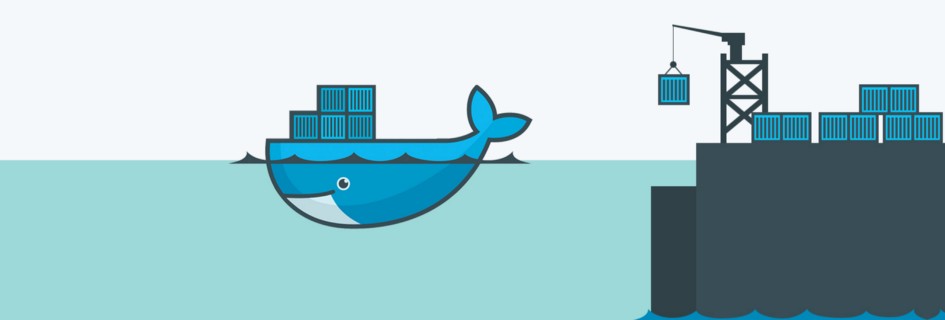
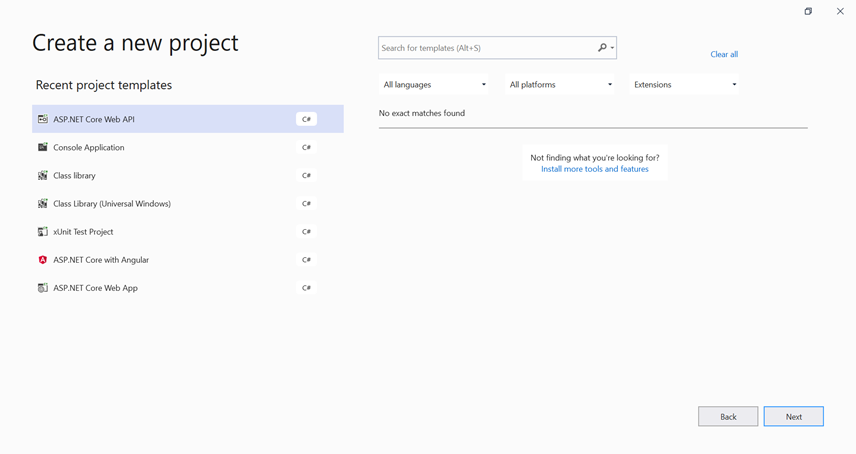
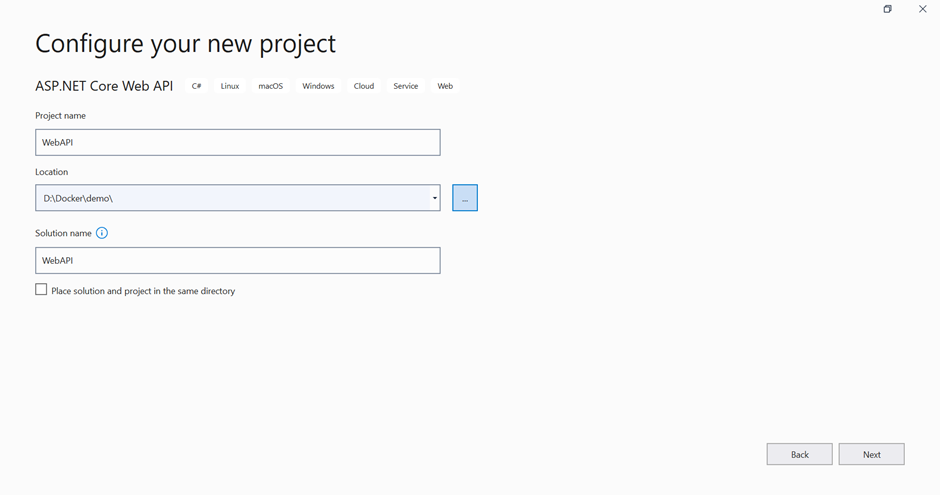
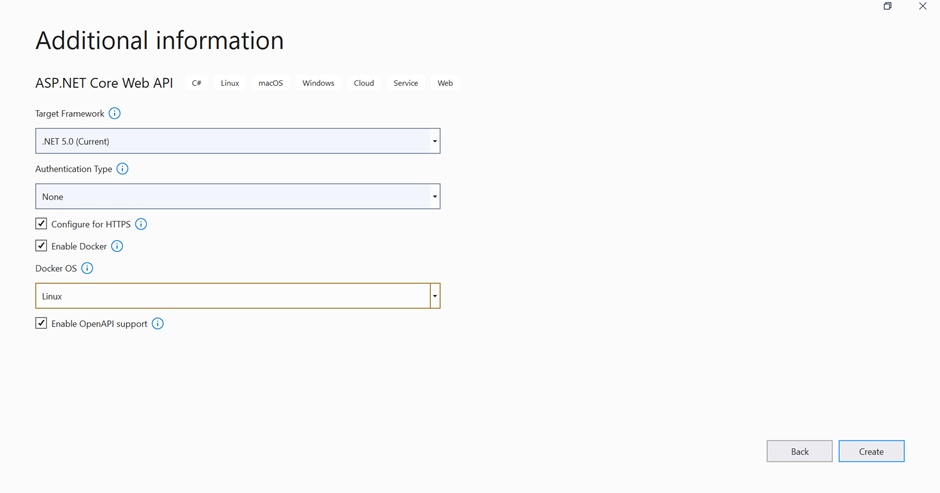
[https://www.c-sharpcorner.com/UploadFile/MinorCatImages/062632AM.png.ashx?width=64&height=64](https://www.c-sharpcorner.com/technologies/dotnetcore)

