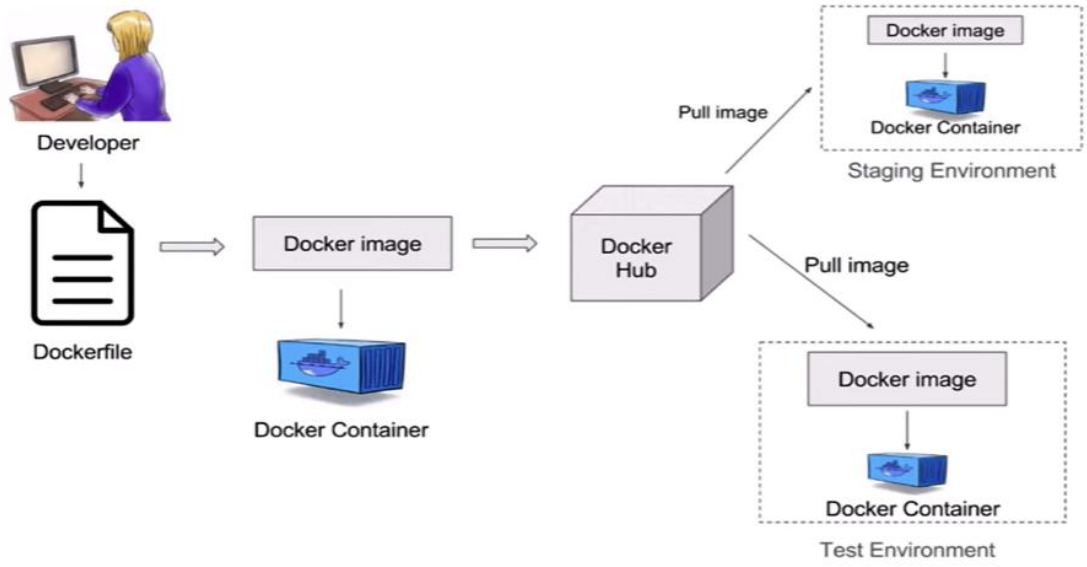
Dockerizing a Spring Boot Application using IntelliJ Environment

Dockerizing a Spring Boot Application - In this tutorial, you will learn how to use Docker to deploy the Spring Boot application in a container. This Spring Boot Docker tutorial is designed for beginners to learn step-by-step how to use Docker with the Spring Boot application.

Docker Workflow



The developer will create a **Dockerfile**, where all the instructions and commands to build the docker image. Dockerfile is a text file, it contains all instructions or commands to build the docker image. Once you create a Dockerfile, you can use the Docker build command to build the docker image. Docker image is an executable package and we can run the docker image in a container. Docker container is just a running instance of Docker image. The process of writing the Dockerfile and building the docker image is called **Dockerization**.

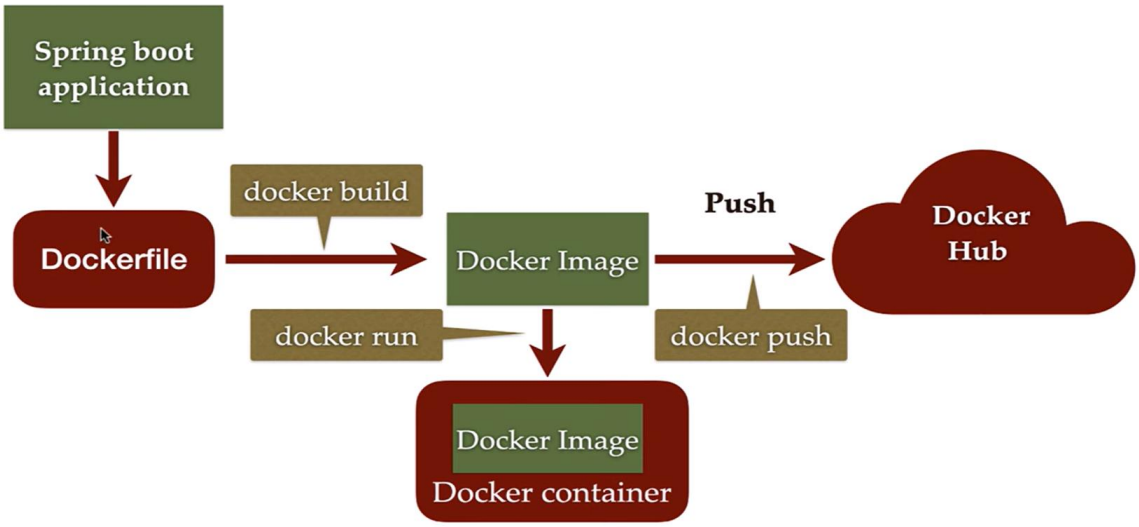
Once we have the Docker image in our local machine, we can push the image to an online Cloud repository called Docker hub. Once we push the image to the Docker hub, we can pull that image on any environment and we can start deploying or running this Docker image in a Docker container. For example, let us say we have different environments like staging, test, production or UAT environment. This is how a typical Docker Workflow looks like.

Architecture

Consider we want to Dockerize the Spring Boot application. We have to first create a Dockerfile where we will define all the instructions or commands to build the Docker Image. Once the Dockerfile is ready, we can use the Docker Build Command to build the Docker image from the Dockerfile. The Docker Image is just an executable package.

Next, we can use the Docker Run Command to run this Docker Image in a container. Docker container is just a running instance or runtime instance of the Docker Image. This complete process is called **Dockerization**.

Once we have a Docker Image on our local machine, then we can go ahead and push this Docker Image on a Docker Hub using Docker Push Command. Once we push the Docker Image to the Docker hub, and if we make this Docker Image as a public on Docker Hub. Then anybody can go ahead and pull this Docker Image from the Docker hub and they can run that Docker image in a Docker Image.



IMPLEMENTATION

Now, let us see how to Dockerize Spring Boot application step by step. We have to implement these steps now.

Dockerizing Spring Boot Application:

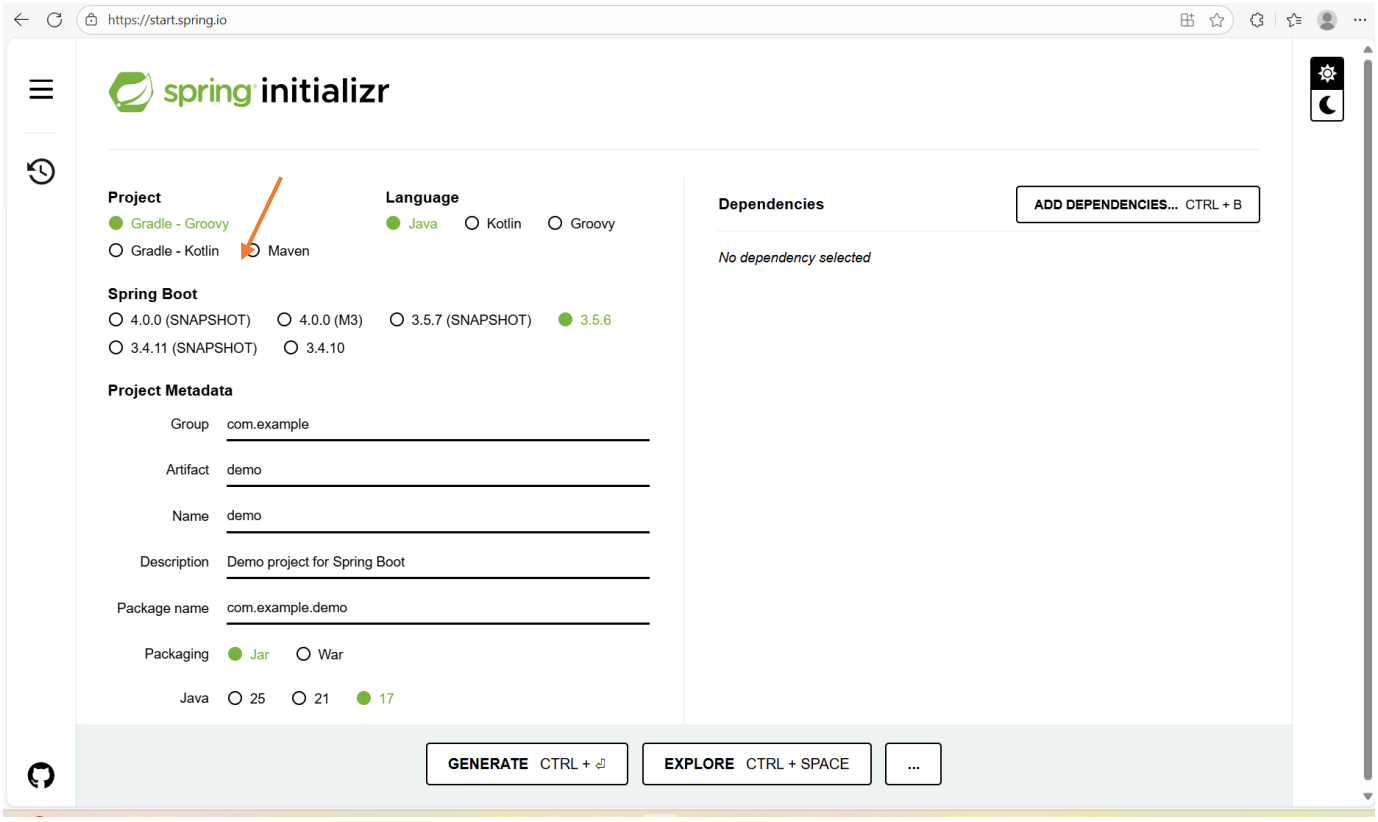
1. Create a Spring Boot Project
2. Build a Simple REST API
3. Build and Package Maven application
4. Create a Dockerfile to Build a Docker Image
5. Build Docker Image from Dockerfile
6. Run Docker Image in a Docker Container
7. Run Docker Image in a Docker Container in Detached Mode
8. Push Docker Image from Local Machine to DockerHub
9. Pull the Docker Image from a Docker Hub

Pull Docker Image from DockerHub

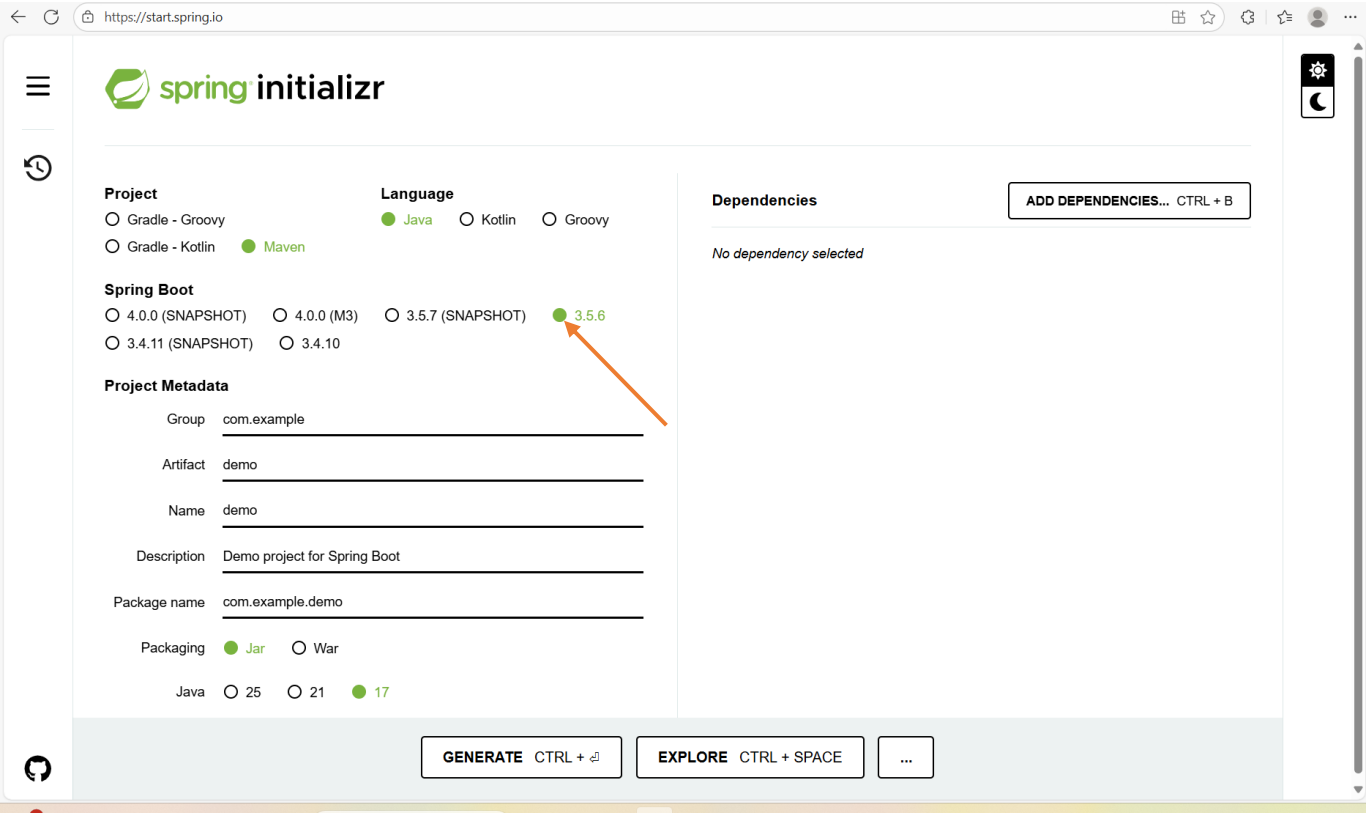
STEP 1: Create a Spring Boot Project

First, we will create a Spring Boot application and we will build some REST API. Let us create a Spring Boot application and we will import that Spring Boot application.

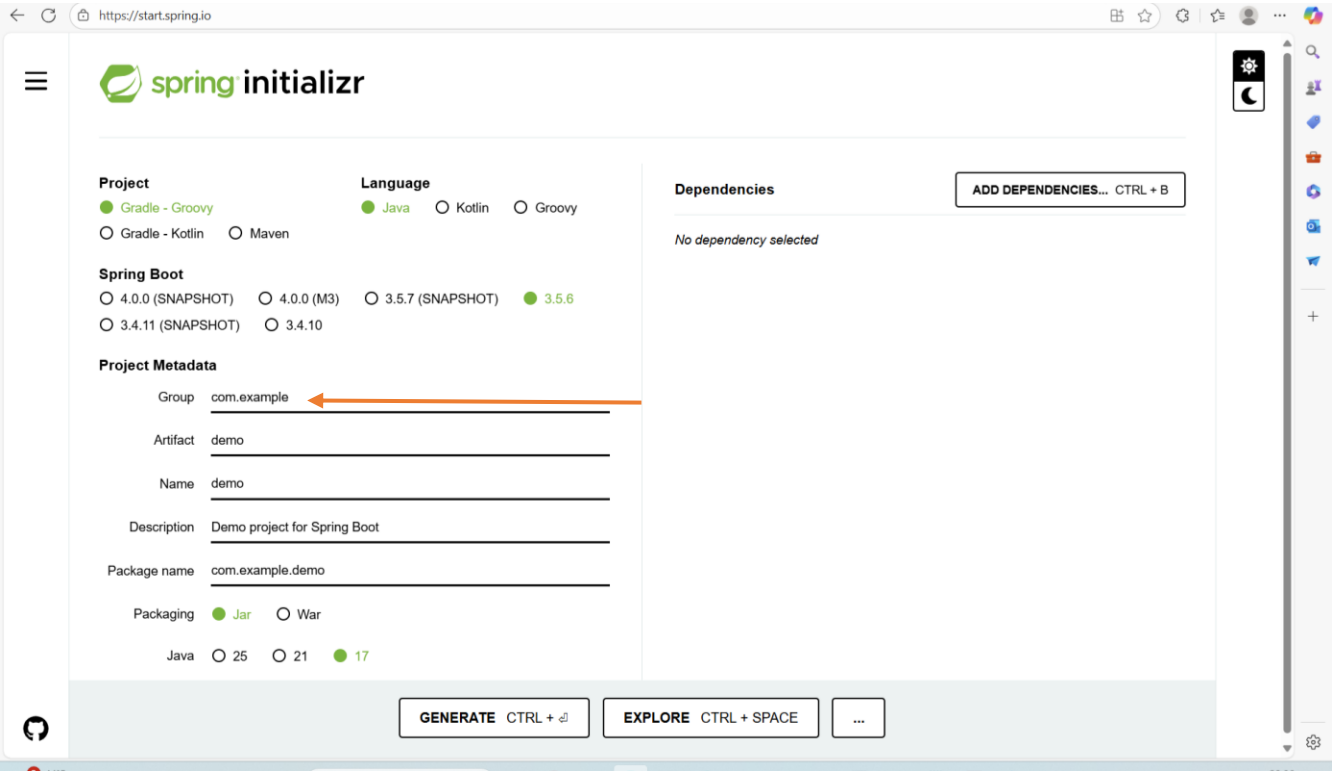
Let us start by generating a simple Spring boot project. Go to http://start.spring.io



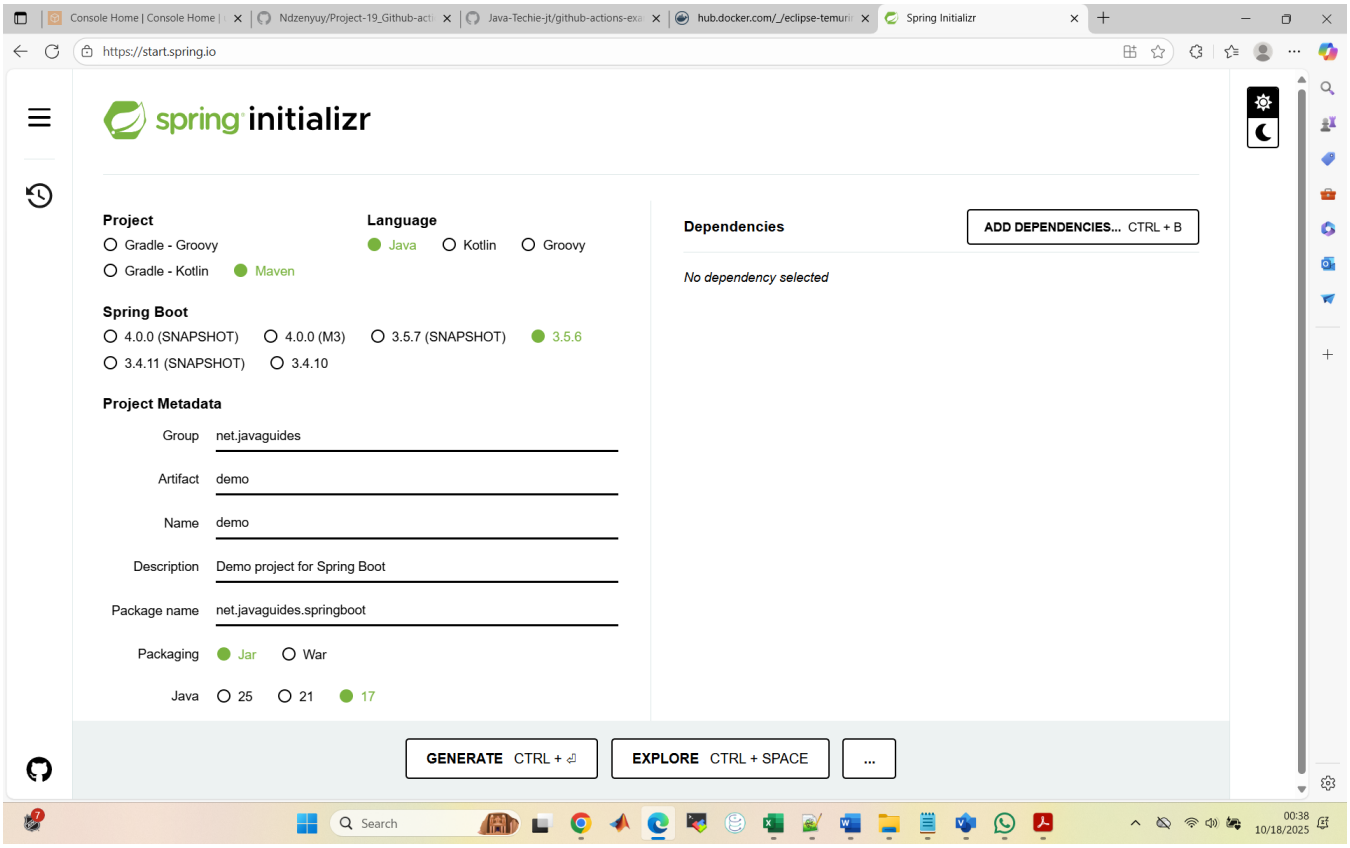
On "**Project**", we will choose "**Maven**"



