image).
* **Purpose:** Reads the Dockerfile in the current directory (or a specified path) and creates a layered image based on the instructions in the file.

1. **-t microservicea:latest (or --tag)**

* **Role:** The -t flag assigns a name and optionally a tag to the image.
* **Details:**
  + **microservicea:** The name of the image (e.g., a microservice named "microservicea").
  + **:latest:** The tag for the image. latest is a default tag used when no explicit tag is provided. It typically represents the most recent version of the image.
* **Purpose:** Tags the image for easy identification and versioning. Example: microservicea:1.0, microservicea:dev.

1. **. (the "build context")**

* **Role:** Specifies the build context, i.e., the directory where Docker will look for the Dockerfile and any files referenced in it (e.g., source code, configuration files).
* **Details:**
  + . refers to the current working directory.
  + Docker copies all files in this directory to the Docker daemon (unless excluded via .dockerignore).
  + The Dockerfile is expected to be in this directory by default. If it has a different name or path, use the -f flag (e.g., -f path/to/Dockerfile).

**Common Errors to Avoid:**

* Forgetting the . at the end (Docker won’t know where to find the build context).
* Using uppercase characters or special symbols in image names (Docker enforces lowercase names with underscores, hyphens, or numbers).

Definition of microserviceb-deployment.yaml file:

apiVersion: apps/v1

kind: Deployment

metadata:

  name: microserviceb

spec:

  replicas: 1

  selector:

    matchLabels:

      app: microserviceb

  template:

    metadata:

      labels:

        app: microserviceb

    spec:

      containers:

      - name: microserviceb

        image: microserviceb:latest

        env:

        - name: ASPNETCORE\_ENVIRONMENT

          value: Development

        - name: ASPNETCORE\_URLS

          value: "http://\*:82"

        imagePullPolicy: Never

        ports:

        - containerPort: 82

---

apiVersion: v1

kind: Service

metadata:

  name: microserviceb

spec:

  selector:

    app: microserviceb

  ports:

    - protocol: TCP

      port: 80

      targetPort: 82

What is the significance of --- in the above script? Do I need the whole script or something could be trimmed off?

The --- in a Kubernetes YAML file is a **document separator**. It allows you to define **multiple resources in the same file**. In your example, you're defining:

1. A Deployment named microserviceb
2. A Service named microserviceb

**Significance of ---**

* It separates two different Kubernetes resources within a single YAML file.
* It's not strictly required if you're applying just one resource, but it **is required** when you define **multiple resources** in the same file so that the Kubernetes parser knows where one ends and the next begins.

**Do you need the whole script?**

Yes, **both parts** are usually necessary for a functioning microservice:

* The Deployment defines the **pod and container(s)**.
* The Service exposes the pod so that other services or users can communicate with it (e.g., via port 80 in your case).

If you **only apply the Deployment**, there will be **no stable endpoint** (like a DNS name or load balancer) to access the microservice unless you do it manually (like using kubectl port-forward).

**You should keep both sections and the --- in between if:**

* You're applying both the Deployment and the Service in a single file.
* You want to ensure everything deploys and works together.

Can you describe each and every keyword of the above script in detail?

1. Deployment Configuration

apiVersion: apps/v1

* apiVersion: Specifies the Kubernetes API version you're using for this object.
* apps/v1 is used for managing Deployments, DaemonSets, and StatefulSets.

kind: Deployment

* kind: The type of Kubernetes object you're creating.
* Here, you're creating a Deployment, which manages replica sets and ensures the desired number of pod replicas are running.

metadata:

name: microserviceb

* metadata: Contains metadata information like the name, namespace, labels, etc.
* name: The unique name of the Deployment object in the cluster.

spec:

replicas: 1

* spec: Specifies the desired state of the object.
* replicas: Number of pod instances to run. Here, it is just 1 replica.

selector:

matchLabels:

app: microserviceb

* selector: Determines which pods belong to this Deployment by matching labels.
* matchLabels: Filters pods whose app label equals microserviceb.

template:

metadata:

labels:

app: microserviceb

* template: Describes the pod template to be used when creating pods.
* metadata.labels: Labels assigned to pods created by this Deployment.
* These labels must match the selector above for the Deployment to manage the pods.

spec:

containers:

- name: microserviceb

* spec (within template): Defines the container specs for the pod.
* containers: A list of containers (in this case, just one).
* name: A unique name for this container within the pod.

image: microserviceb:latest

* image: Docker image to use. microserviceb:latest means use the latest tag of the microserviceb image.

env:

- name: ASPNETCORE\_ENVIRONMENT

value: Development

- name: ASPNETCORE\_URLS

value: [http://\*:82](http://*:82)

* env: Specifies environment variables to set inside the container.
* ASPNETCORE\_ENVIRONMENT: Typically Development, Staging, or Production.
* ASPNETCORE\_URLS: Tells ASP.NET Core which URL/port to bind to. http://\*:82 binds to port 82 on all interfaces.

imagePullPolicy: Never

* imagePullPolicy: Controls when the image is pulled from a registry.
* Never means Kubernetes will not pull the image — it assumes it's already available locally on the node.
* Useful during local development/testing with local images.

ports:

- containerPort: 82

* ports: Specifies container ports that are exposed by the container.
* containerPort: Port inside the container that the app is listening on (port 82).

1. Service Configuration

---

* This is a YAML document separator. It lets you define multiple resources in the same file.

apiVersion: v1

* The API version for the Service resource.

kind: Service

* You're creating a Kubernetes Service.
* A Service allows network access to a set of pods.

metadata:

name: microserviceb

* Metadata for the service. The name will be used as the service DNS entry (e.g., microserviceb.default.svc.cluster.local).

spec:

selector:

app: microserviceb

* selector: Selects pods that match the given labels.
* Here, it matches pods with the label app: microserviceb (like the ones defined in the Deployment).

ports:

- protocol: TCP

port: 80

targetPort: 82

* ports: List of ports exposed by the service.
* protocol: Network protocol (typically TCP).
* port: Port on which the service is exposed internally.
* targetPort: Port on the pod that receives traffic from the service. It must match the container's listening port.

So:

* The service listens on port **80**, and routes that traffic to port **82** in the pod.

|  |  |
| --- | --- |
| Keyword | Description |
| apiVersion | API version to use for the resource |
| kind | Type of Kubernetes object |
| metadata | Identifiers and labels |
| spec | Specification of the desired state |
| replicas | Number of pods to run |
| selector | How to identify pods for management or exposure |
| template | Pod blueprint |
| containers | List of containers in a pod |
| env | Environment variables |
| imagePullPolicy | When the image should be pulled |
| containerPort | Port exposed by the container |
| --- | Separates multiple resources in the YAML file |
| Service | Creates a stable endpoint to access pods |

Definition of deployment.yaml file:

apiVersion: apps/v1

kind: Deployment

metadata:

  name: microservicea

spec:

  replicas: 1

  selector:

    matchLabels:

      app: microservicea

  template:

    metadata:

      labels:

        app: microservicea

    spec:

      containers:

      - name: microservicea

        image: microservicea:latest

        env:

        - name: ASPNETCORE\_URLS

          value: "http://\*:80"

        imagePullPolicy: Never

        ports:

        - containerPort: 80

---

apiVersion: v1

kind: Service

metadata:

  name: microservicea

spec:

  selector:

    app: microservicea

  ports:

    - protocol: TCP

      port: 80

      targetPort: 80

  type: NodePort