Docker .Net Core API and Angular End-To-End Application Using Docker Compose

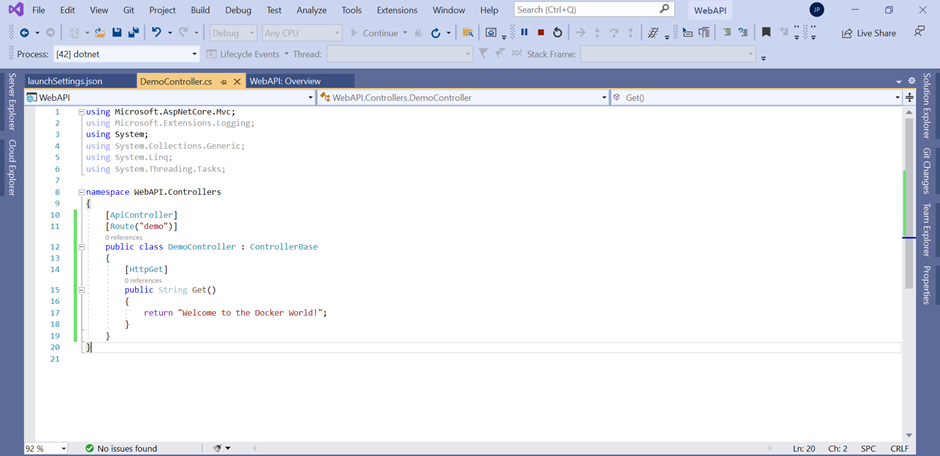


Build the .NET Core REST-API Backend

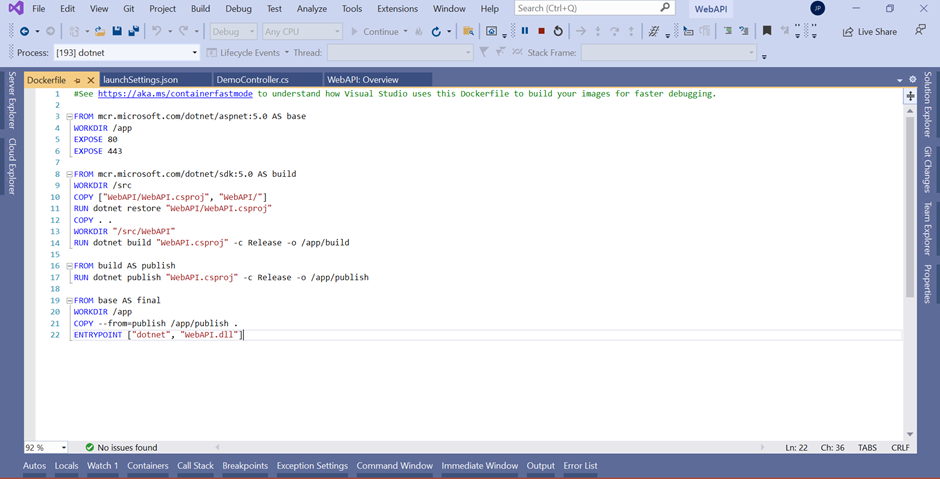
**Install**[Visual Studio Community](https://visualstudio.microsoft.com/en/vs/community/) (it’s free) with the ASP.NET and web development workload.

1. Create a new ASP.NET Core Web API Project  
   
2. Configure the new project  
   
3. Set Additional Information of Project like Target Framework, Authentication Type, Enable Https, Enable Docker, Docker OS Support Etc.  
   

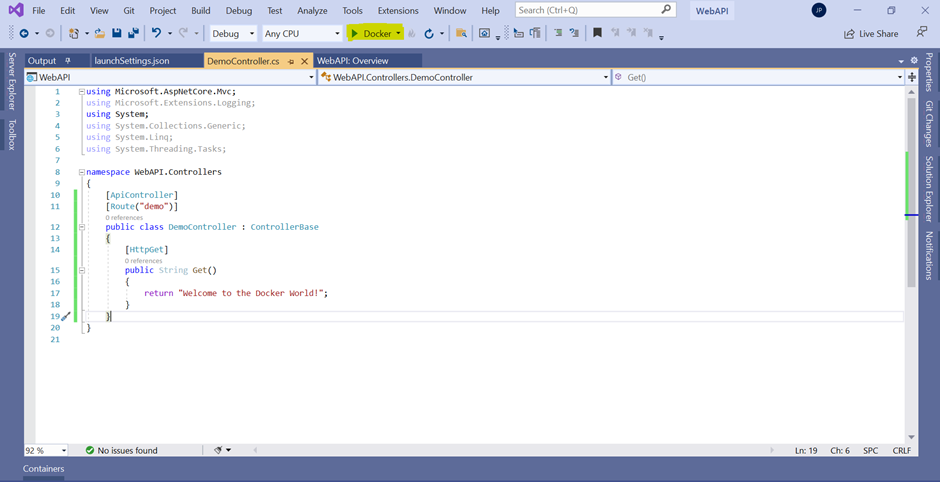
Add the new Controller “DemoController”. It will return Welcome to the Docker World! when we hit the get method.