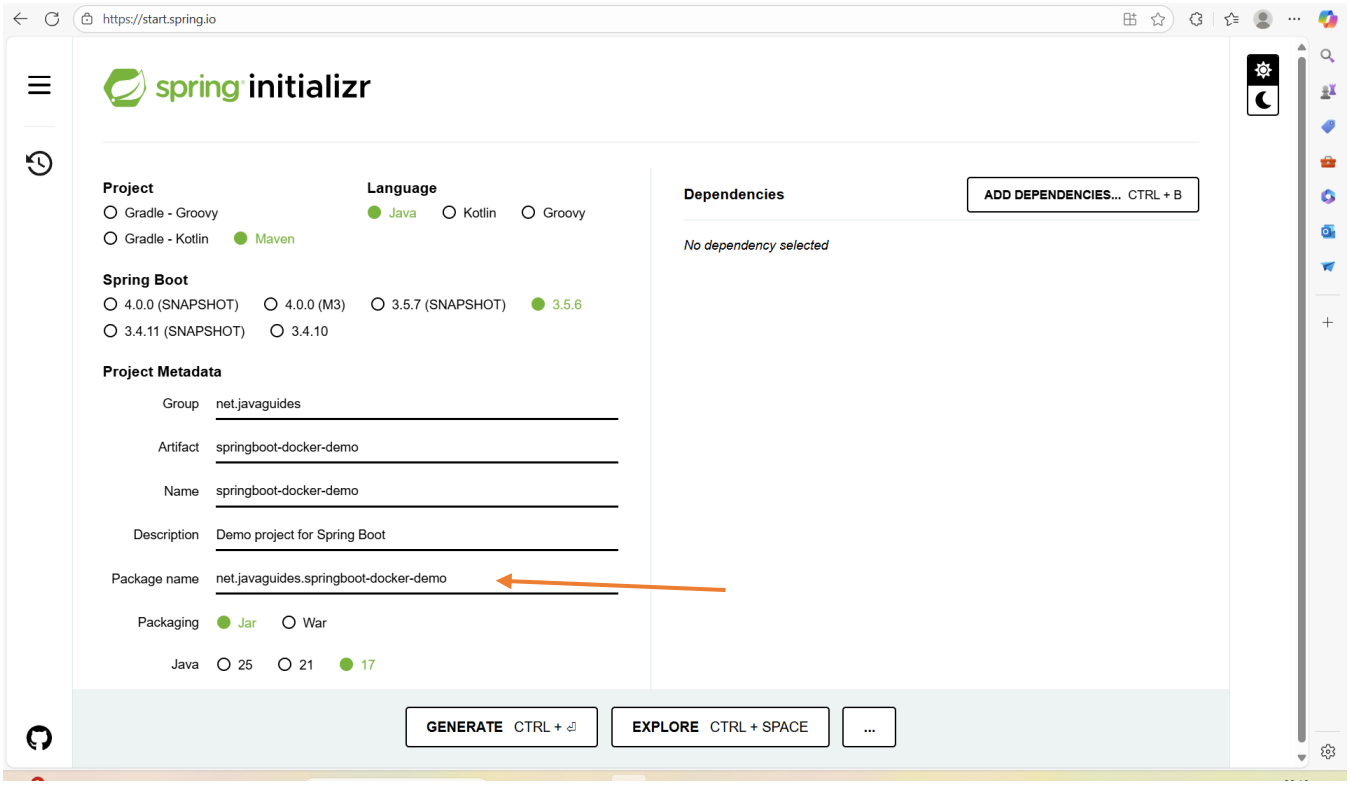
On "**Spring Boot**", we will use version "**3.5.6**"



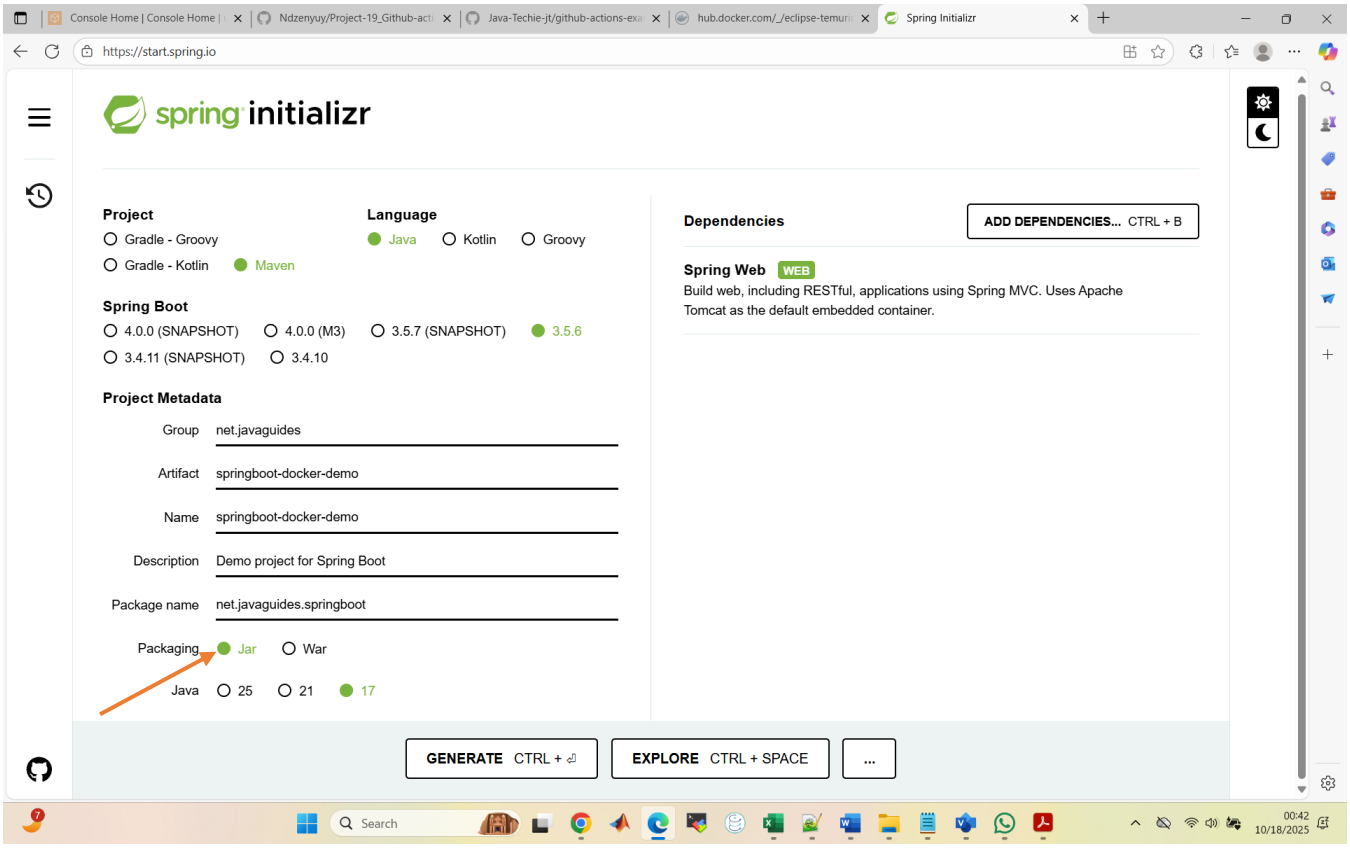
Let us change the Group to "**net.javaguides**"



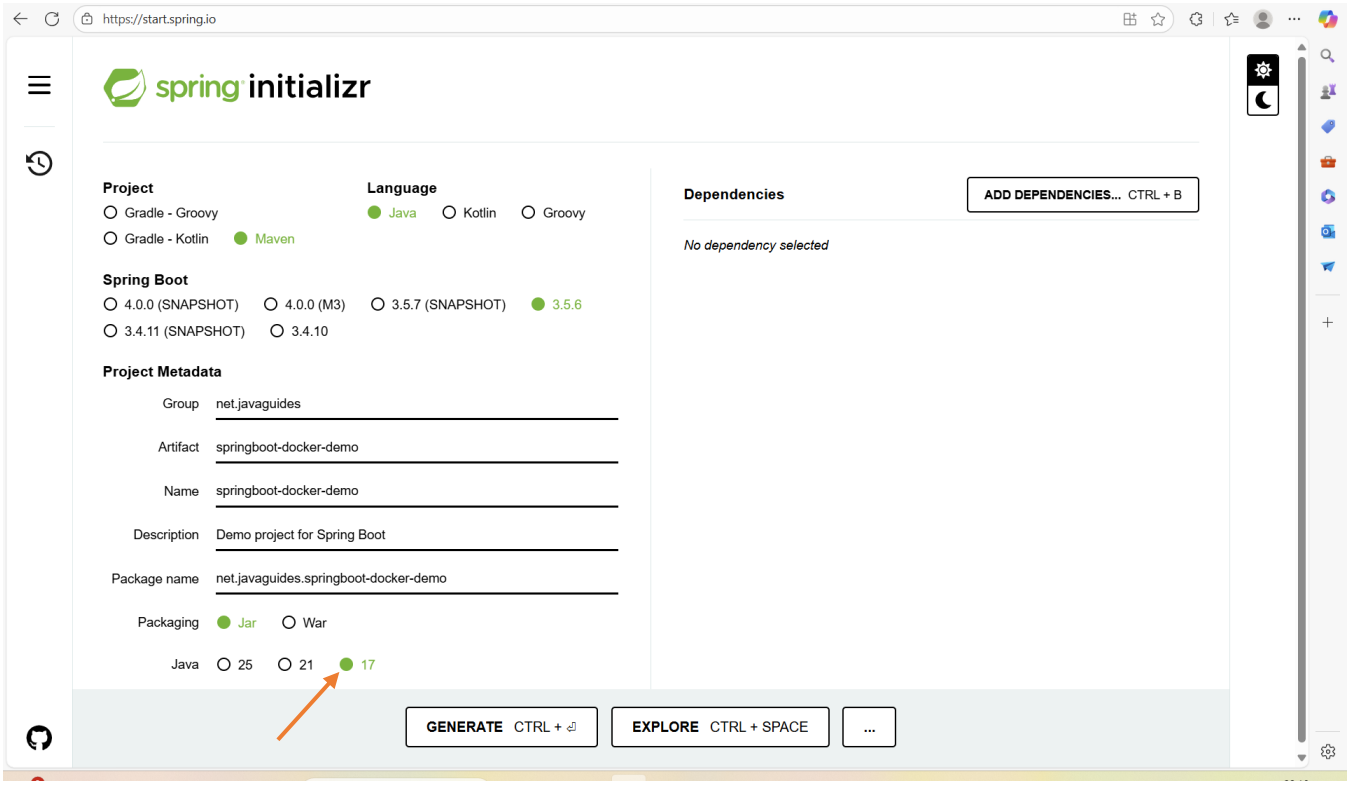
We have to give the project a name, I will call it "**springboot-docker-demo**"



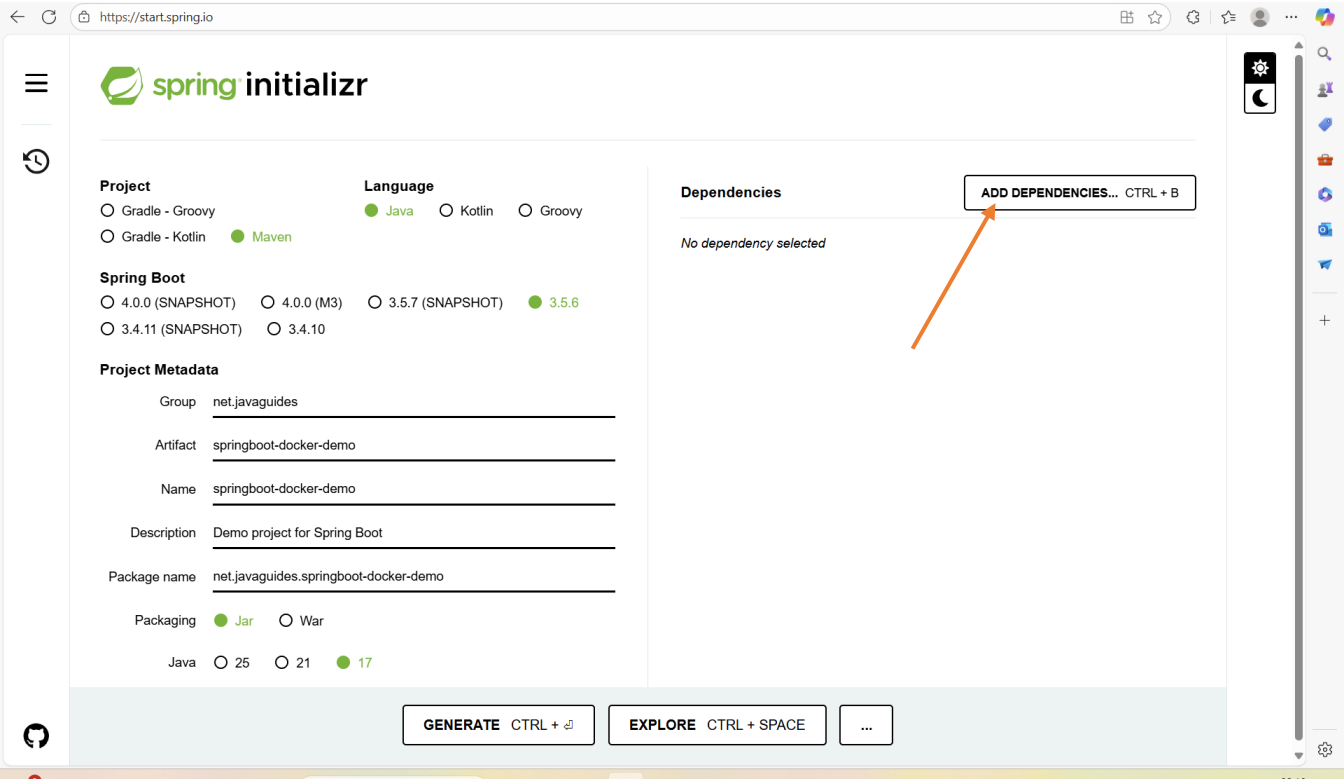
For "**Package Name**", we will use "**net.javaguides.springboot**"



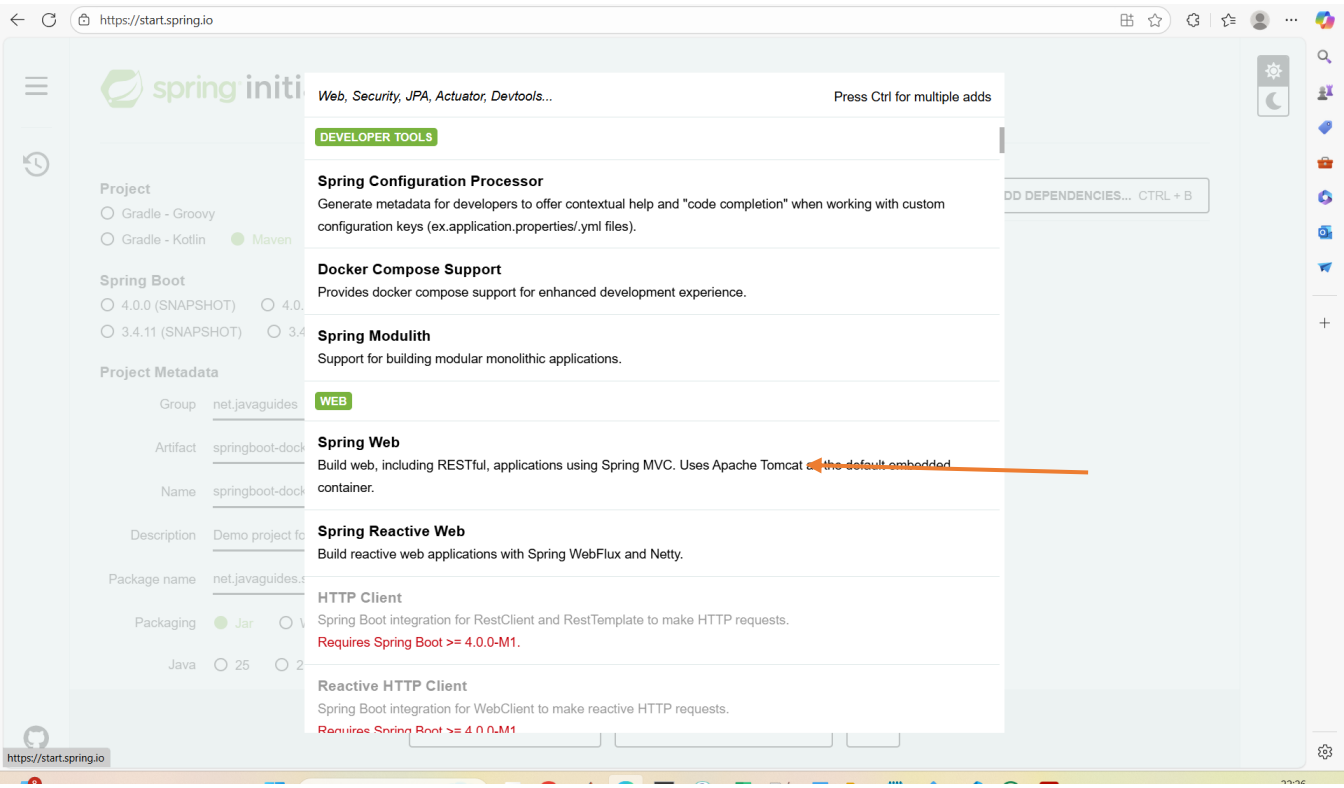
On “**Packaging**”, we will use “**Jar**”



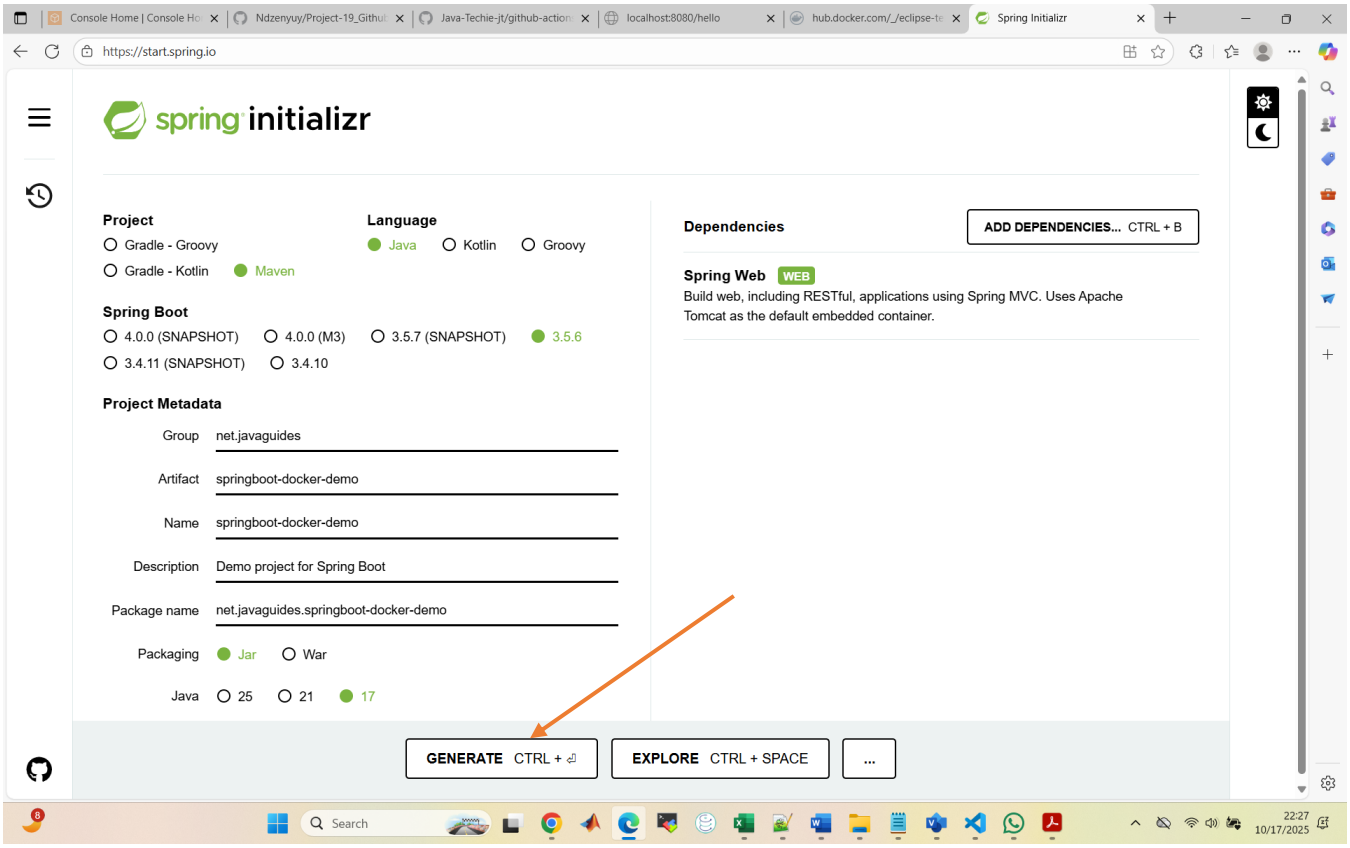
We will use Java version “**17**” or above. So, we will just use Java Version 17