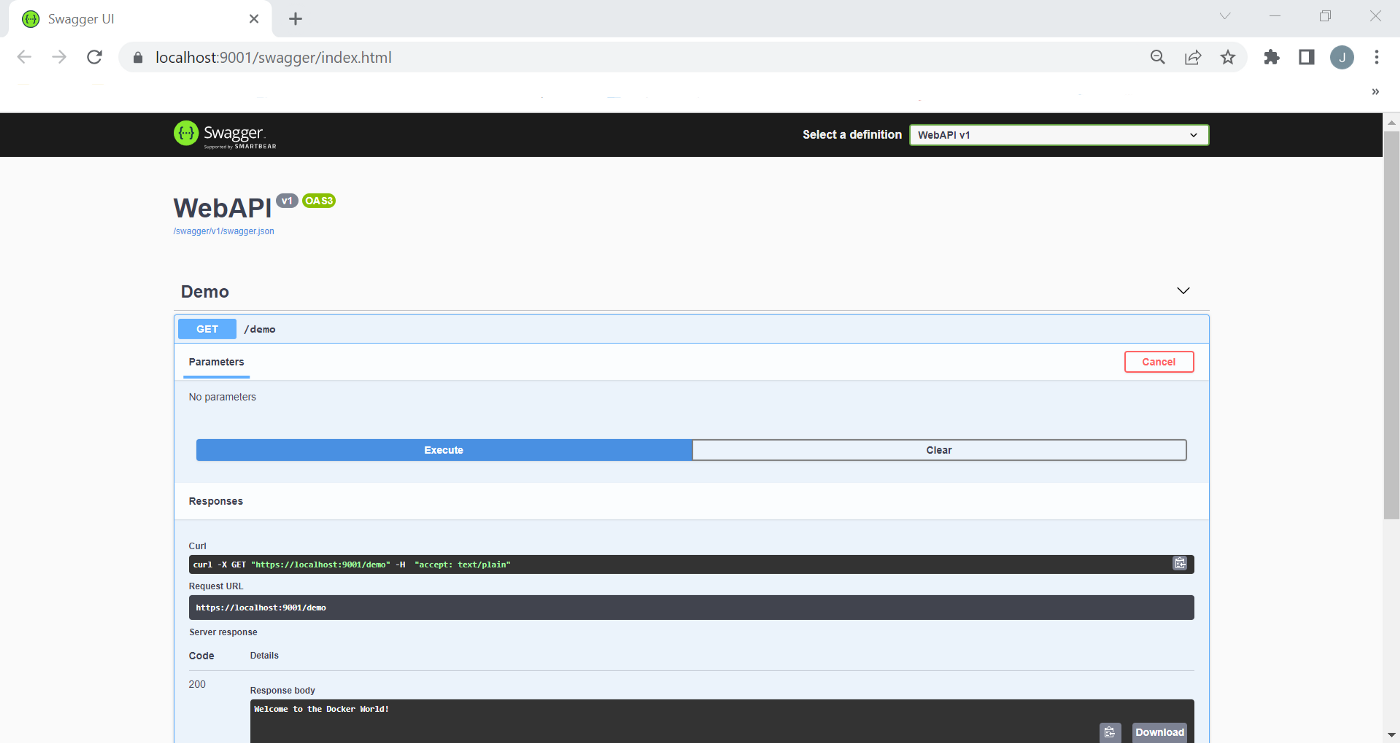
Then this is the Dockerfile that is created by Visual Studio when we enable the Docker Support option while creating the new Web Application. In that there are different steps like first we use a base image of aspnet then set the default docker app directory and container default port, later on, add .net sdk for build the application then copy all content from host to docker directory and set the entry point.



After that when we run the application by using Docker then It will create the Docker Image in Docker Desktop which you installed on your local machine. (Please confirm the Docker Desktop is running properly on your machine and it will be in running mode)



Here you will see the output after executing the get method of DemoController.



CORS Support

The full form of CORS is Cross-Origin Resource Sharing. It is a W3C standard that allows a server to make cross-domain calls from the specified domains while rejecting others By default due to browser security it prevents a web page from making one domain Ajax request to another domain.

But so many times we are using multiple domain applications which are required to call from one domain to another domain; in this case, we need to allow a cross-origin policy.

For that, we need to add CORS in the ConfigureServices method of the startup class.

services.AddCors(c =>

{

c.AddPolicy(“AllowOrigin”, options =>

{

options.AllowAnyOrigin()

.AllowAnyHeader()

.AllowAnyMethod();

});

});

C#

Copy

Later on, Also add middleware into the Configure method

app.UseCors(“AllowOrigin”);

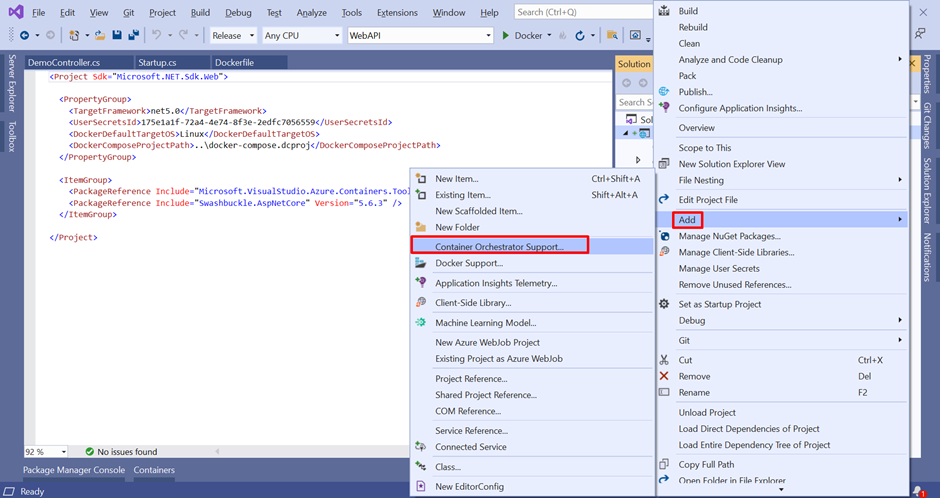
C#

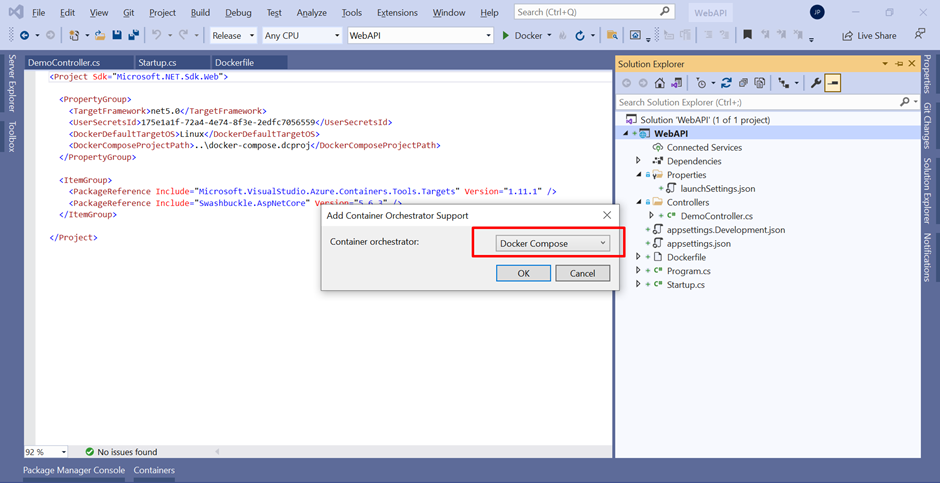
Copy

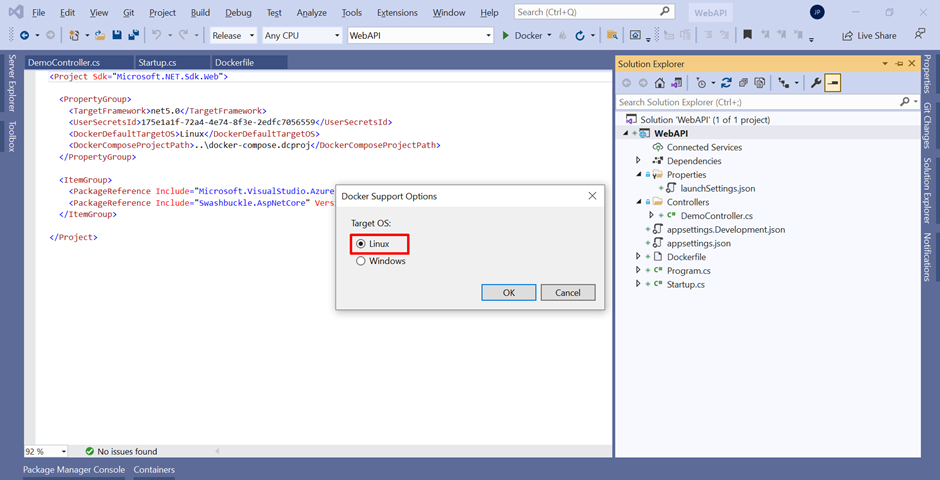
DOCKER-COMPOSE

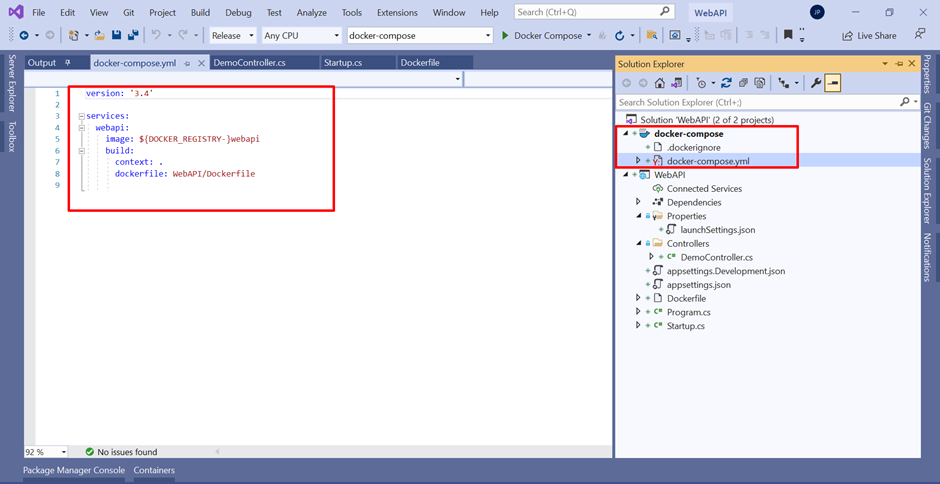
Compose is **a tool for defining and running multi-container Docker applications**. With Compose, you use a YAML file to configure