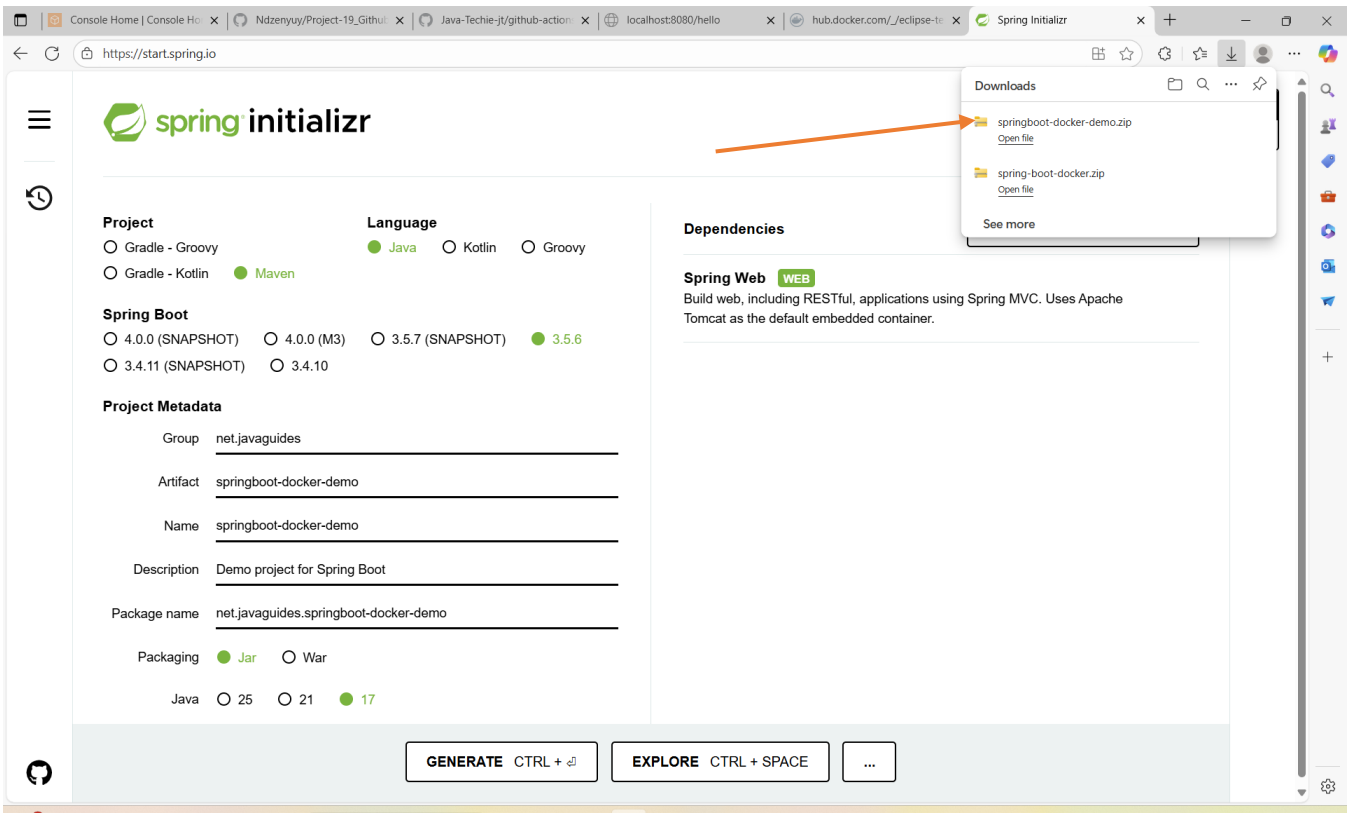
Click on "**Add Dependencies**"



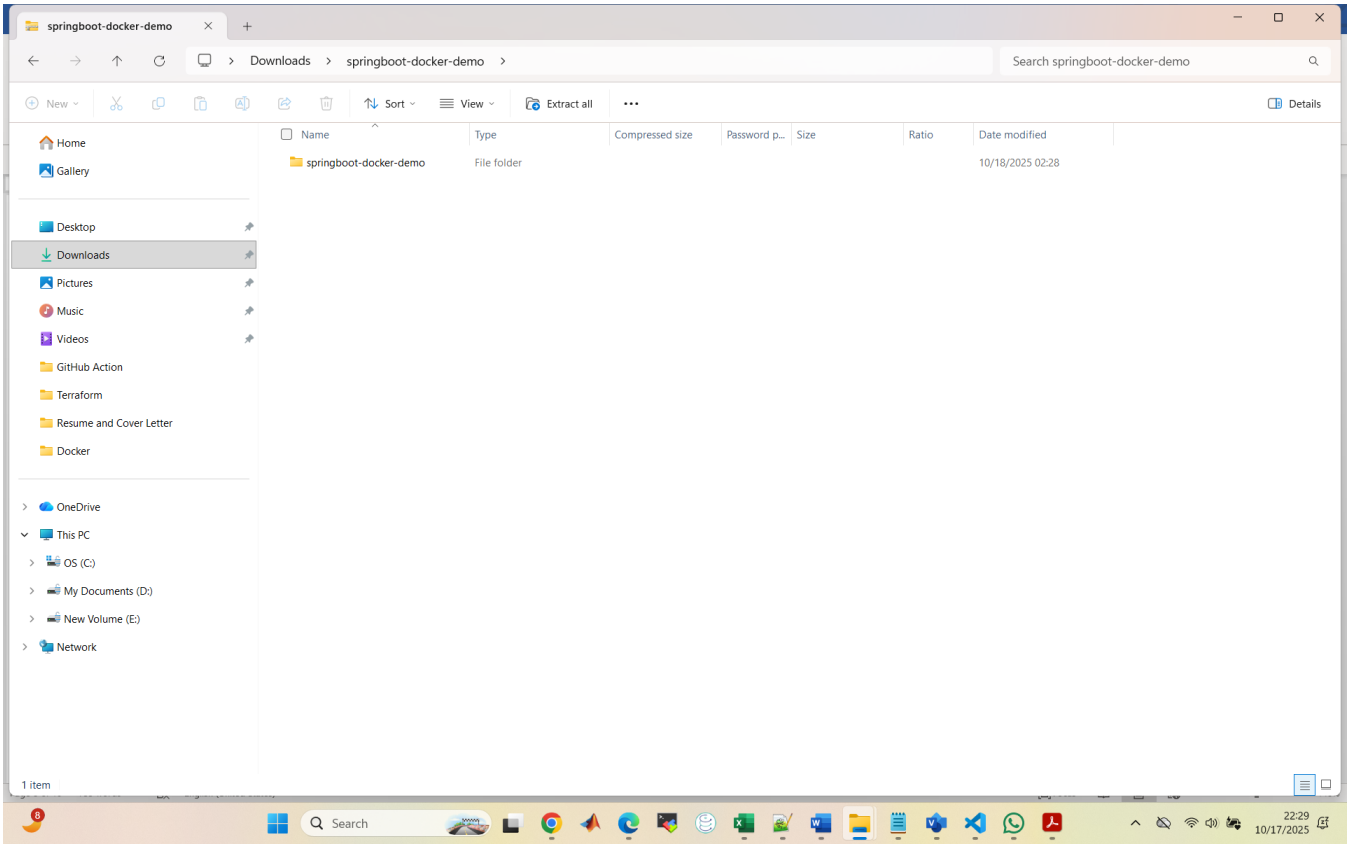
Click on "**Spring Web**"



Then click on "**Generate CTRL**"

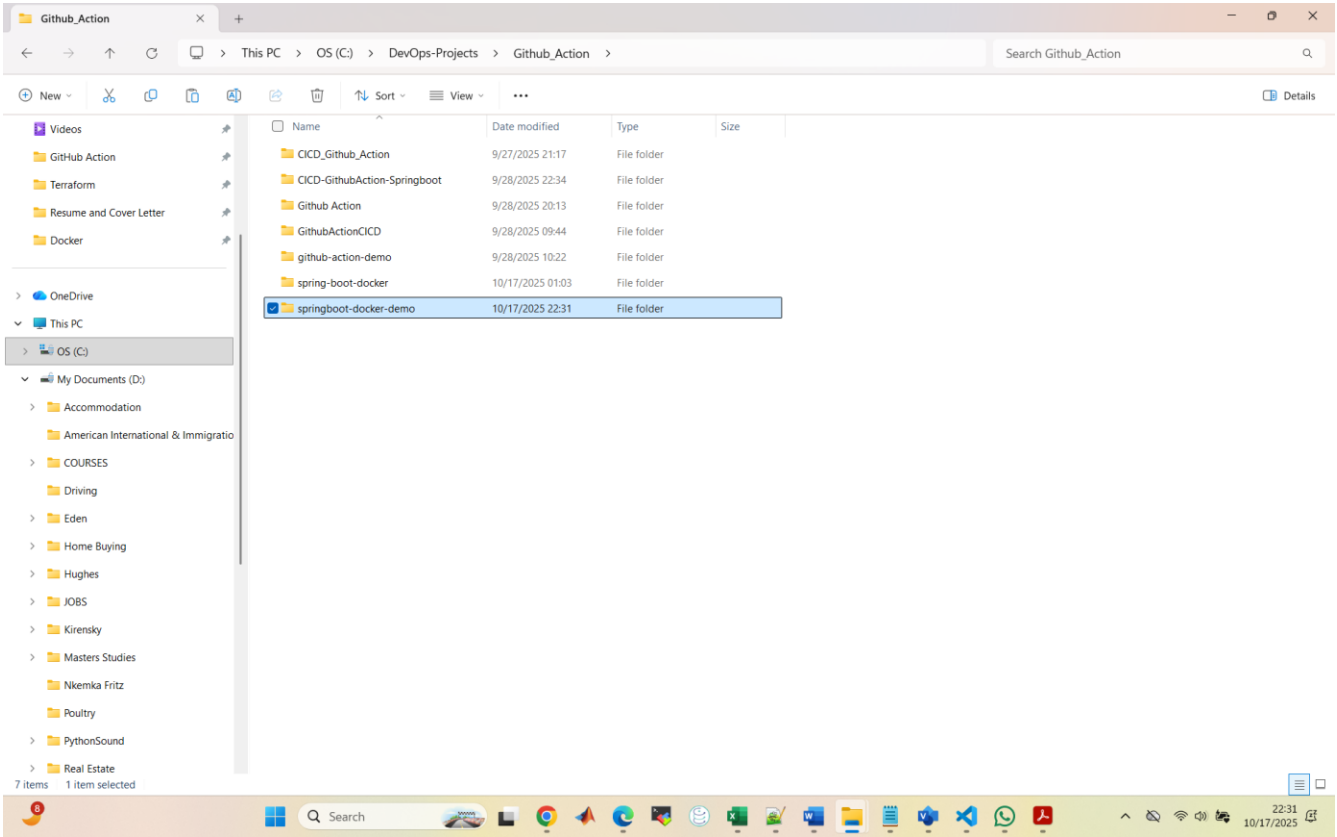


The project has been created and the project folder has been downloaded to the "**Downloads**" folder.



Copy the project folder and save in a suitable location. I will save it in this location:

`C:\DevOps-Projects\Github_Action\springboot-docker-demo`

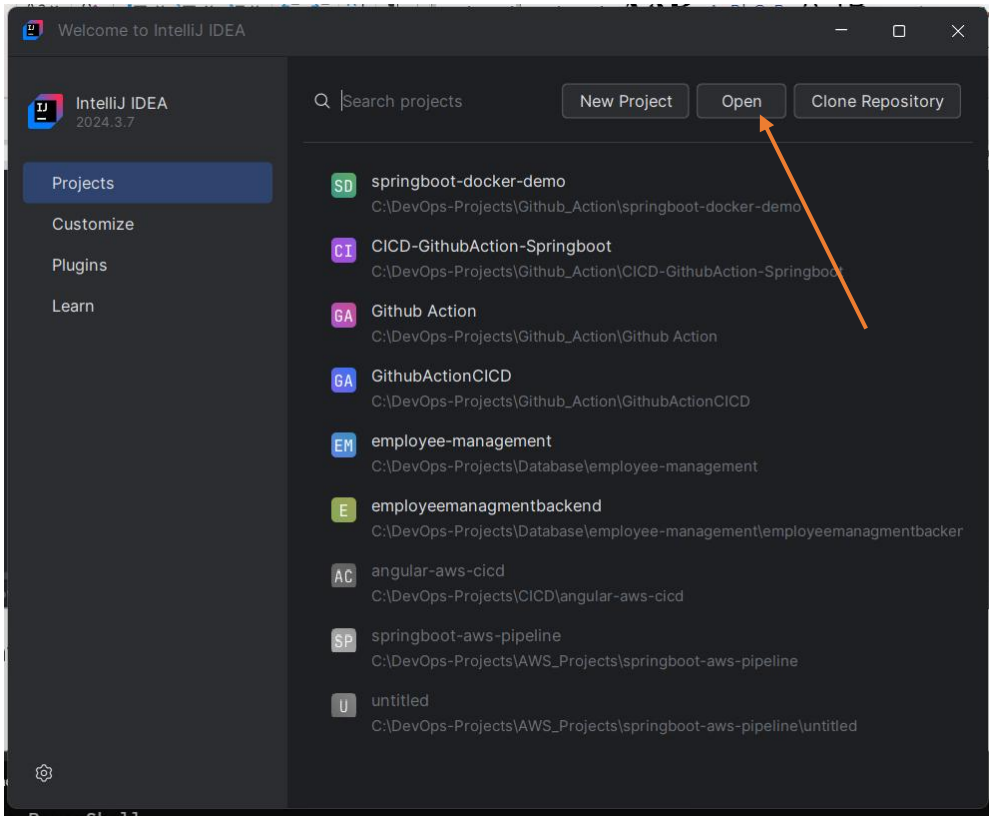


STEP 2: Build a Simple REST API

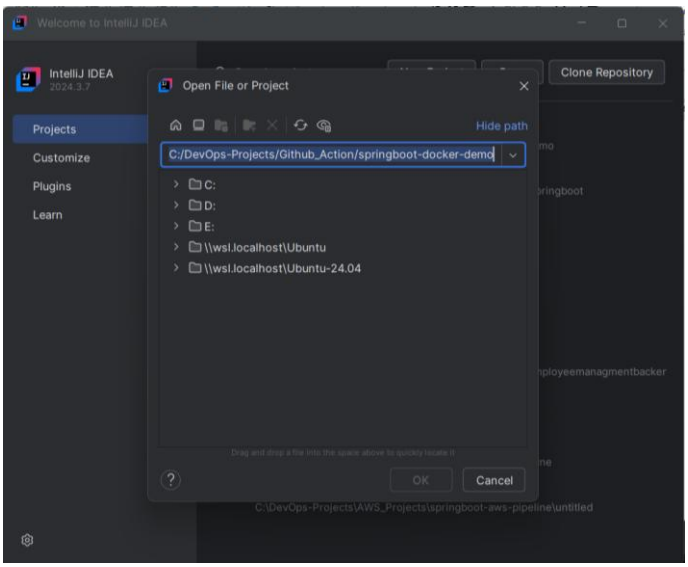
We are going to open the project in IntelliJ. This is done as follows

Part 1: Open IntelliJ and Navigate to Project Folder

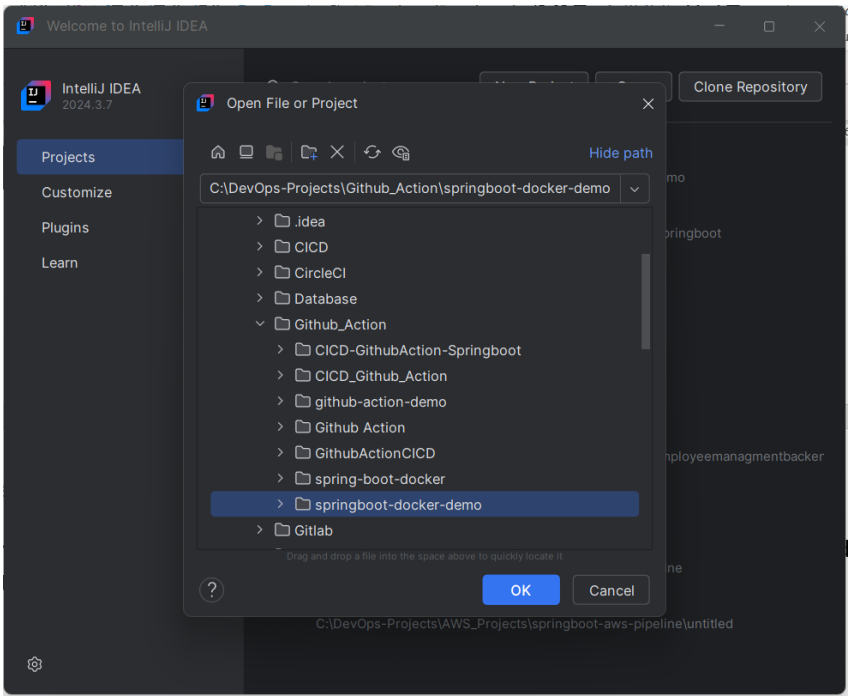
Open IntelliJ



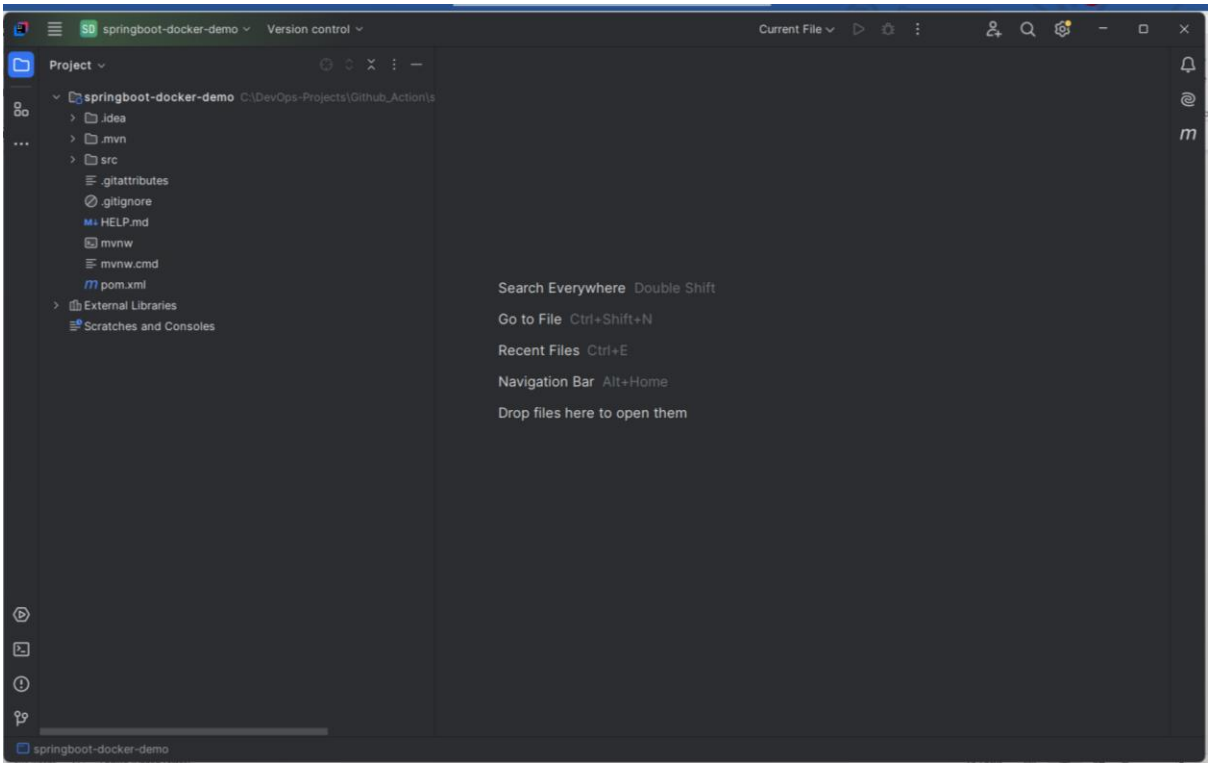
Navigate to the project folder. Click on "**Open**"



Our project file is located in C:\DevOps-Projects\Github_Action\springboot-docker-demo. So, we navigate to the project folder.

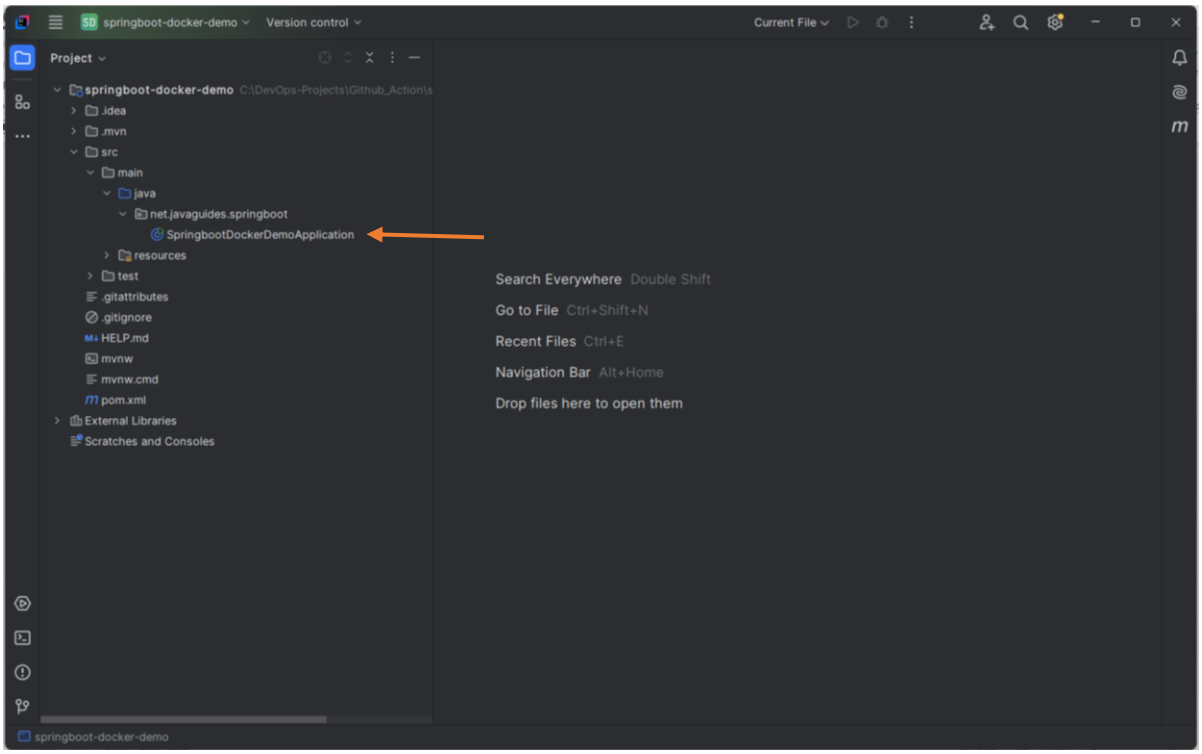


Click on "**OK**"

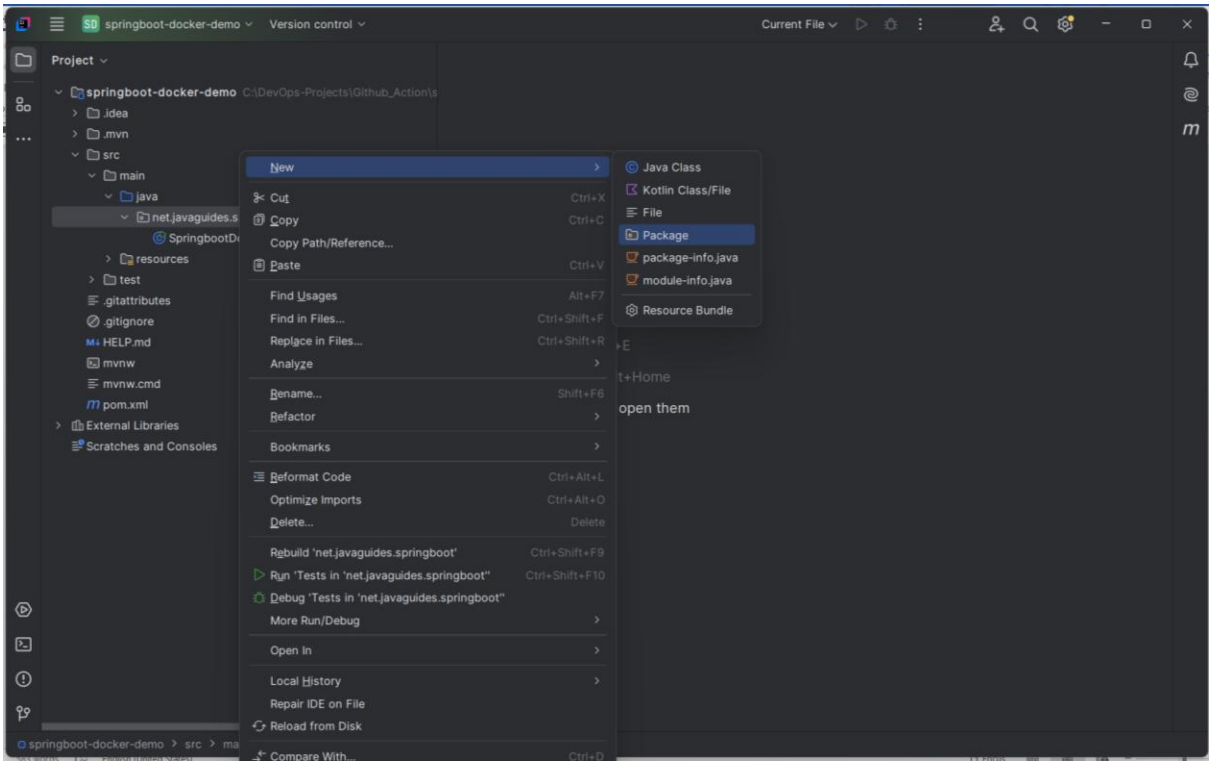


Part 2: Create a Java Package

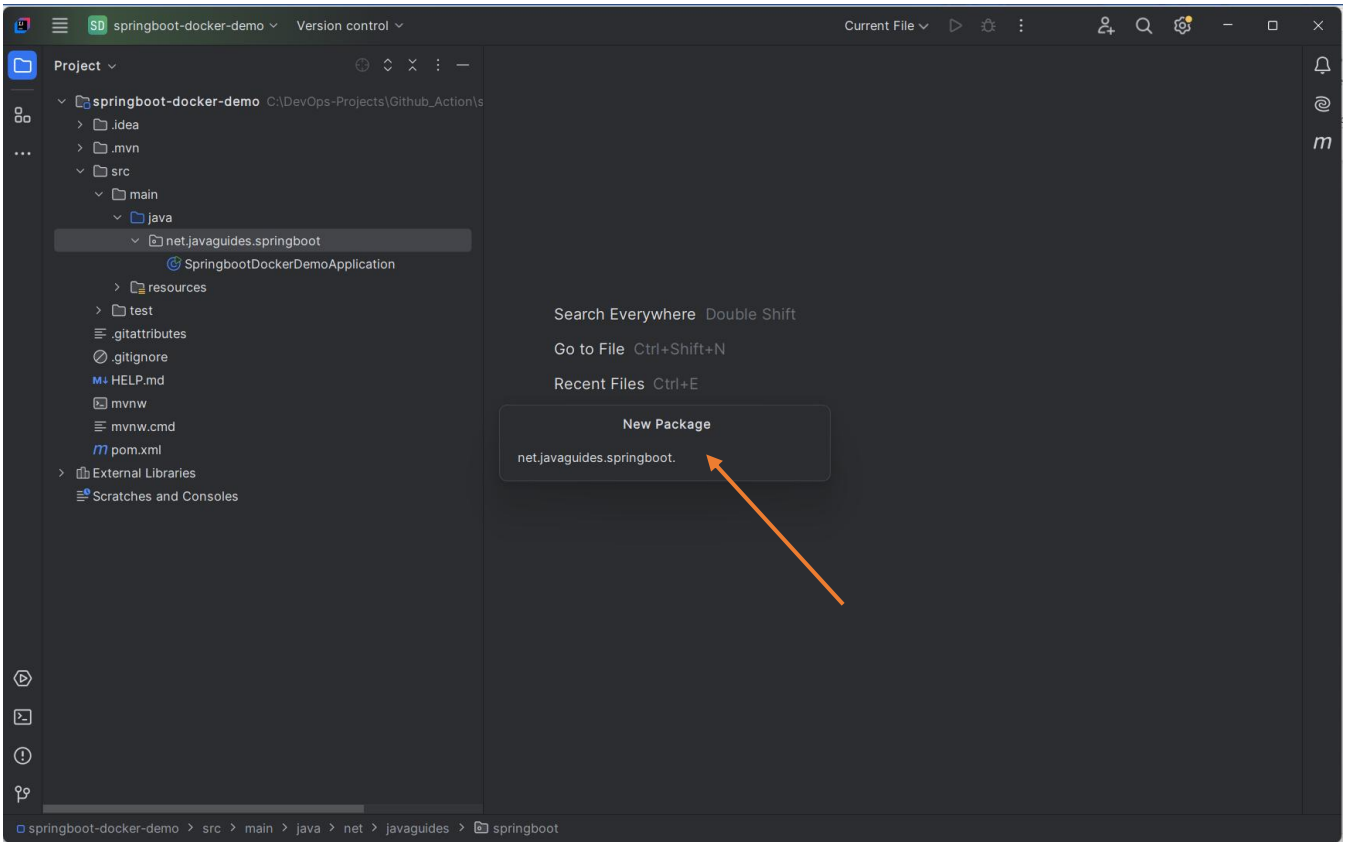
Let us create a new folder in "**src**" -> "**main**" -> "**Java**"



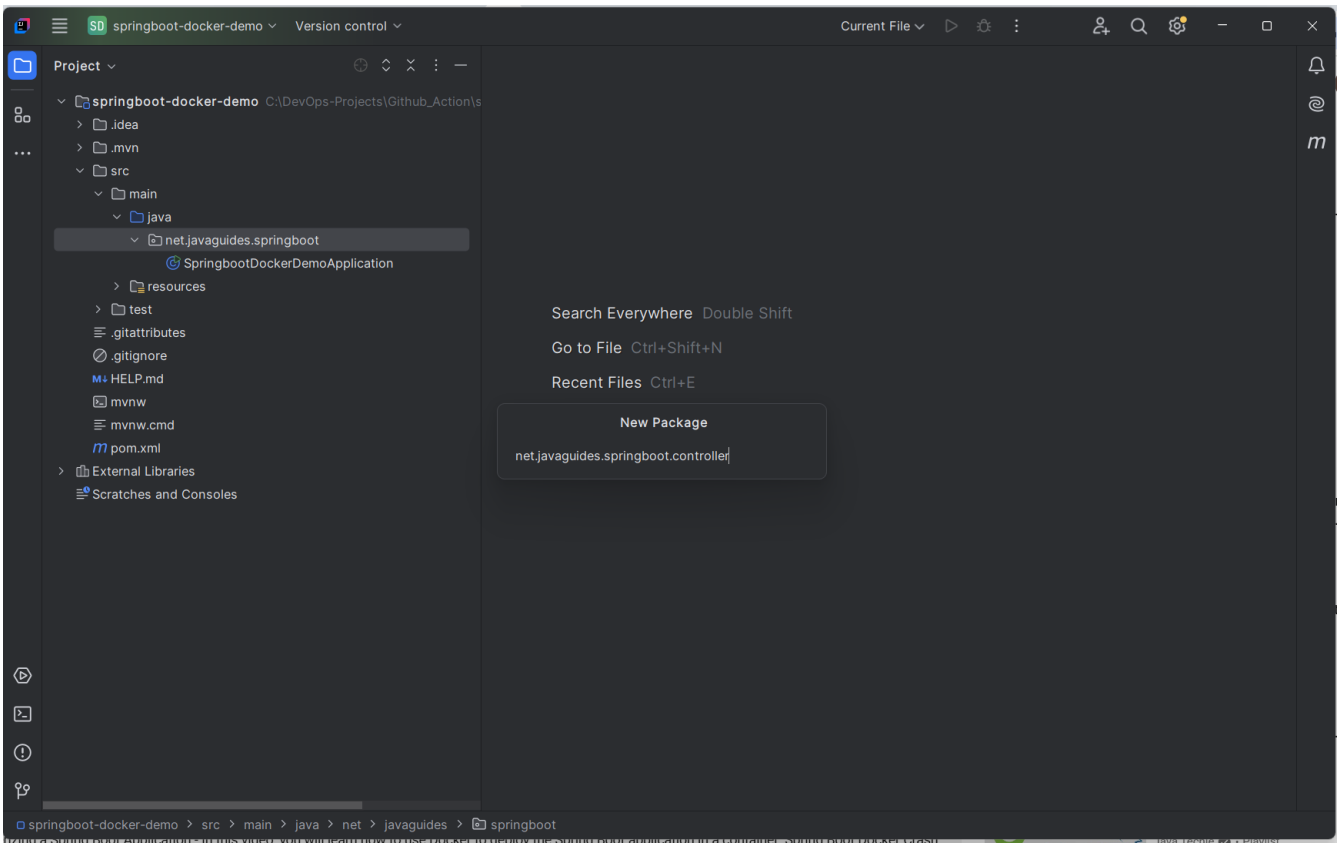
We will create a package. Right-click on "**net.javaguides.springboot**"



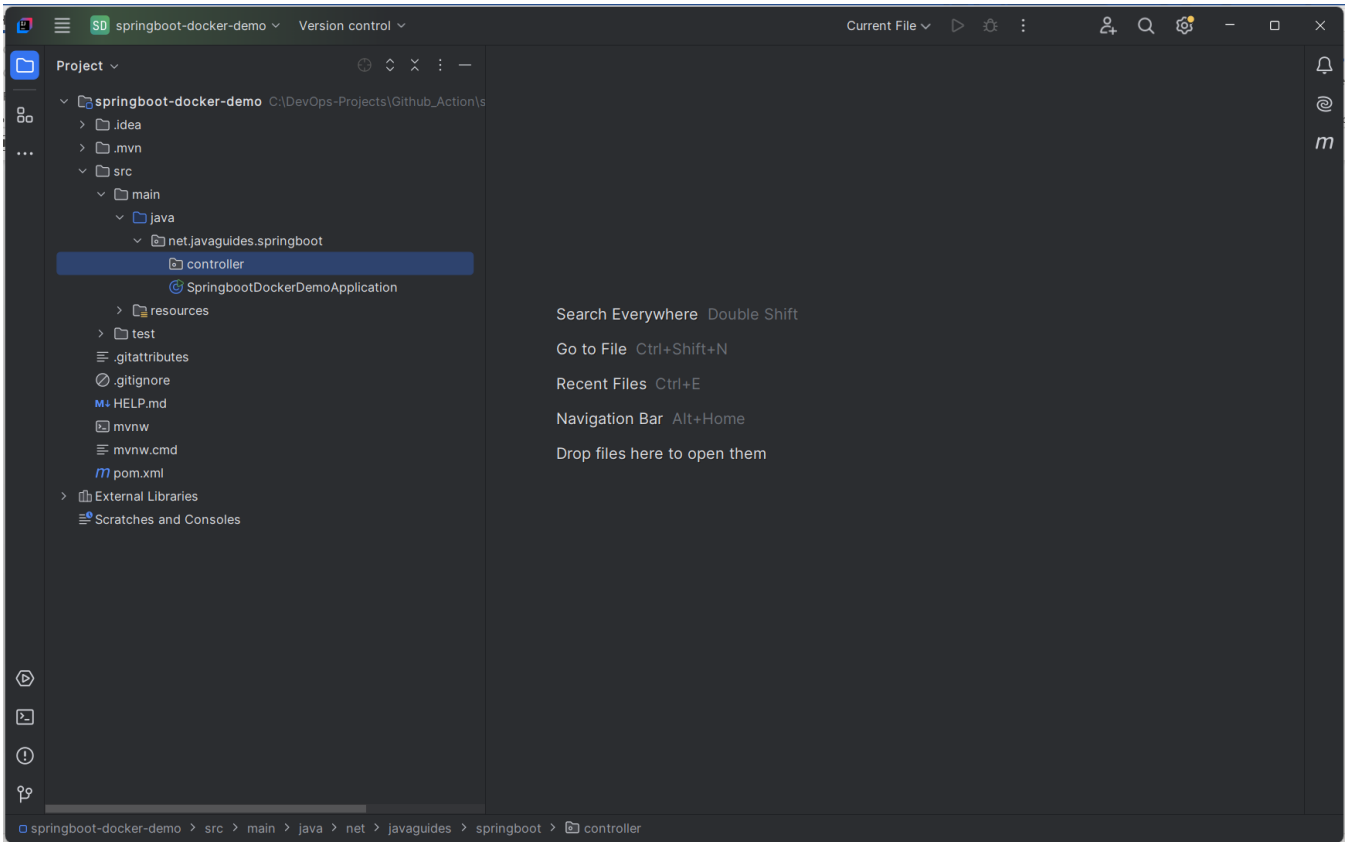
Select "**Package**"



Name it "**controller**"

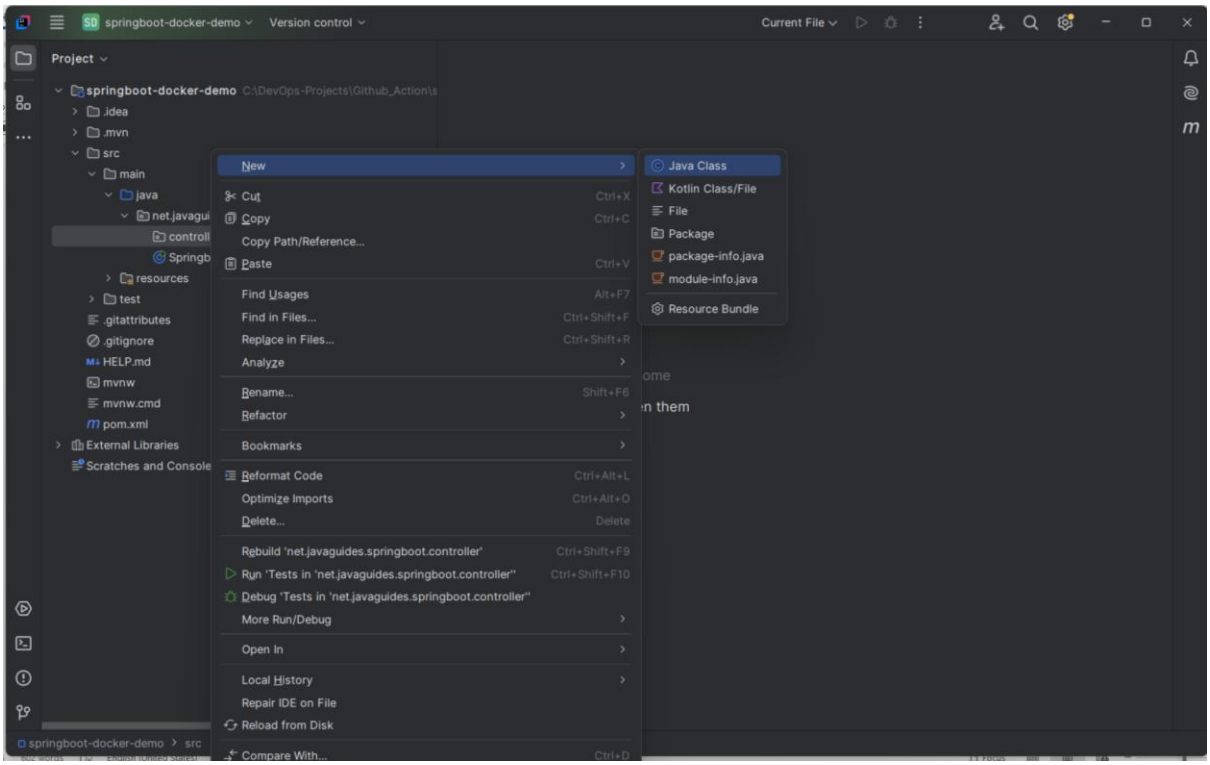


And press "**Enter**"