your application’s services. Then, with a single command, you create and start all the services from your configuration.

Now Add Docker-Compose to set the dynamic port whatever we want for our Backend Application after adding Docker-Compose Container Orchestration Support in Visual Studio as mentioned in the image.

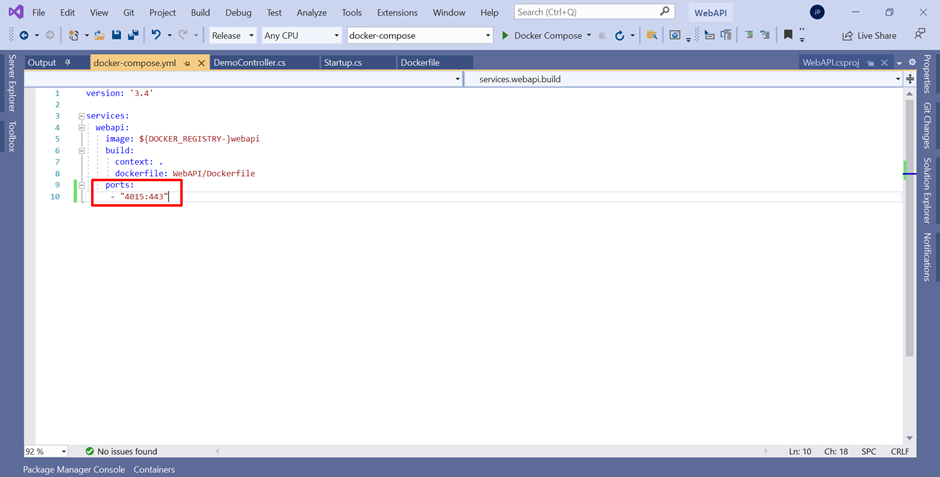




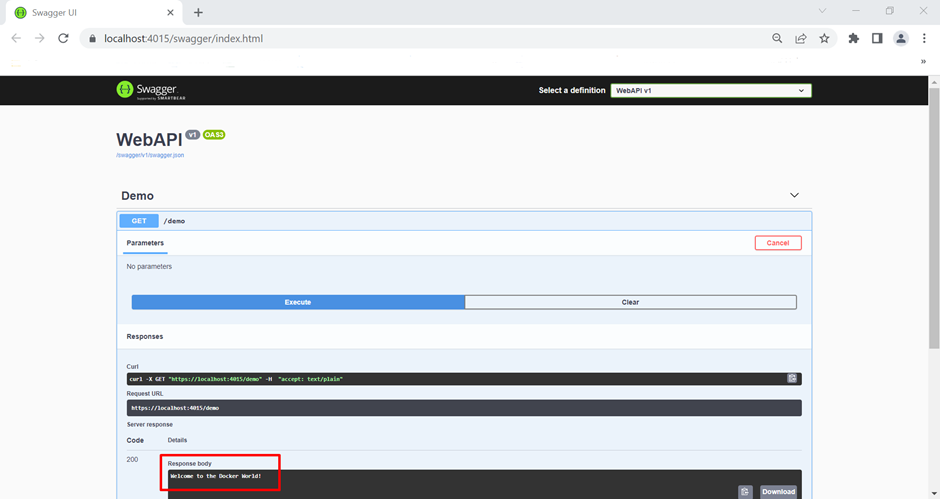




Now we are going to set the port for our backend application to run the application each time by using that port



And now when we run our application It will create the image in docker desktop and run the application on 4015 Port.



This is all about the backend web application. Now we are going to look into Frontend Angular Application.

Build Front-End Angular Application

Open the Visual Studio Code

1. Create the new Directory at a particular location

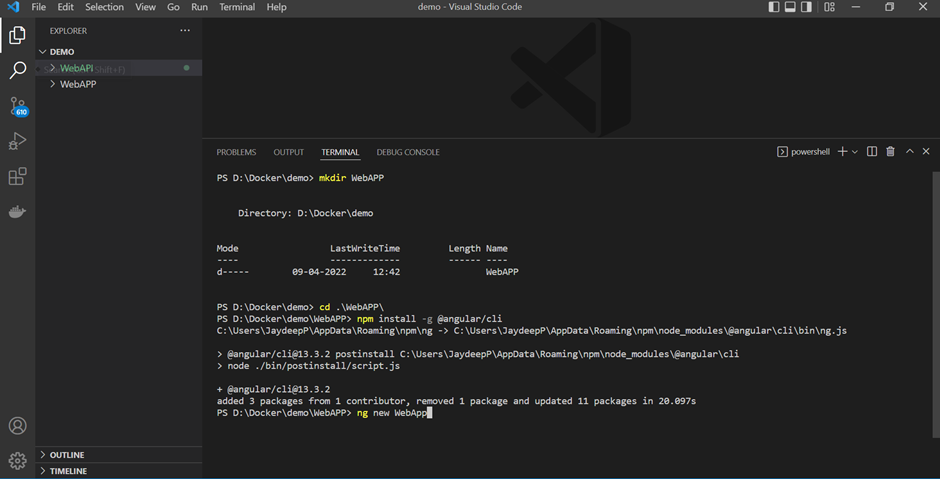
mkdir WebAPP

2. Go inside the WebAPP Directory

cd .\WebAPP\

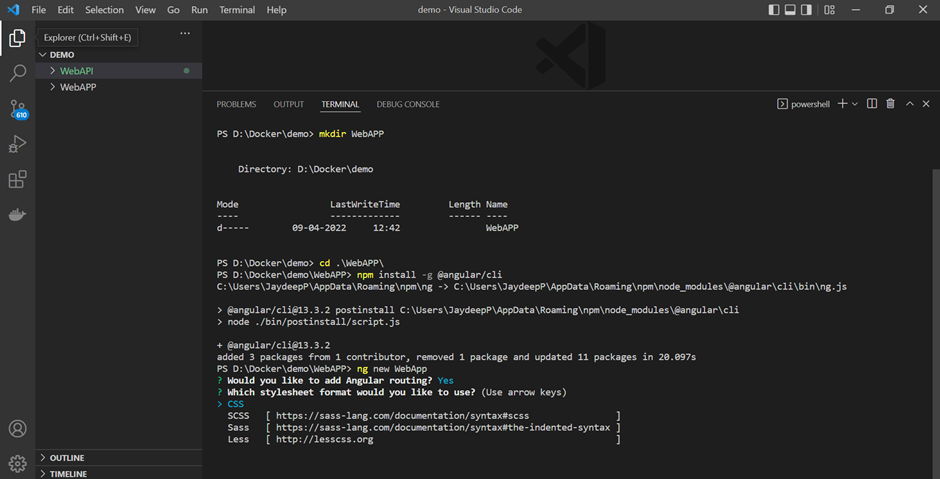
3. Install the Angular CLI inside that.

npm install -g @angular/cli



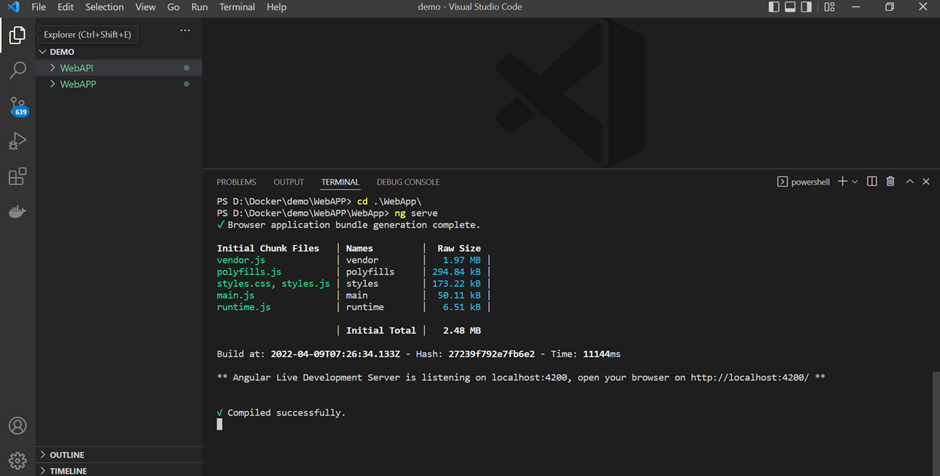
4. After Installing Angular CLI, Create the new Angular Project.