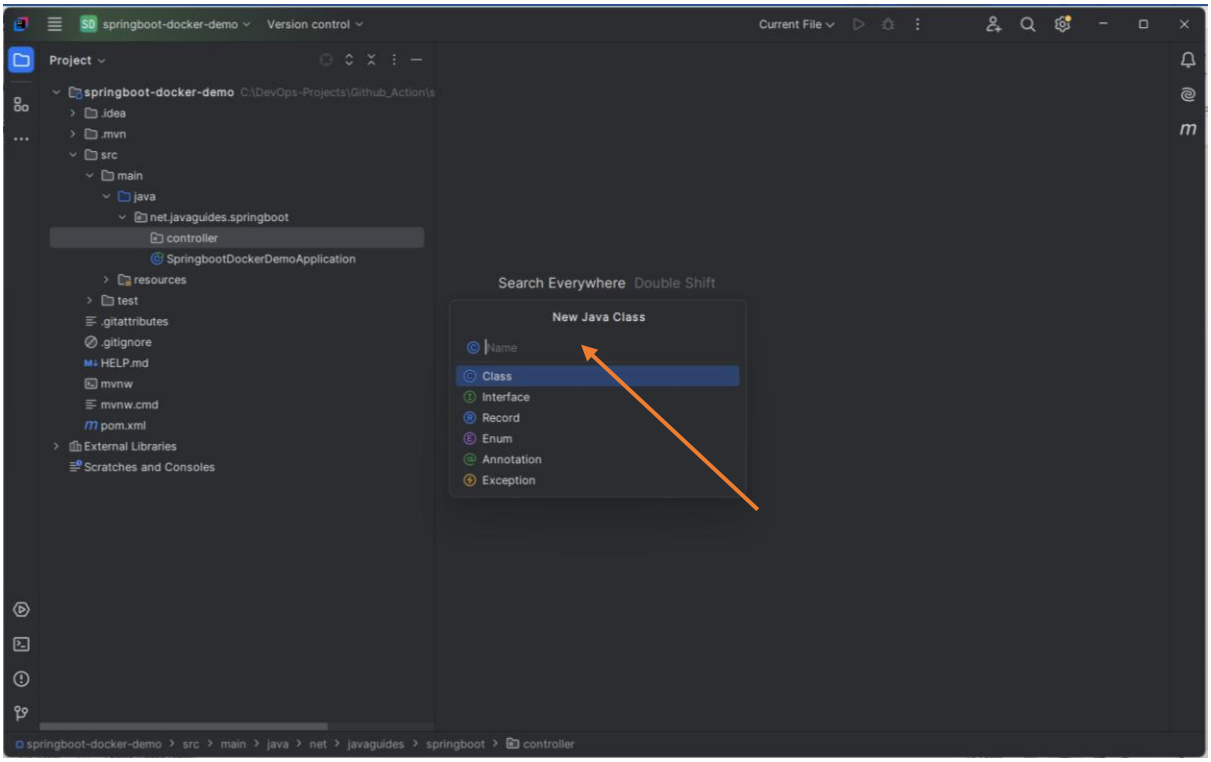


Part 3: Create Java Class file

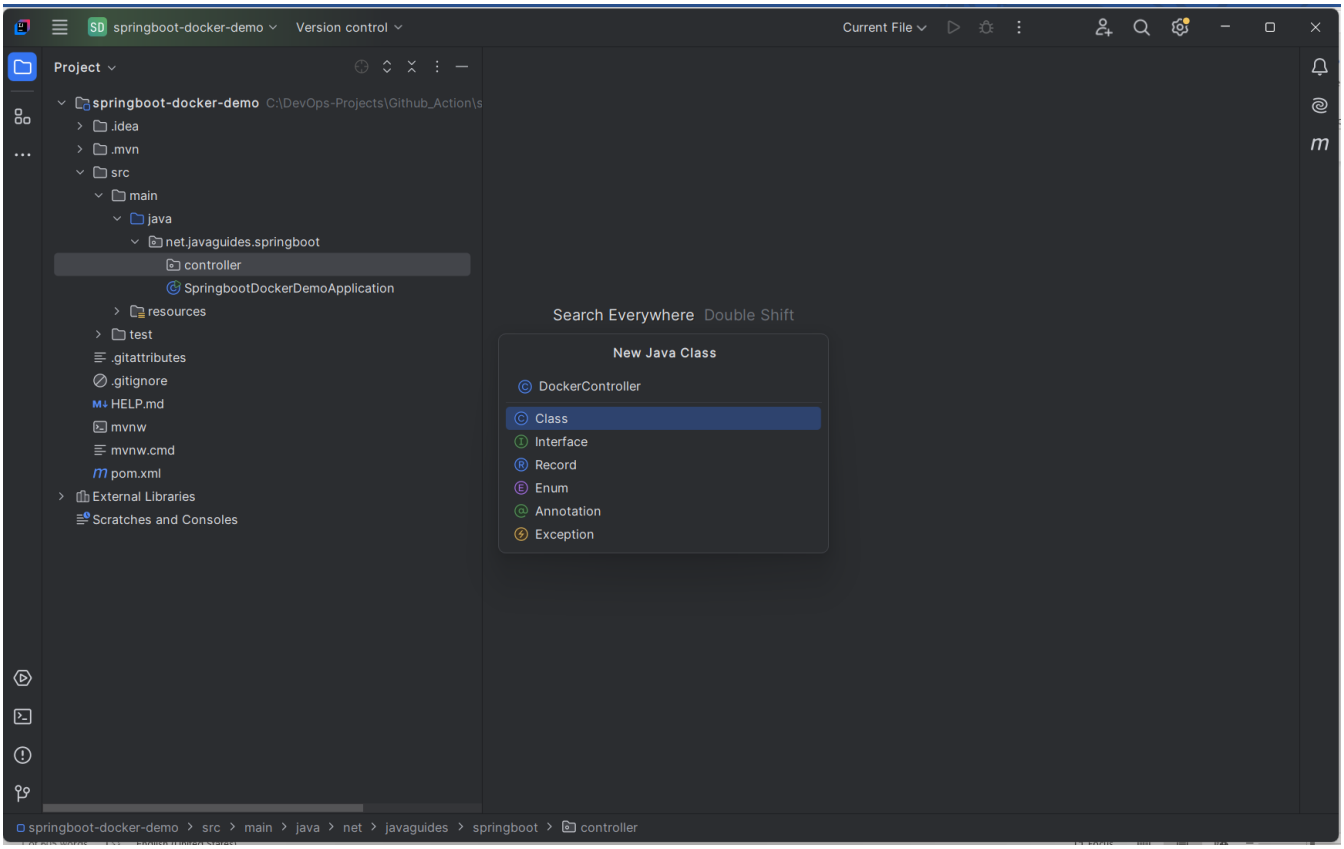
Let us create a new Java class file in the "**controller**" package. Right-click on "**controller**"



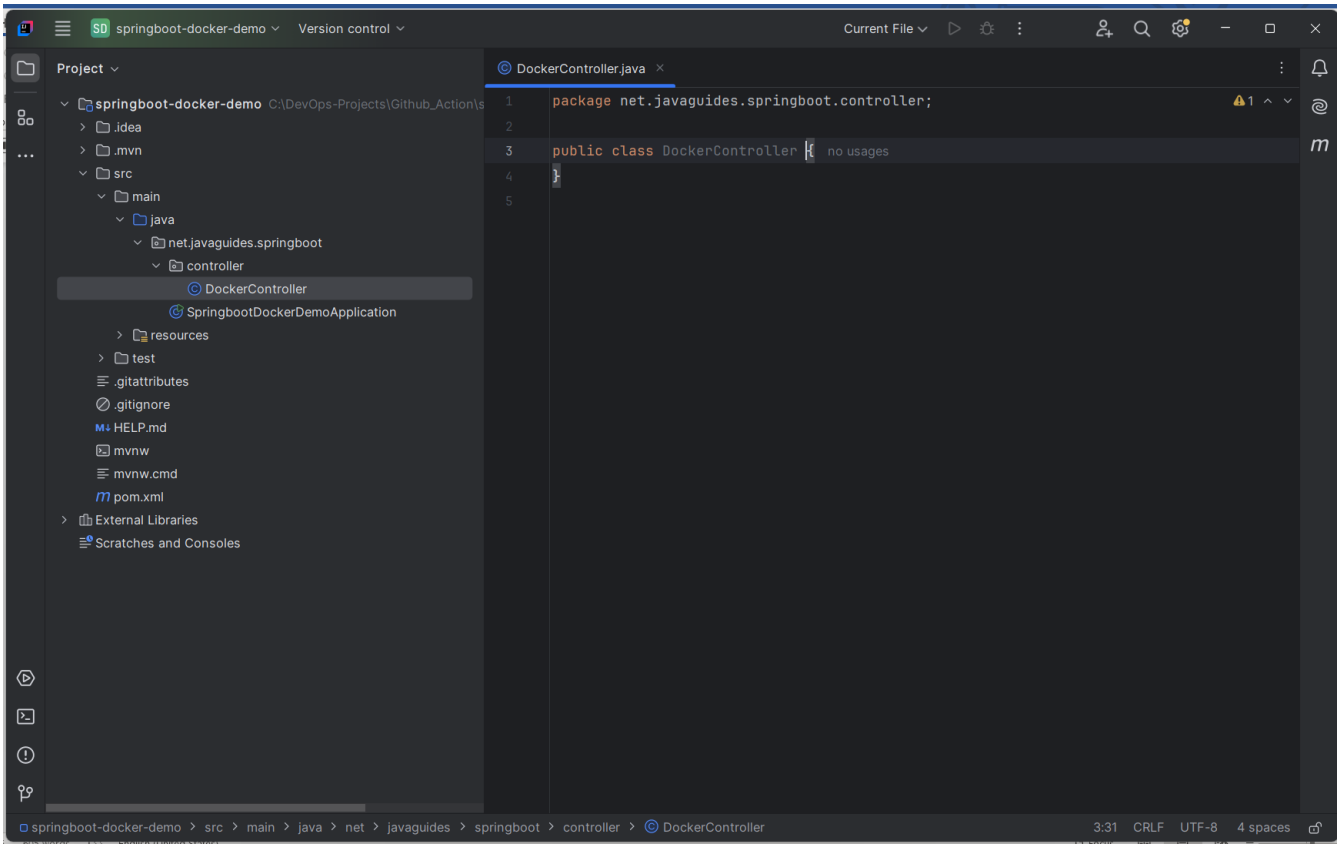
Select "**Java Class**"



We will create a simple endpoint called "**docker**". So, type "**DockerController**"

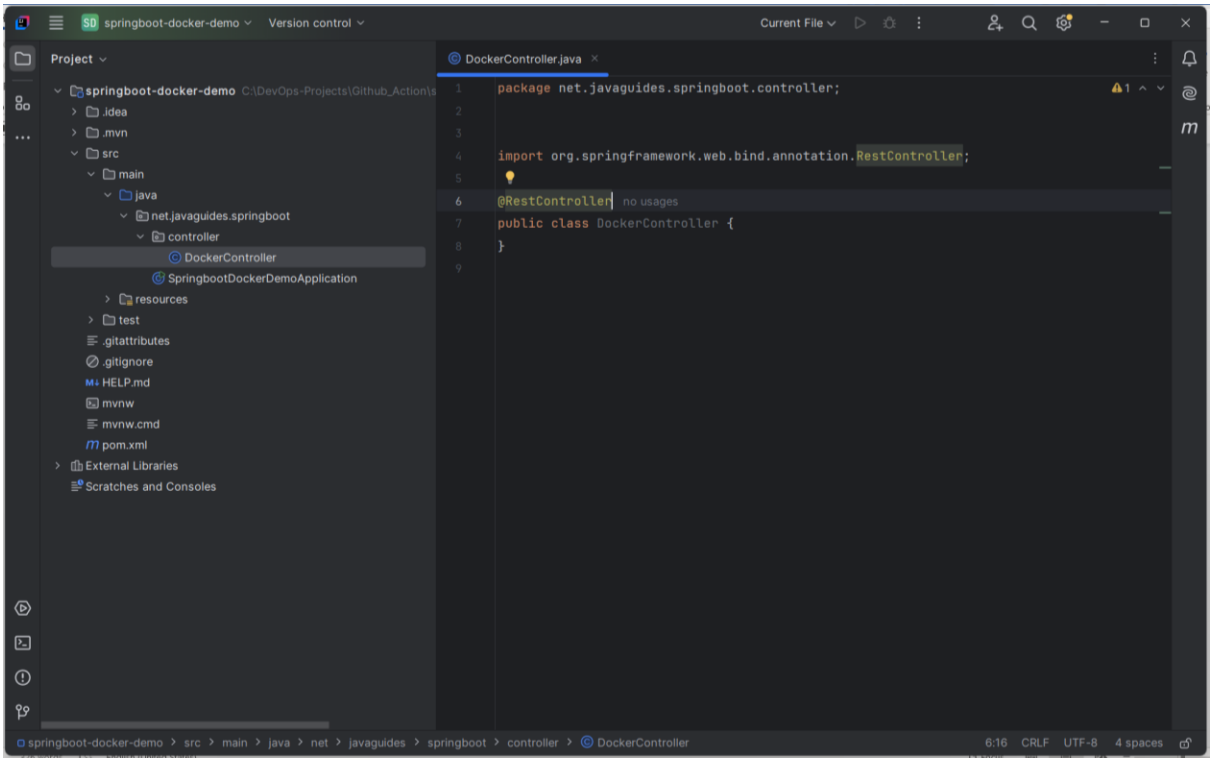


And press "**Enter**"



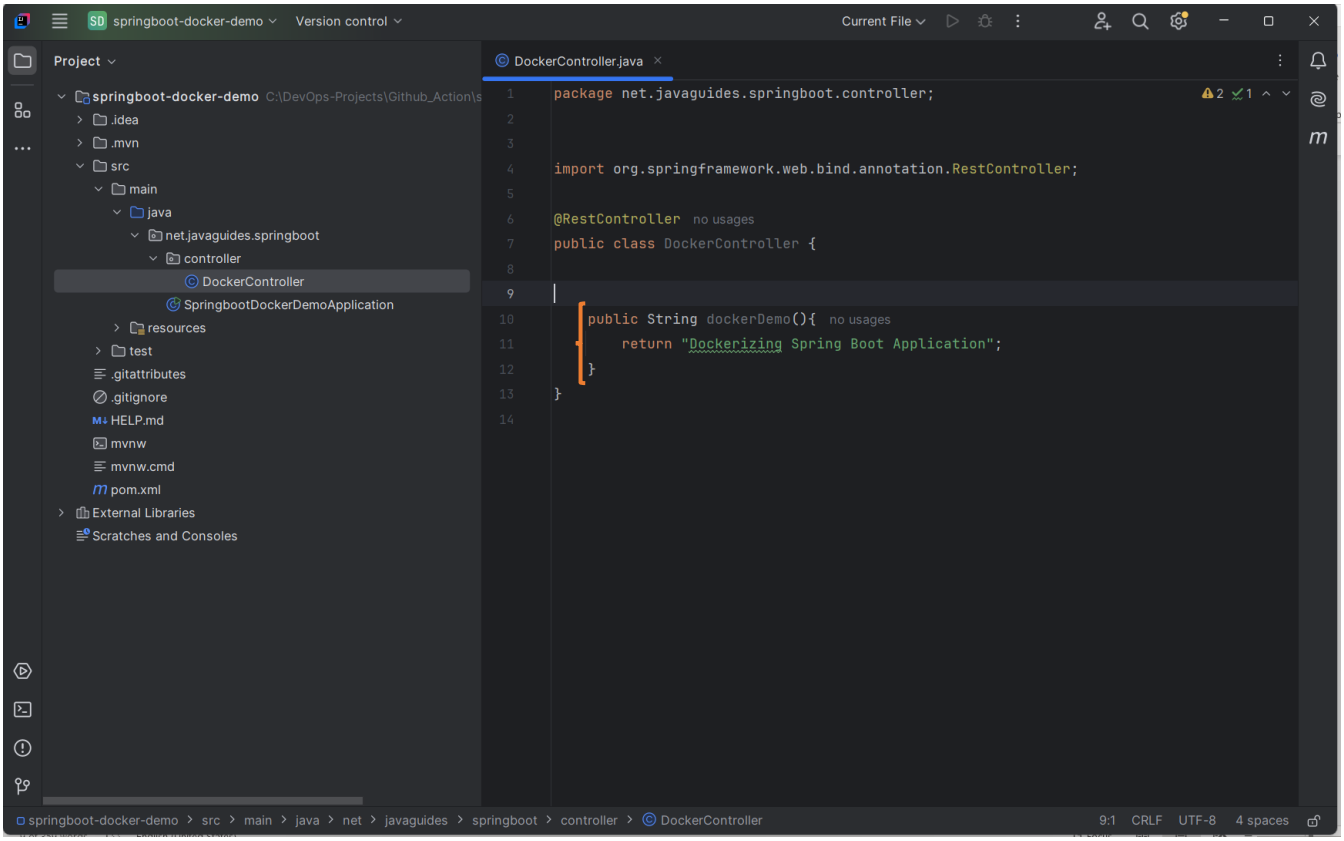
Let us go ahead and annotate this controller class to make this class a REST API controller. We will add this line of code.

```
@RestController
```



Within this controller, we can define a REST API by adding a method called "**dockerDemo**" that will return a string "**Dockerizing Spring Boot Application**"

```
public String dockerDemo(){
        return "Dockerizing Spring Boot Application";
    }
```

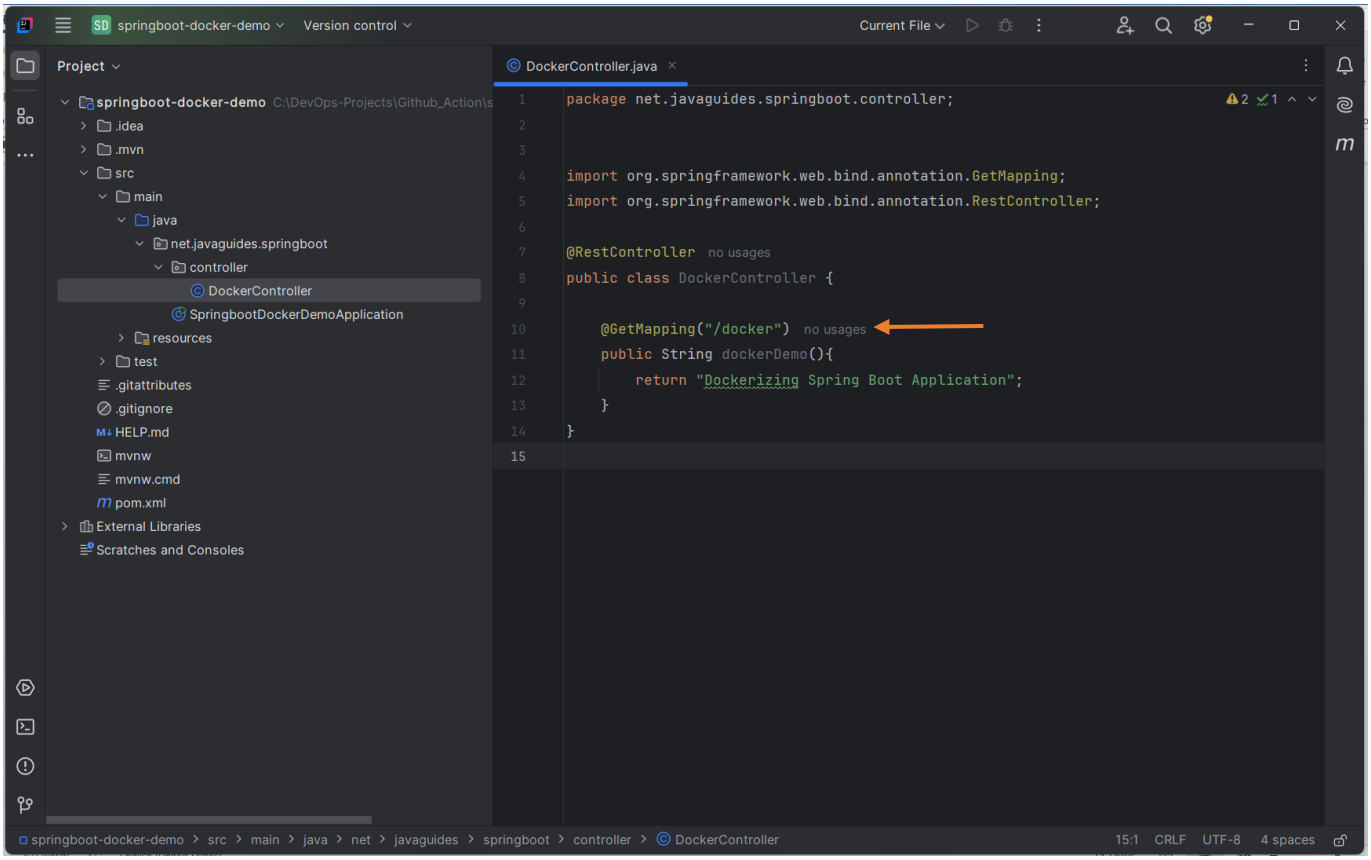


Let us annotate this method with by adding the line

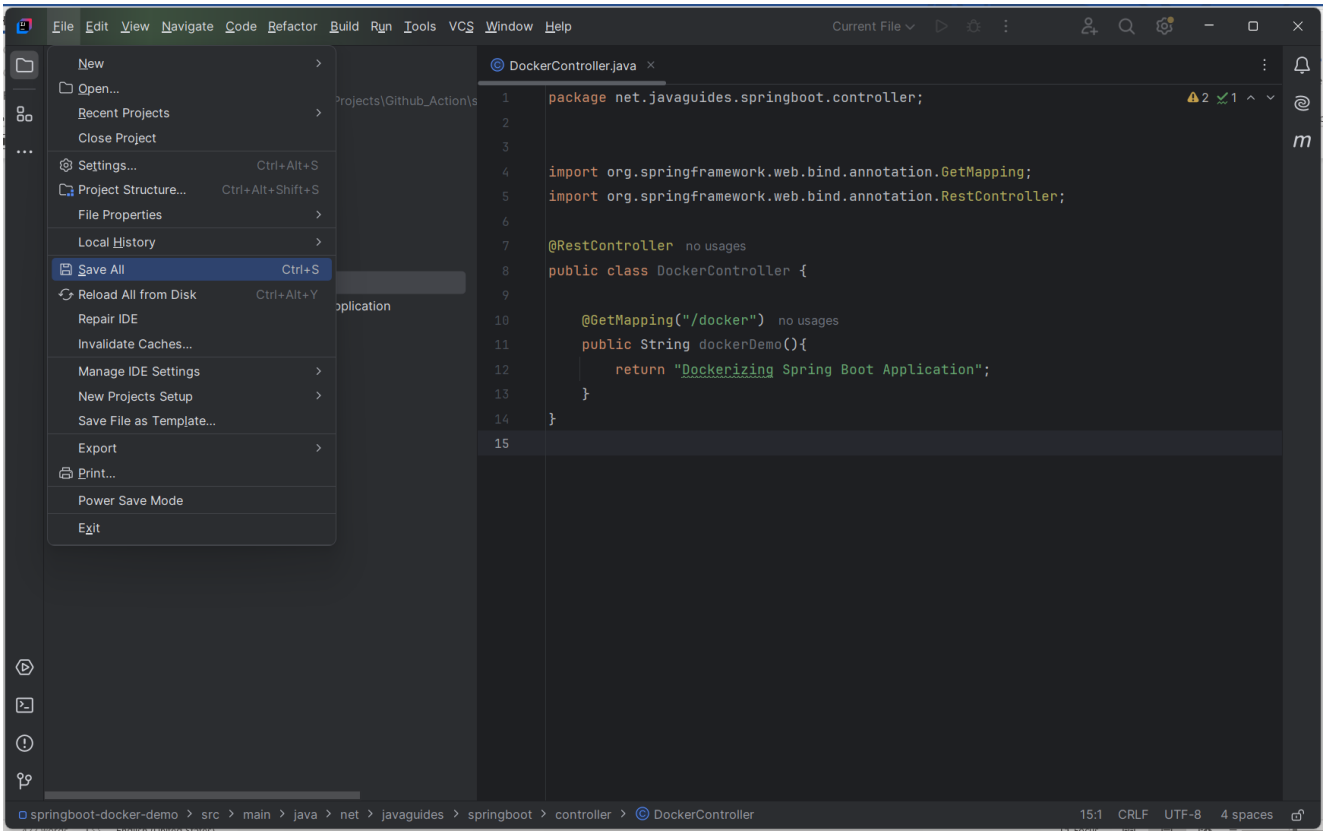
```
@GetMapping("/docker")
```

In this line, we have added an annotation called "GetMapping" with API URL called "docker".

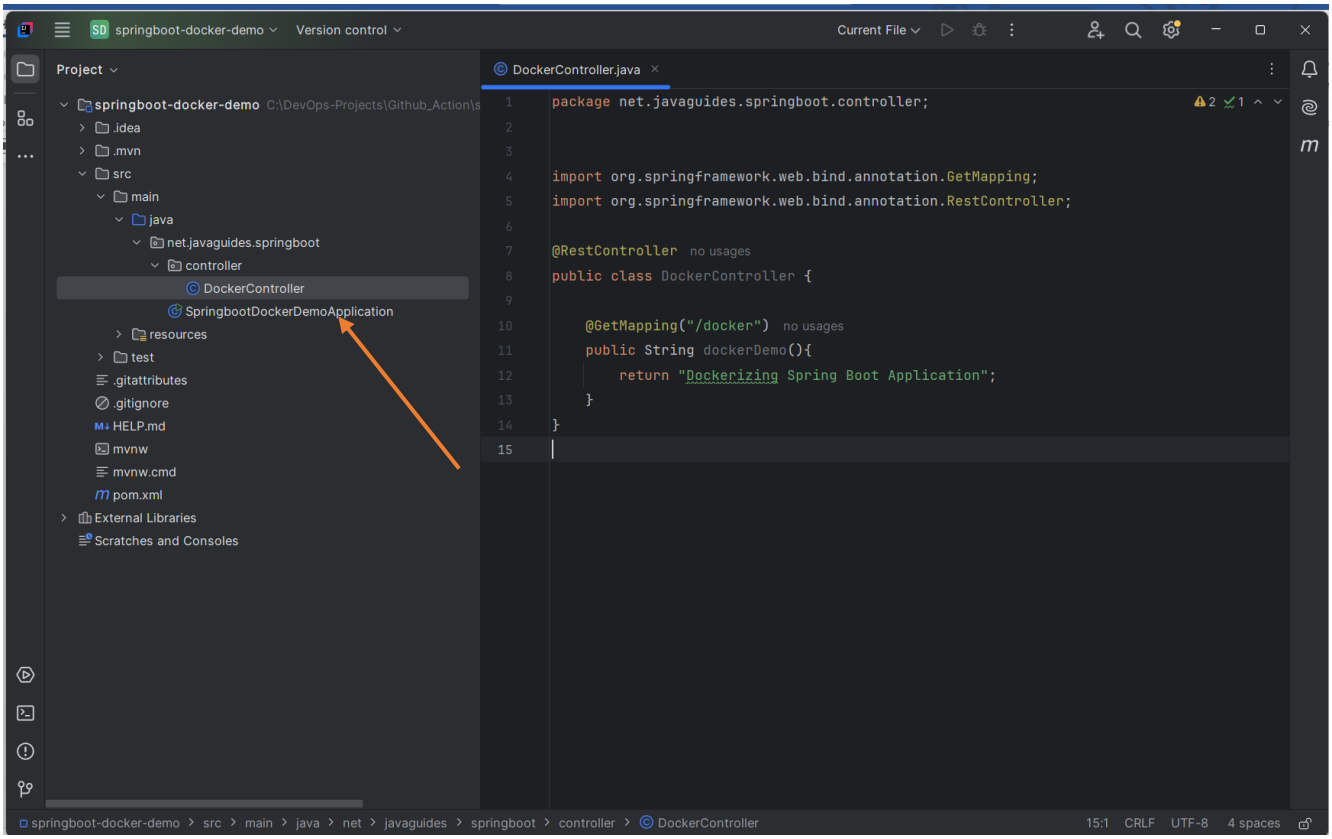
```
package net.javaguides.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class DockerController {

    @GetMapping("/docker")
    public String dockerDemo(){
        return "Dockerizing Spring Boot Application";
    }

}
```



Now, we have built a simple REST API. Save the project by clicking on "**File**"



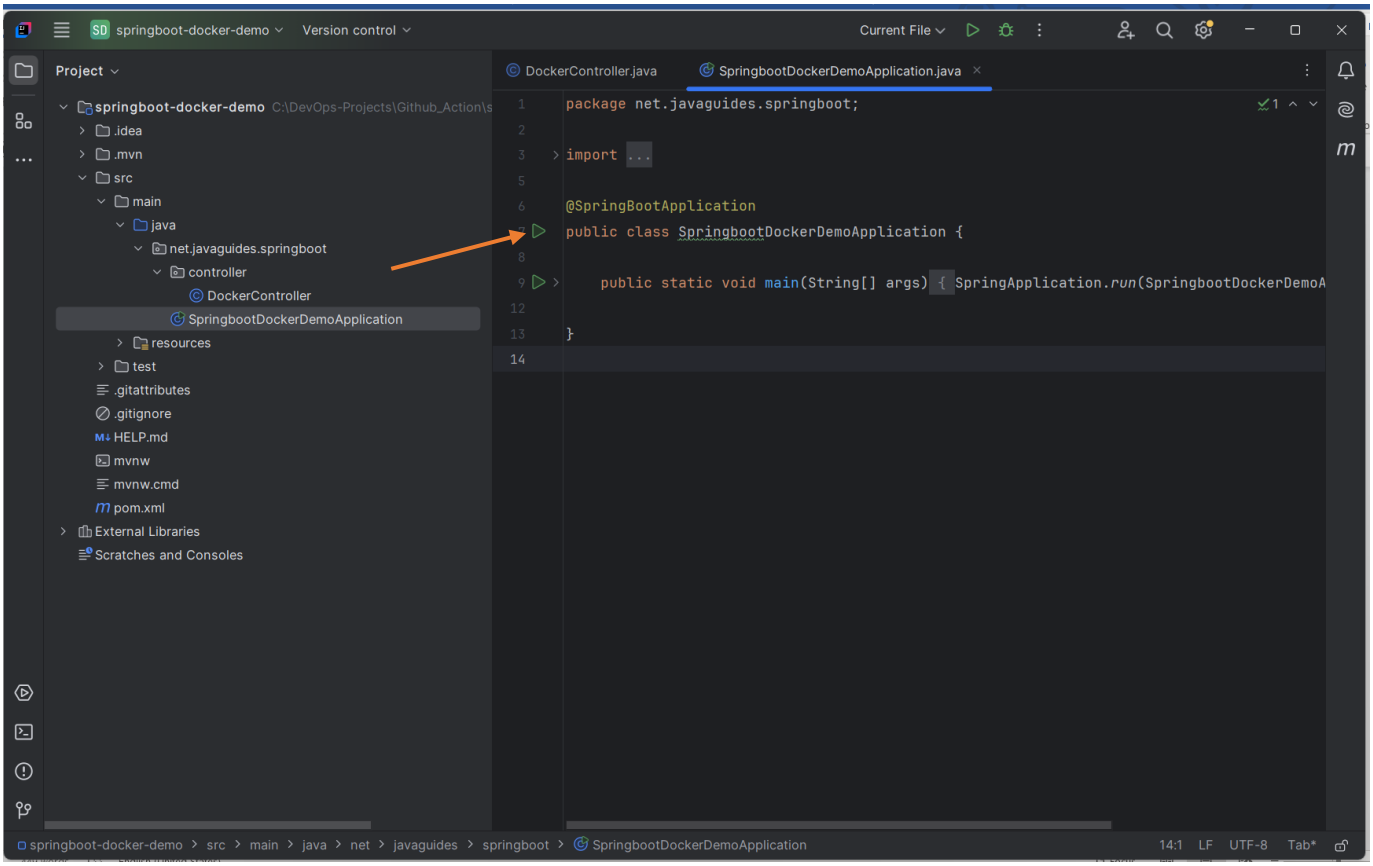
Then select "**Save All**"



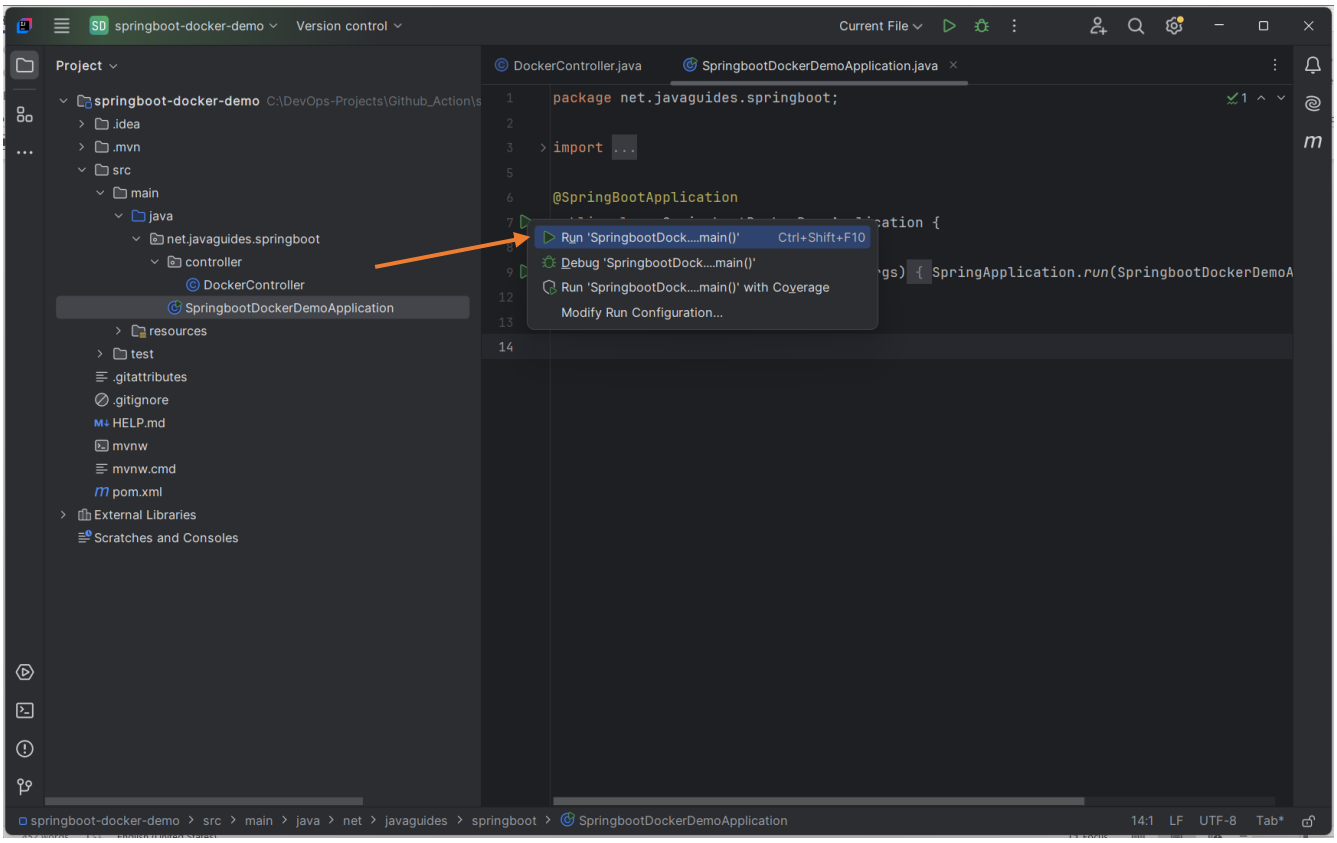
Part 4: Test the REST API on Browser

Now, let us test the application by running it.

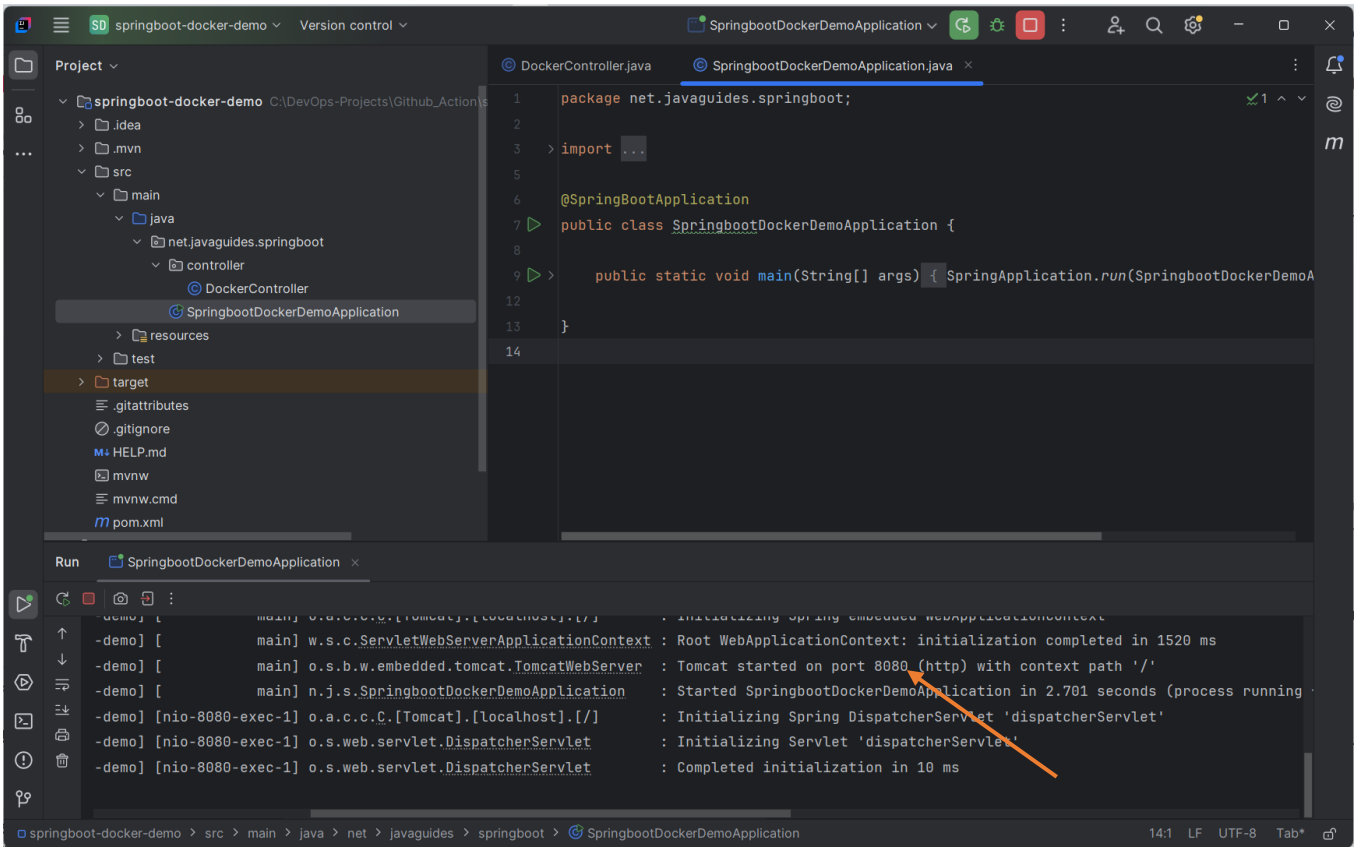
To do this, double-click on "**SpringbootDockerDemoApplication**"



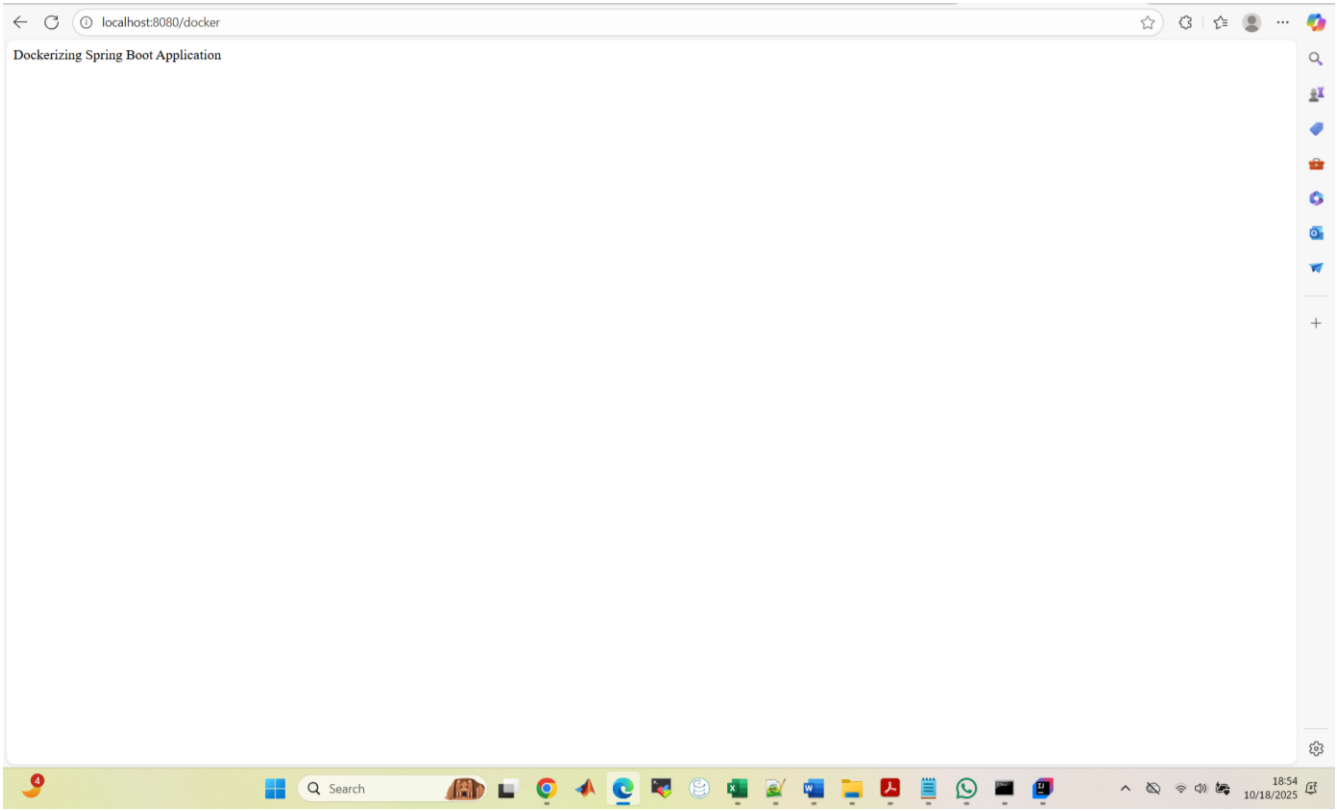
Click on "**Run**"



Select "**Run 'SpringbootDocker…main'**"