ng new WebApp

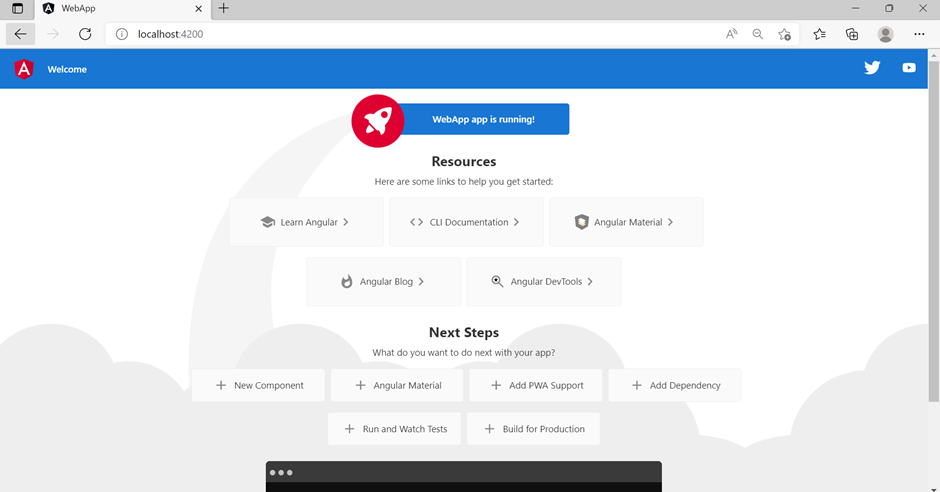


5 . Later on go inside your newly created app and run your Angular Application.

ng serve



Here you will see your application is in the running mode now



6 . Add HTTP Client Module in app.module.ts file which we are going to use in service to connect with our backend application to fetch the data.

import { NgModule } from ‘@angular/core’;

import { BrowserModule } from ‘@angular/platform-browser’;

import { AppRoutingModule } from ‘./app-routing.module’;

import { AppComponent } from ‘./app.component’;

import { HttpClientModule, HttpClient } from ‘@angular/common/http’;

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

AppRoutingModule,

HttpClientModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

JavaScript

Copy

7. Also add the Config.Json file under src/app/assets/config.json, which we use to configure our backend API URL and use throughout the application to fetch the data.

{

“apiServer”: {

“url”: “https://localhost:4015",

“version”: “v1”

}

}

JavaScript

Copy

8. Add the Demo Service using the following command for our WebApp Application

ng generate service demo

After executing the above command there will be one file created demo.service.ts , Inside that add the following code, which we are going to use to fetch the data from the backend application which is already in running mode.

import { Injectable } from ‘@angular/core’;

import configurl from ‘../assets/config/config.json’;

import { Observable, Subject } from ‘rxjs’;

import { HttpClient, HttpHeaders } from ‘@angular/common/http’;

@Injectable({

providedIn: ‘root’

})

export class DemoService {

config = {

ApiUrl: configurl.apiServer.url,

};

constructor(private http: HttpClient) {

this.getJSON().subscribe((data) => {

this.config.ApiUrl = data.apiServer.url;

});

}

public getJSON(): Observable<any> {

return this.http.get(‘./assets/config/config.json’);

}

getData(): Observable<any> {

return this.http.get(this.config.ApiUrl + ‘/demo’,

{responseType: ‘text’}

);

}

}

JavaScript

Copy

9. Now add the following code in app.component.ts file.

import { Component } from ‘@angular/core’;

import configurl from ‘../assets/config/config.json’

import { DemoService } from ‘./demo.service’;

@Component({

selector: ‘app-root’,

templateUrl: ‘./app.component.html’,

styleUrls: [‘./app.component.css’]

})

export class AppComponent {

config = {

ApiUrl: configurl.apiServer.url,

};

title = ‘WebApp’;

response = “No data loaded, yet”;

constructor(private sharedService: DemoService)

{

}

ngOnInit(): void {

setTimeout(() => {

this.refreshList();

}, 2000);

}

refreshList() {

this.sharedService.getData().subscribe(data =>{

console.log(data)

this.response = data;

});

}

}

JavaScript

Copy

10. Finally repace the app.component.ts code with the following HTML.

<[div](https://webplatform.github.io/docs/html/elements/div)>

<[h2](https://webplatform.github.io/docs/html/elements/h2)>Data loaded from the Web API:</[h2](https://webplatform.github.io/docs/html/elements/h2)>

{{ response }}

</[div](https://webplatform.github.io/docs/html/elements/div)>

Markup

Copy

Add the Docker Support to the Angular Application

1. First, we create nginx-custom.conf file which is related to the Nginx server and we use that to configure a few things as per our needs like reverse proxy and load balancing, etc.

server {

listen 80;

sendfile on;

default\_type application/octet-stream;

gzip on;

gzip\_http\_version 1.1;

gzip\_disable “MSIE [1–6]\.”;

gzip\_min\_length 256;

gzip\_vary on;

gzip\_proxied expired no-cache no-store private auth;

gzip\_types text/plain text/css application/json application/javascript application/x-javascript text/xml application/xml application/xml+rss text/javascript;

gzip\_comp\_level 9;

root /usr/share/nginx/html;

location / {

try\_files $uri $uri/ /index.html =404;

}

}

Bash

Copy

2. Then we create Dockerfile which we use to create the image of our web application after adding all dependencies and configuration that we will need when we run Image into the Container.