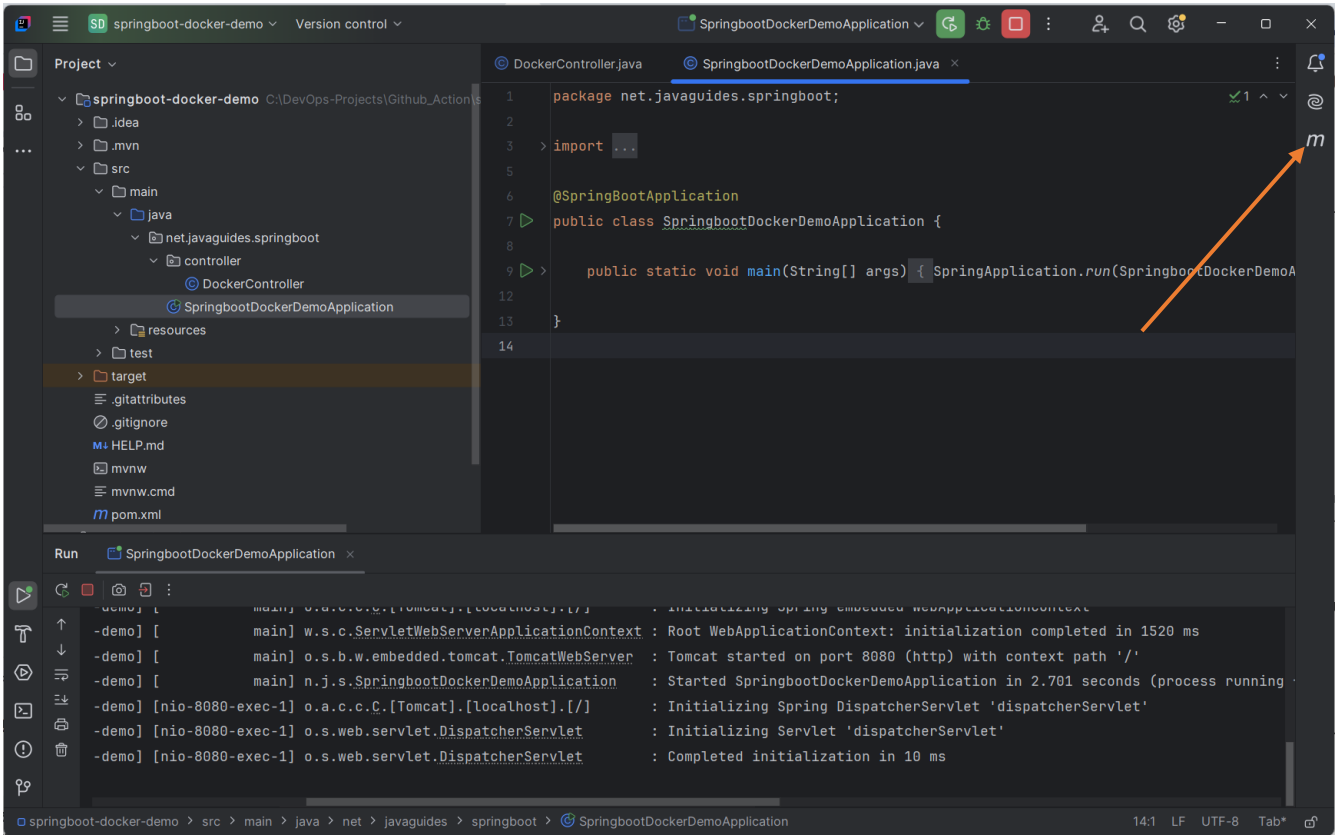


Now, verify on your browser by typing: **localhost:8080/docker**

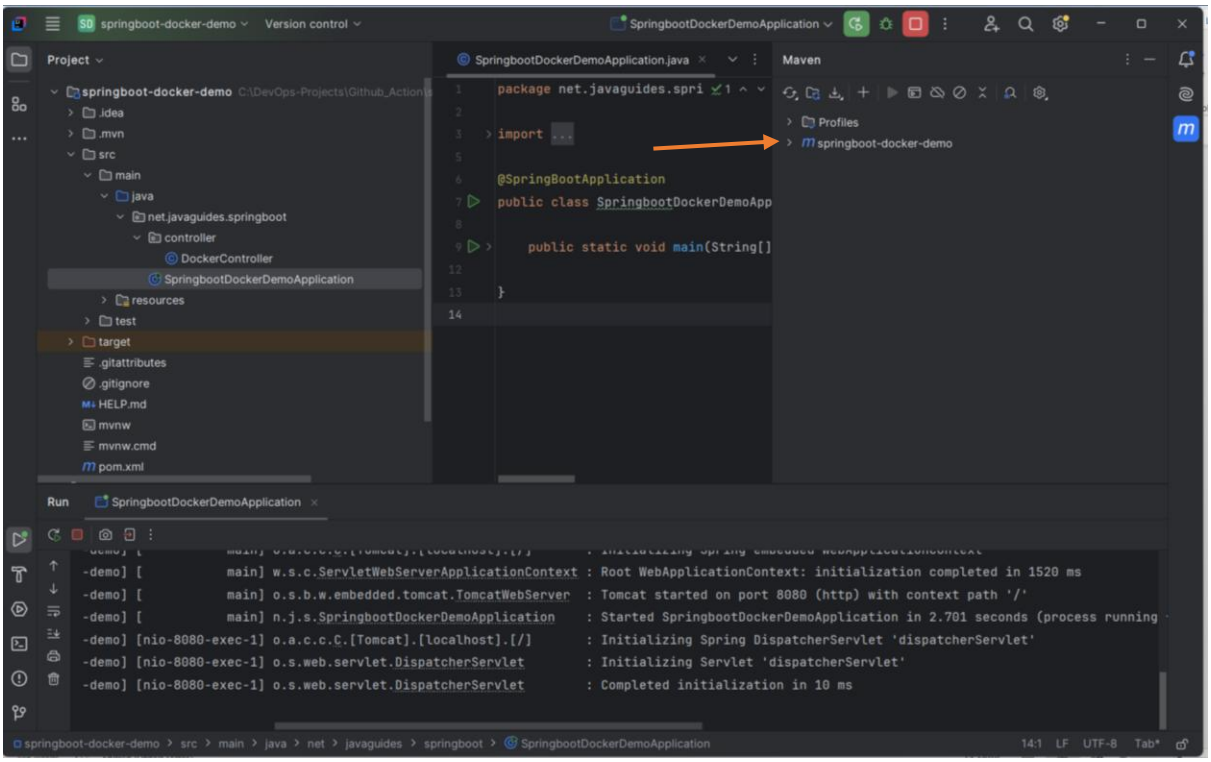


The application is working. We can now proceed to Dockerize the application.

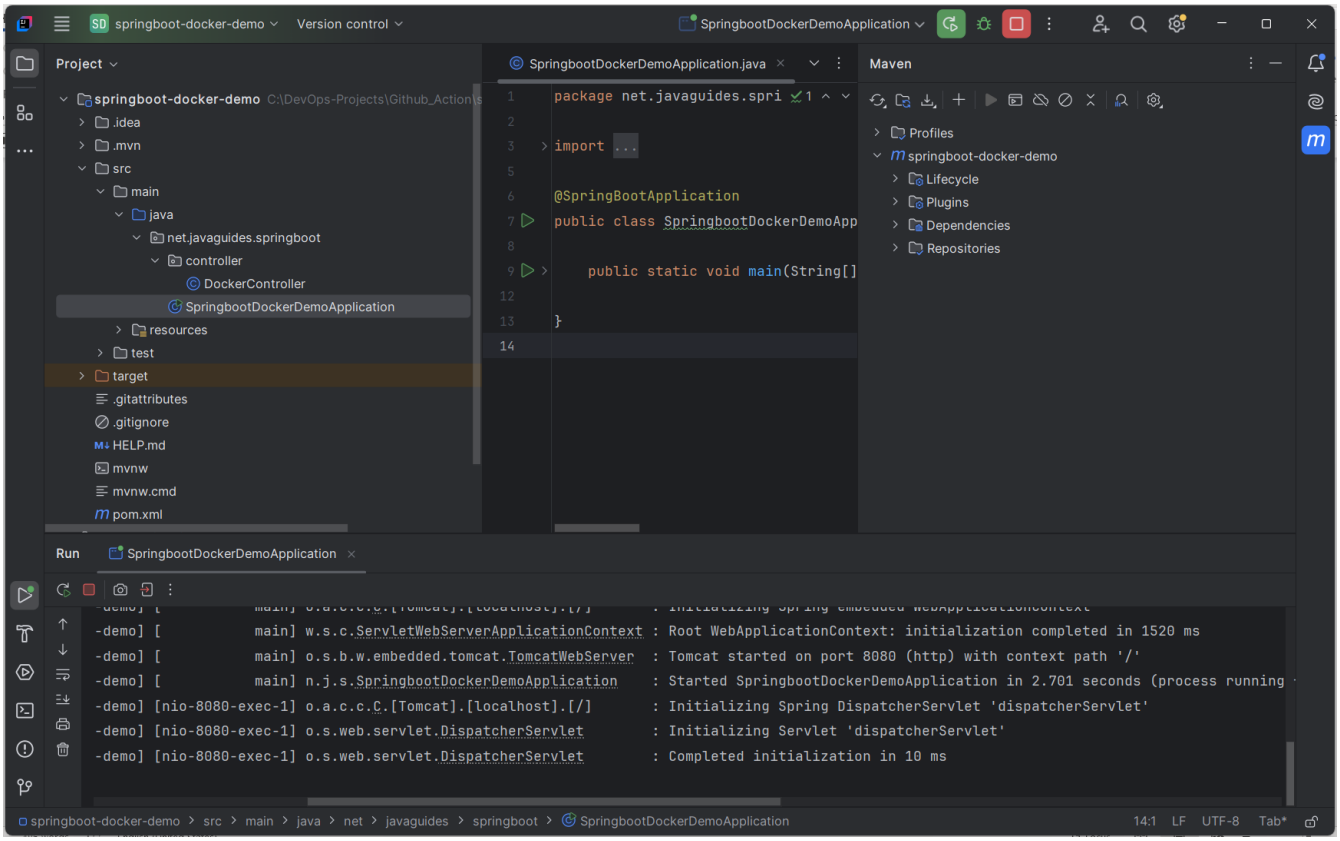
STEP 3: Build and Package Maven Application



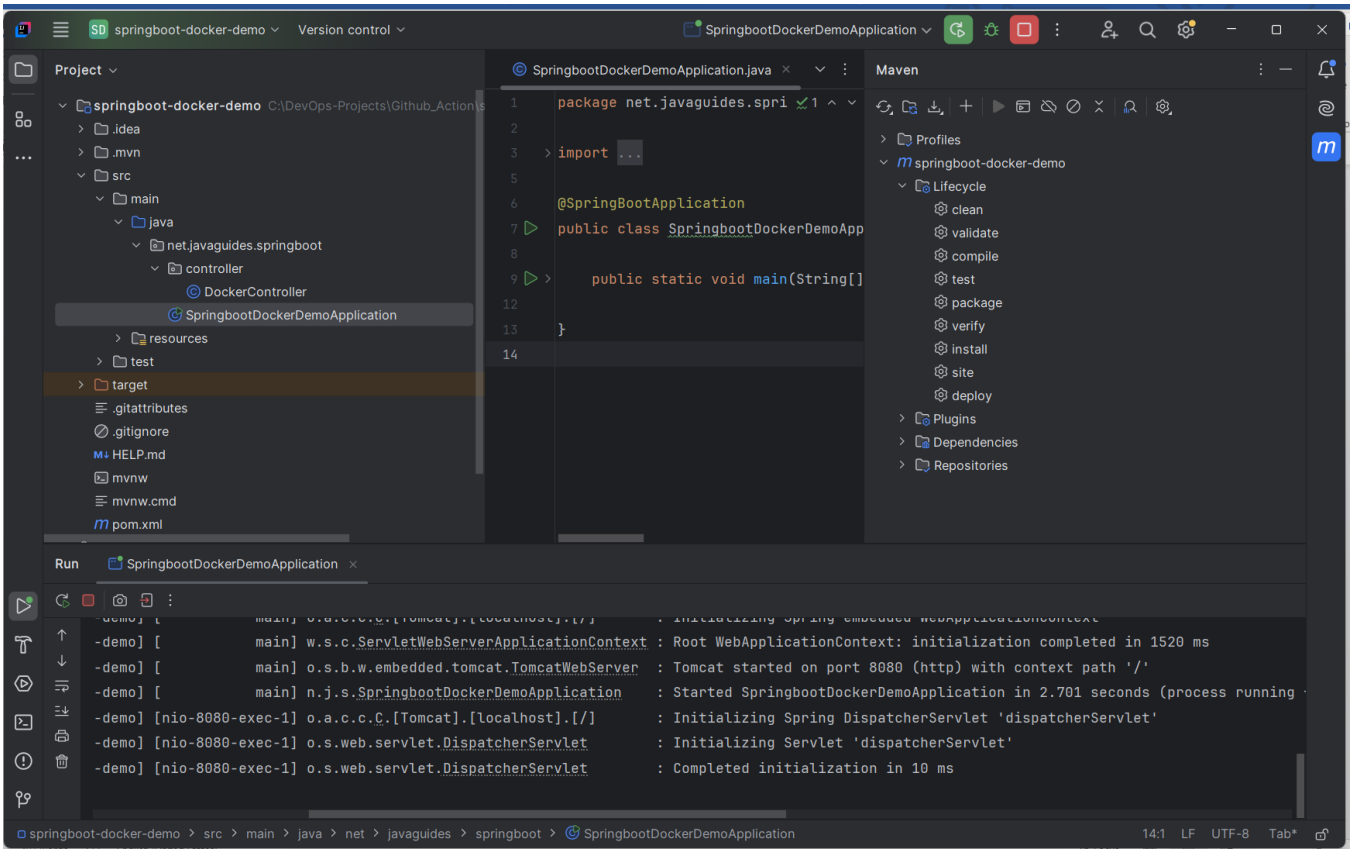
Click on "**Maven**" icon



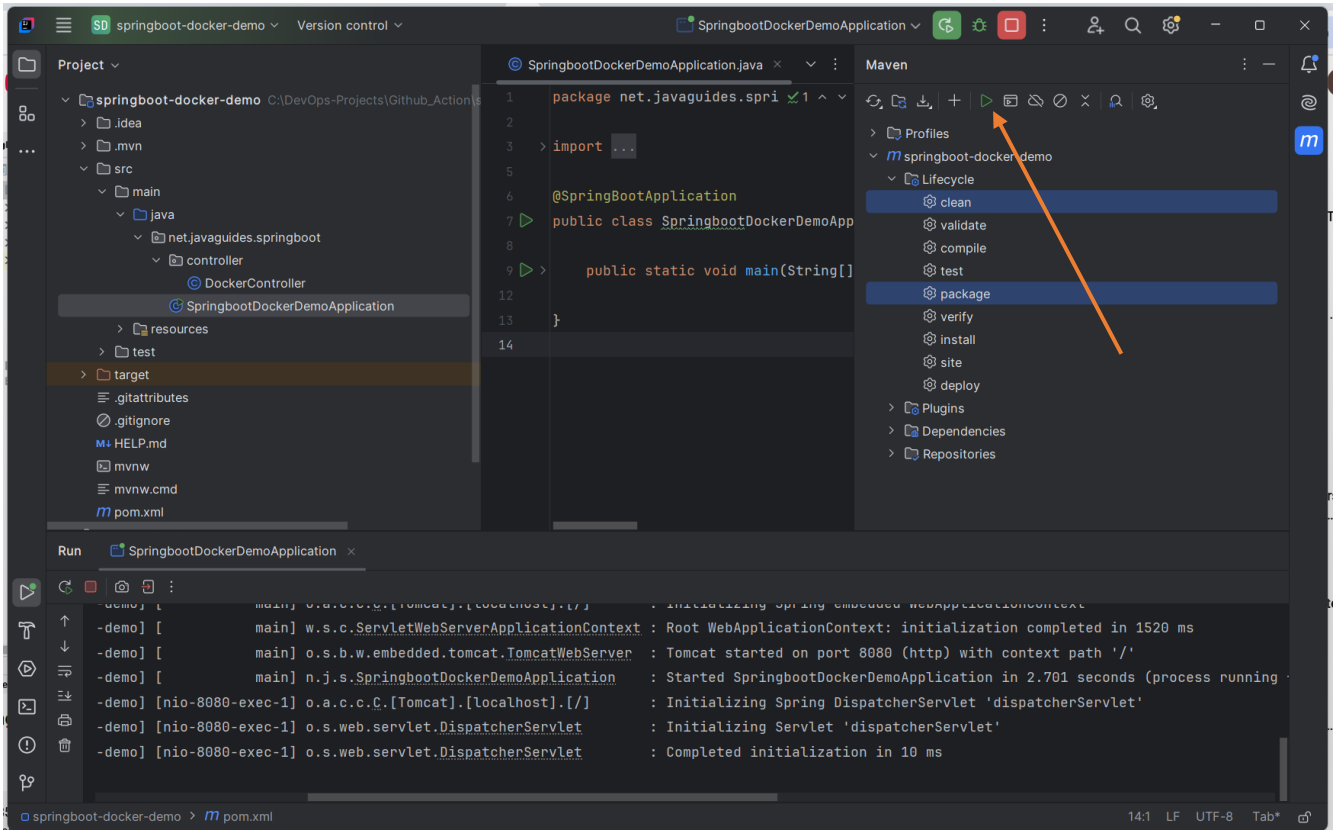
Click on "**springboot-docker-demo**"



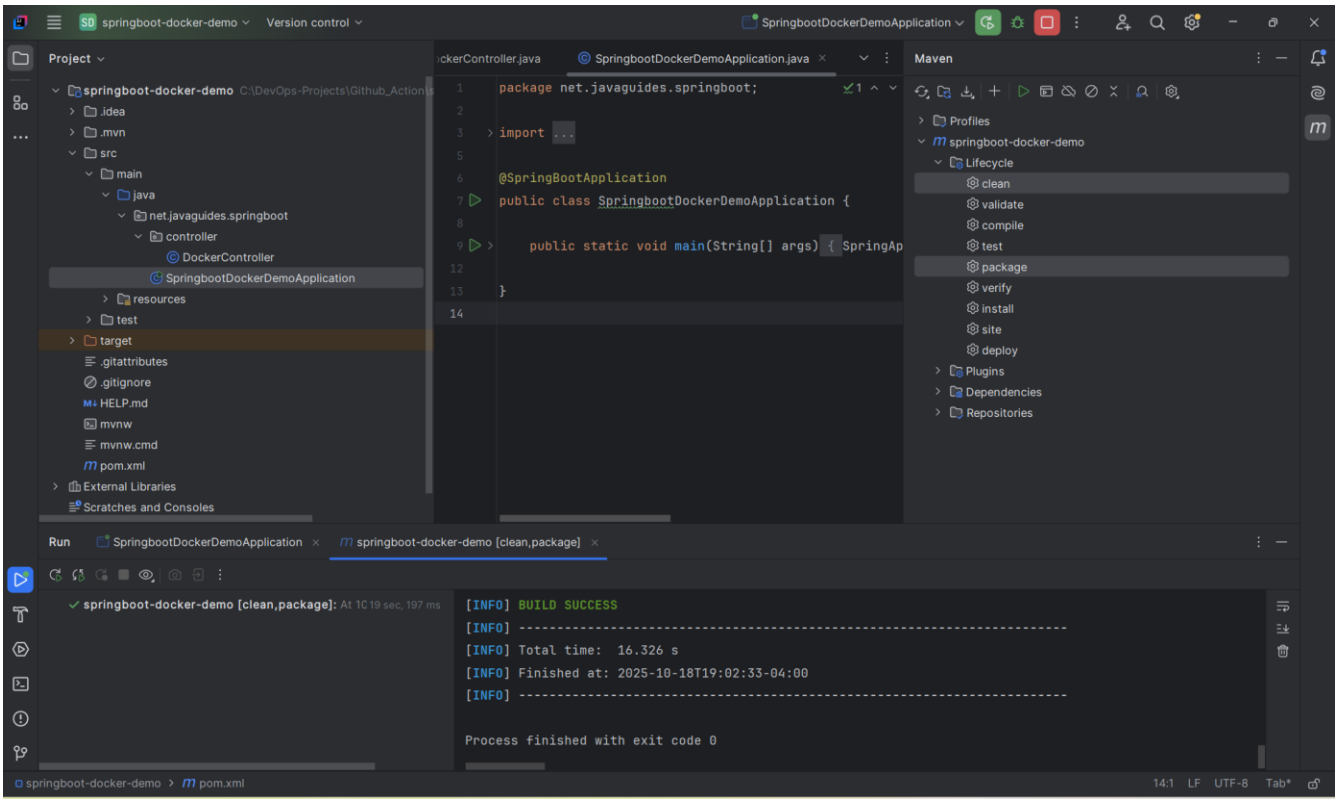
Select “**Lifecycle**”



Select “**clean**” and “**package**”

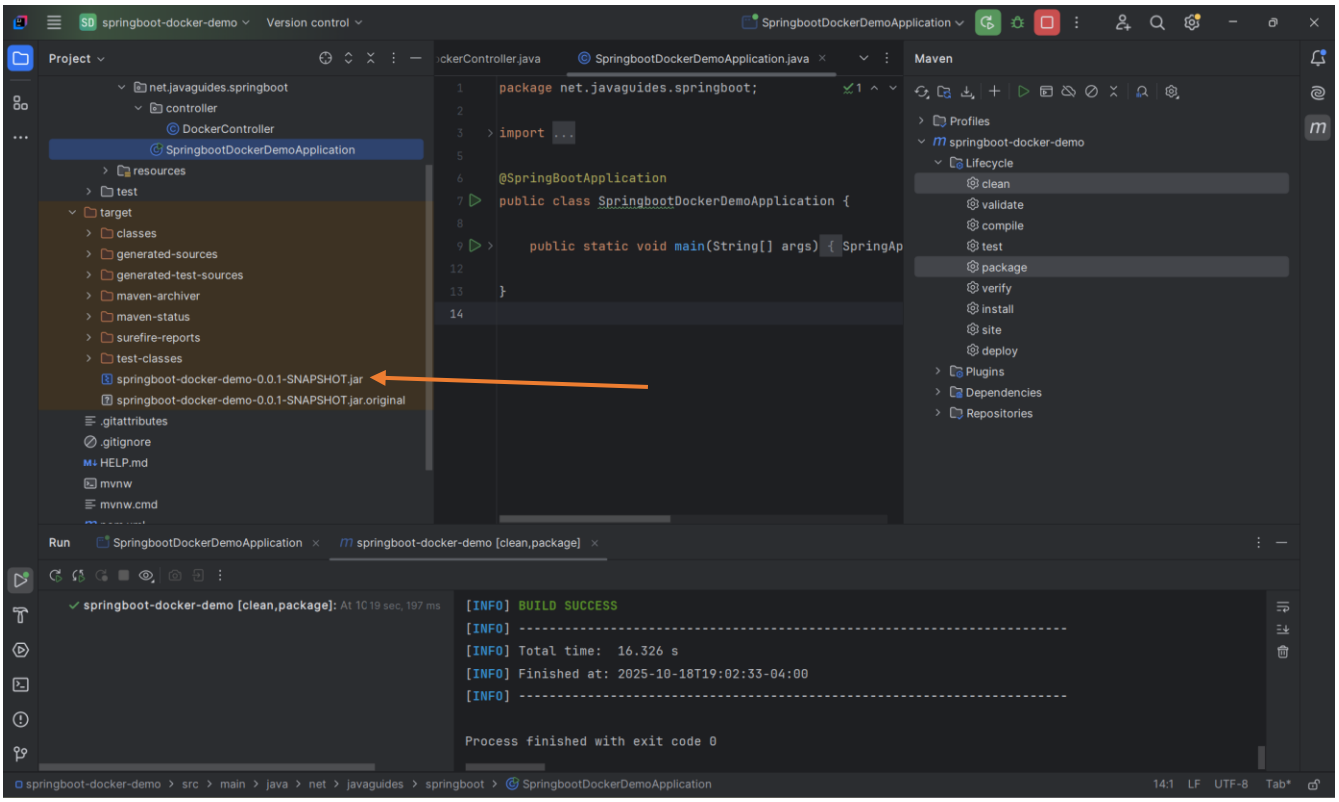


Click on “**Run Maven Build**”