# Stage 1: Build an Angular Docker Image

FROM node as build

WORKDIR /apps

COPY . .

RUN npm install

COPY . /apps

ARG configuration=production

RUN npm run build — — outputPath=./dist/out — configuration $configuration

# Stage 2, use the compiled app, ready for production with Nginx

FROM nginx:alpine

COPY /nginx-custom.conf /etc/nginx/conf.d/

RUN rm -rf /usr/share/nginx/html/\*

COPY — from=build /apps/dist/out/ /usr/share/nginx/html

As I mentioned in Dockerfile, first we use node image and set the default docker working directory(app), then copy the content to the app directory and install all dependencies using npm and create production build using npm build which creates the dist folder. Secondly, we use the Nginx server image to run our web application and add nginx-custom.conf file inside that, and copy the content of dist.

3. Finally we are going to add a Docker-Compose file to build the image and run our application.

version: ‘3.5’

services:

saas-frontend:

image: ${DOCKER\_REGISTRY-}webapp:v1

build:

context: .

dockerfile: Dockerfile

ports:

- “4014:80”

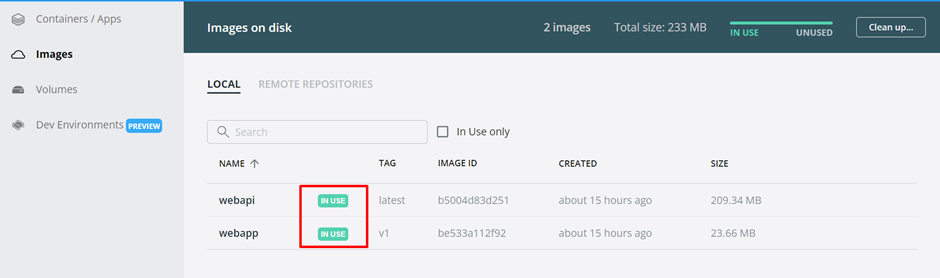
4. Now run the following command to build a docker image using docker-compose.

docker-compose build

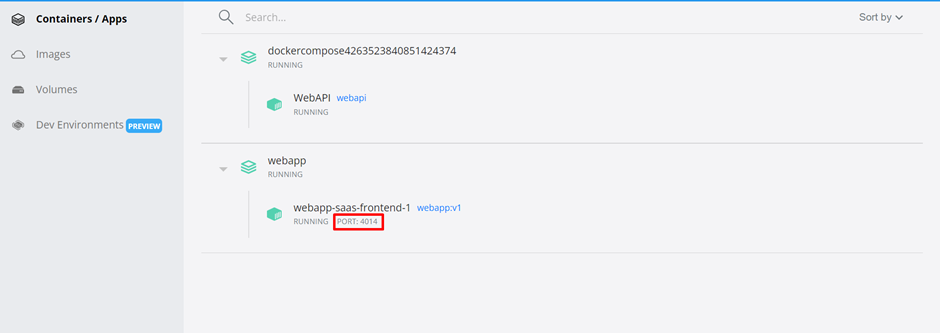
5. Finally run the following command to run the image using docker compose.

docker-compose up

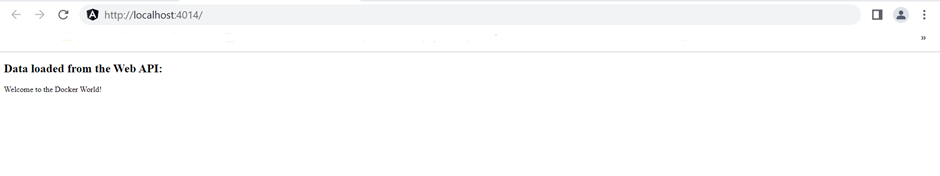
6. After running both applications, you can see the docker images in running mode on the docker desktop as I mentioned in the screenshot.



7. Also the Container you see the container is in the running mode and port to access that.



8. Finally when you hit <http://localhost:4014/> in the browser then you see the output of the application which we get from the backend API get method and patch that output into the Angular Application.



Hopefully, it has been helpful to you to understand the concepts.

Also if you want more like dynamic API\_URL Configuration using docker compose then please visit my following blog.

* [Dynamic API\_URL Configuration Of Angular Application using Docker Compose YML file](https://www.c-sharpcorner.com/blogs/dynamic-configuration-of-angular-api-url-using-docker-compose-yml-file)