The build is successful

To verify the Maven build is successful. Click on the "**target**" folder

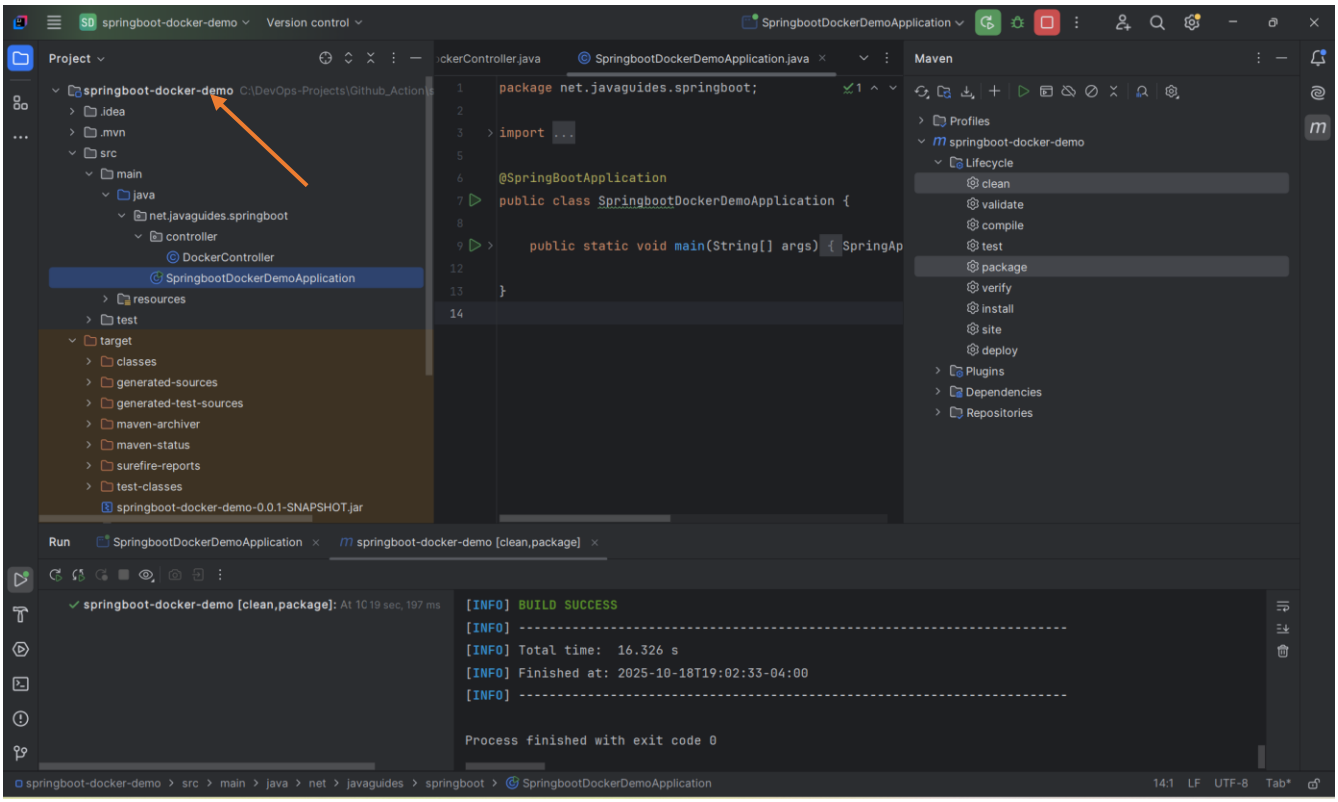


You can see that "**Springboot-docker-demo-0.0.1-SNAPSHOT.jar**" has been created.

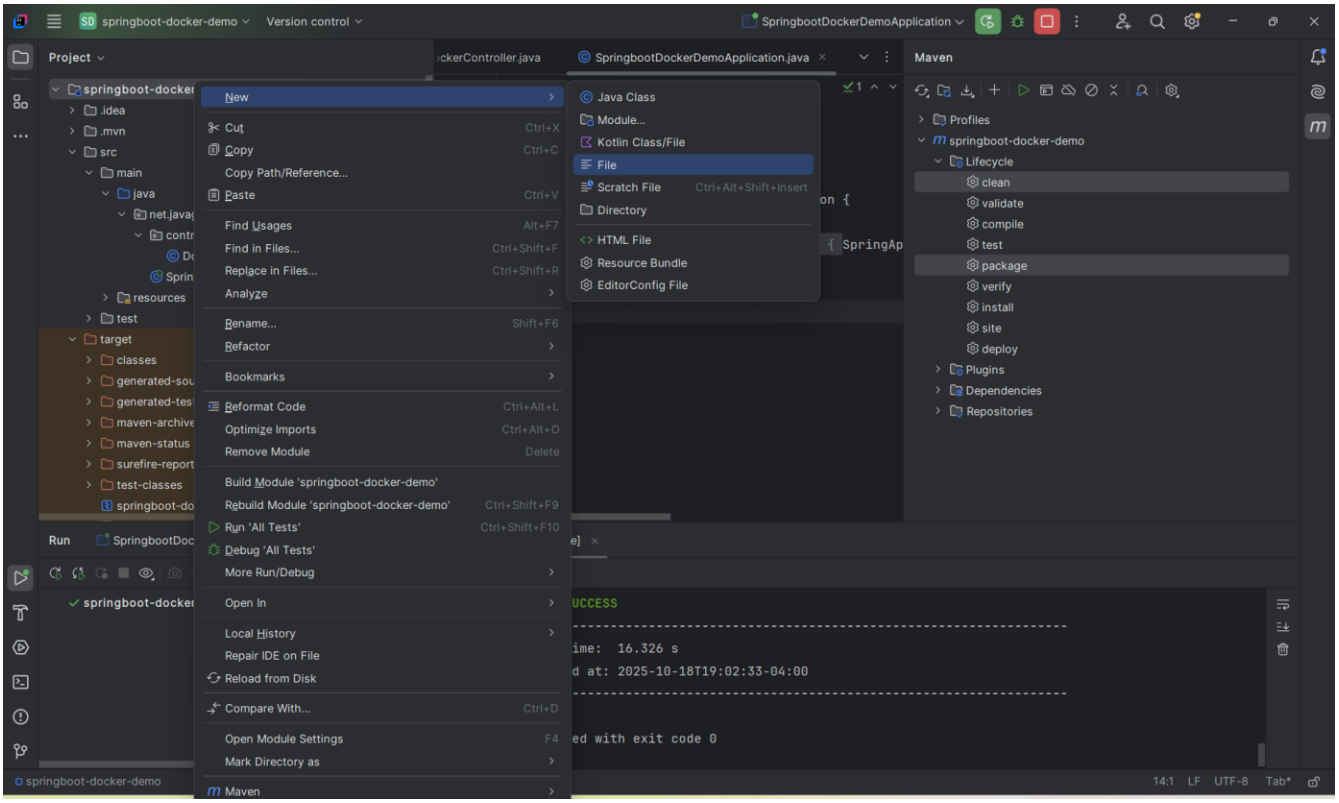
STEP 4: Create a Dockerfile

Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. This page describes the commands you can use in a Dockerfile.

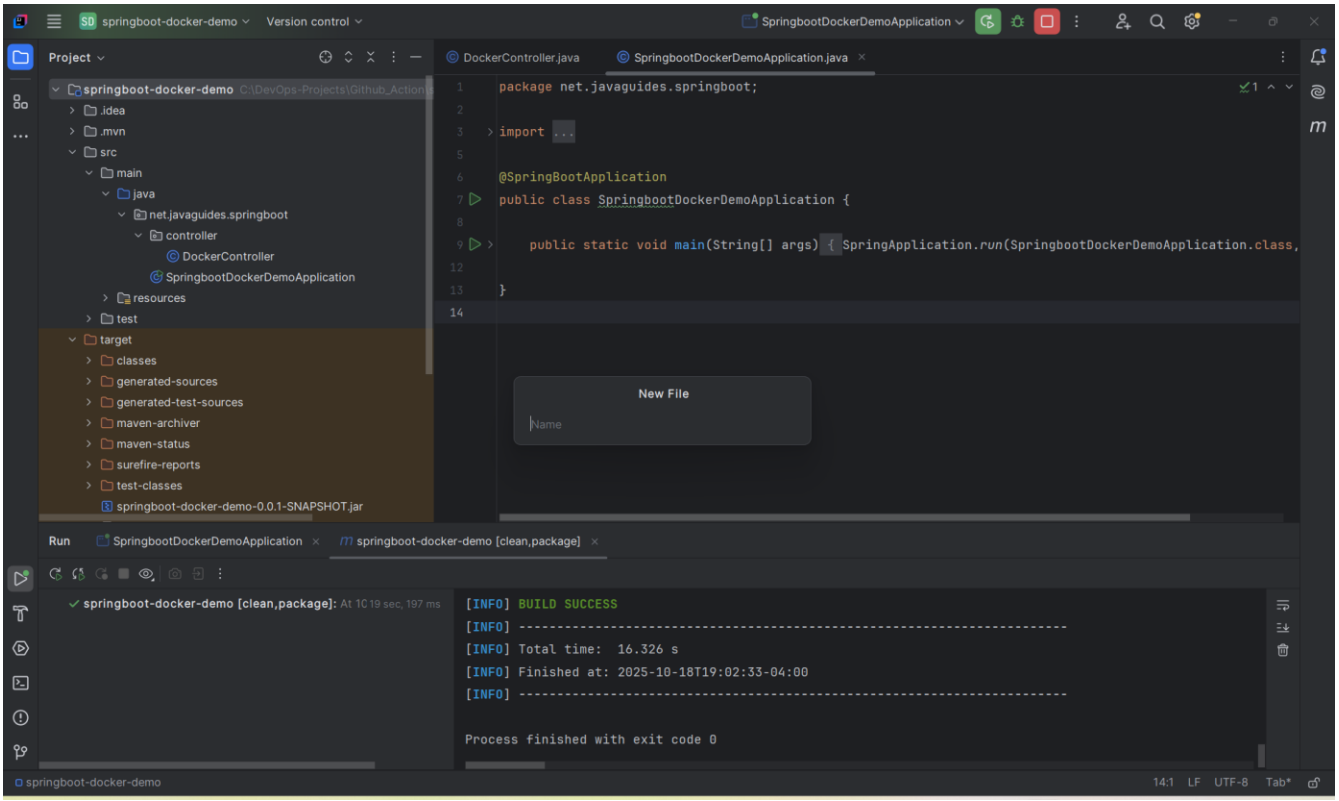
We will create a Dockerfile where we will define all the instructions and commands to build the Docker Image.



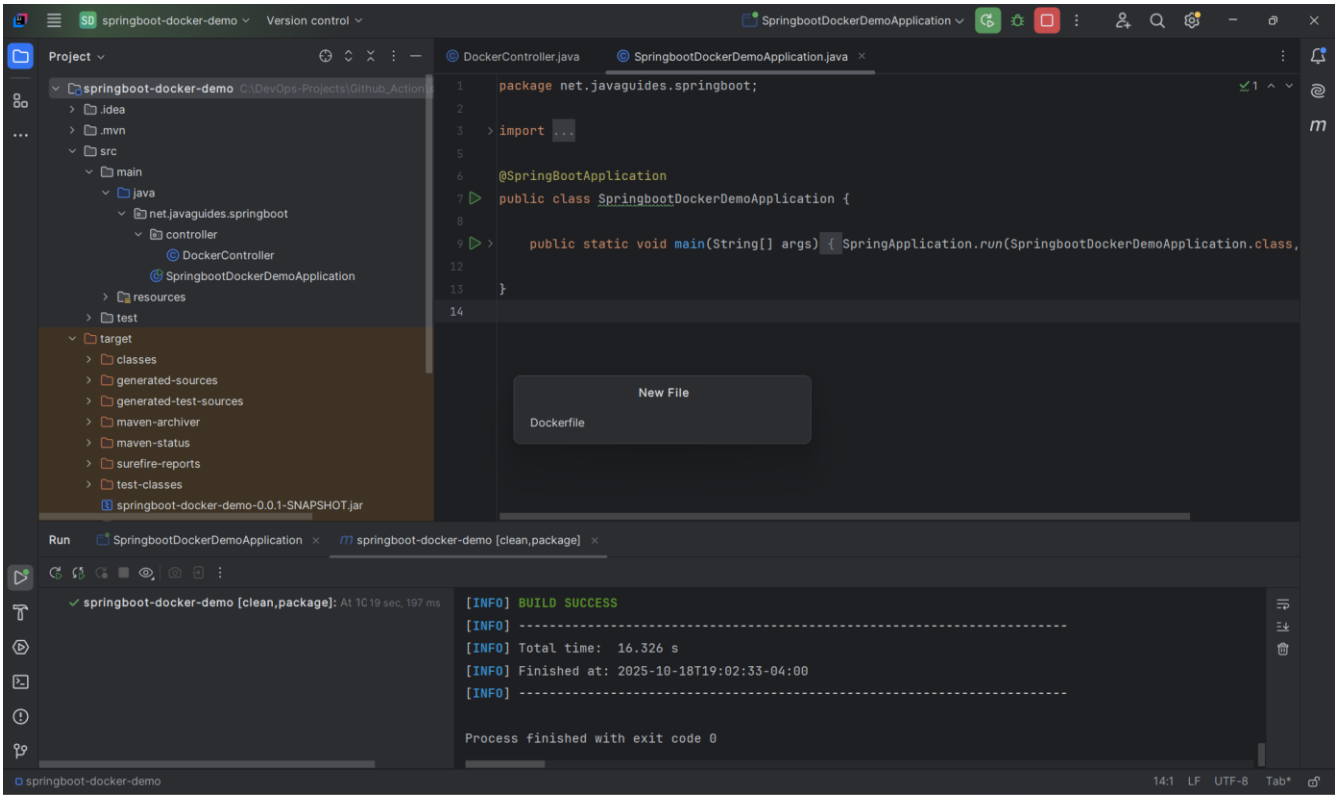
Right-click on the project name "**springboot-docker-demo**"



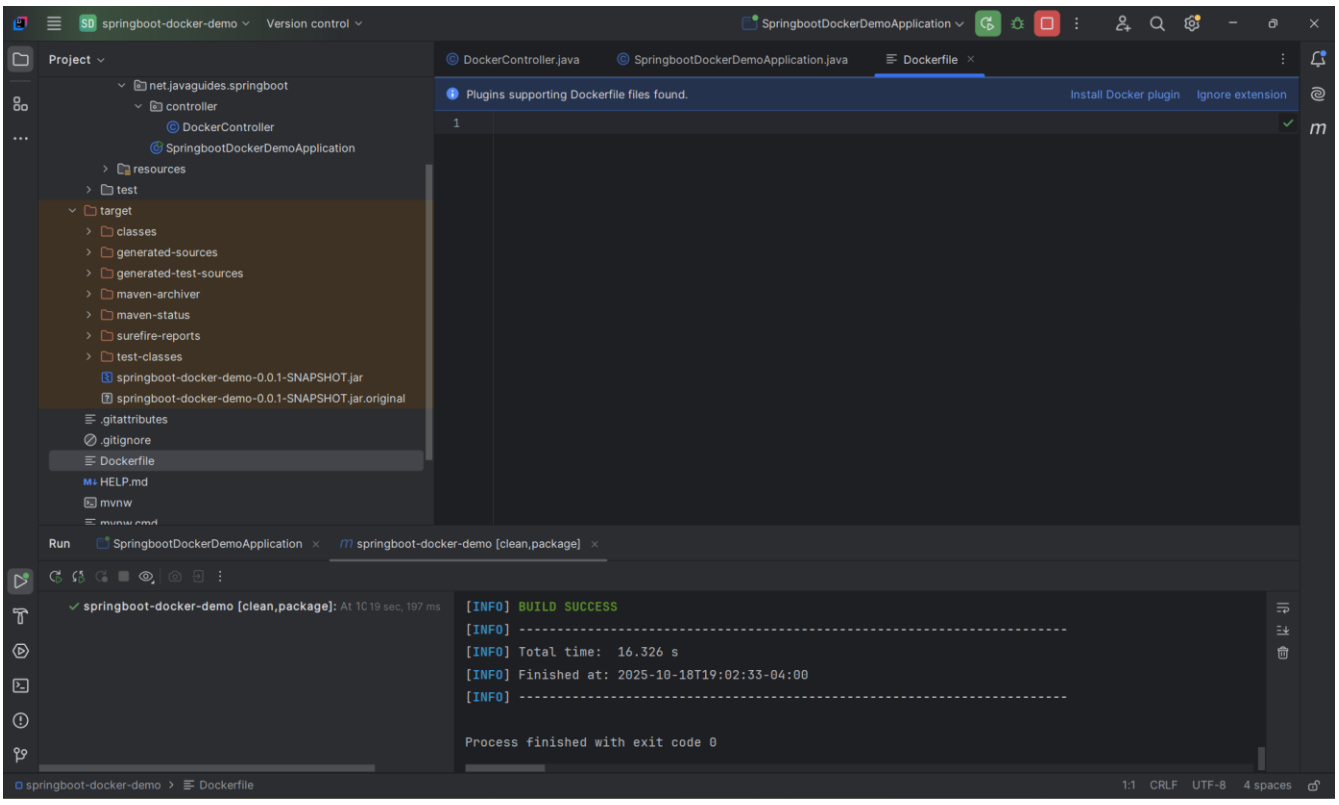
Select "**File**"



Enter the name of the file as "**Dockerfile**"



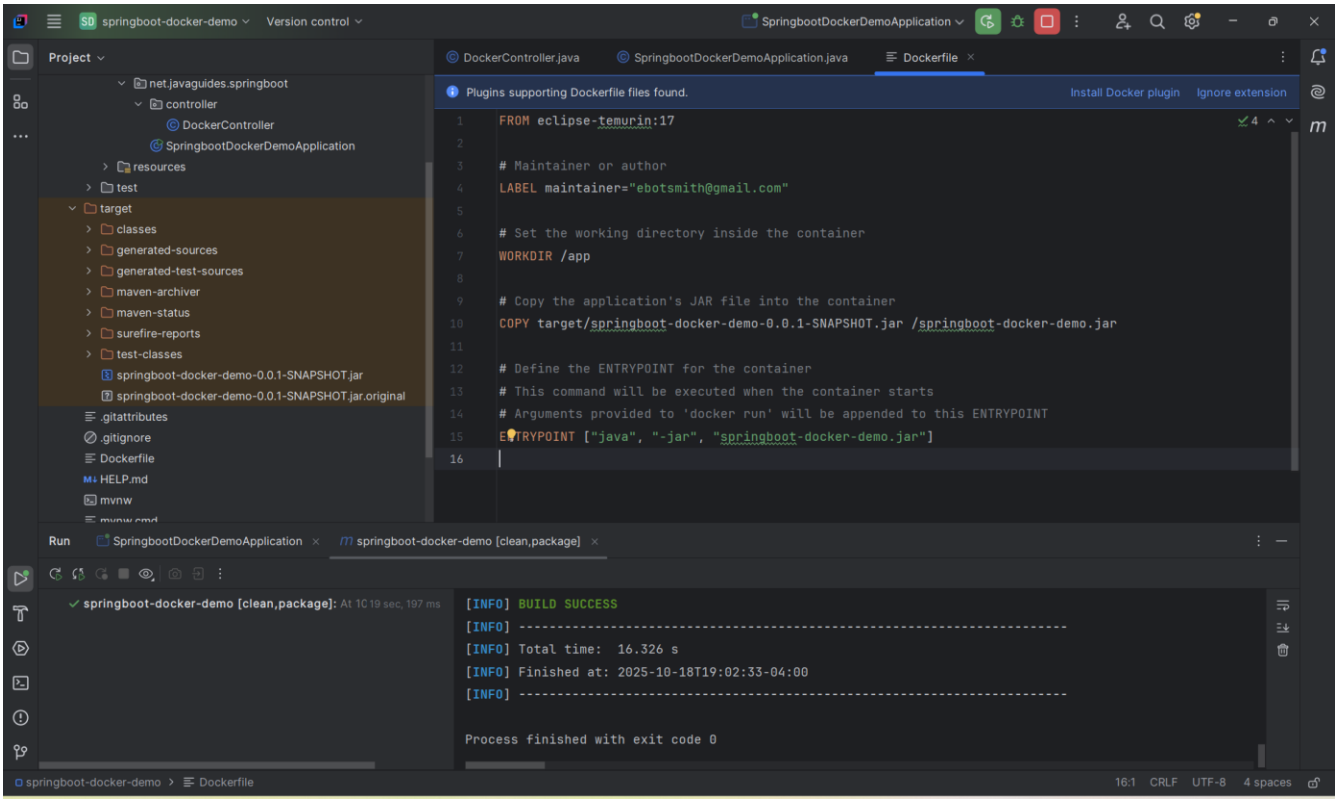
Press "**Enter**"



Let us now write the commands on our Dockerfile

```
# Use an official Eclipse Temurin base image
```

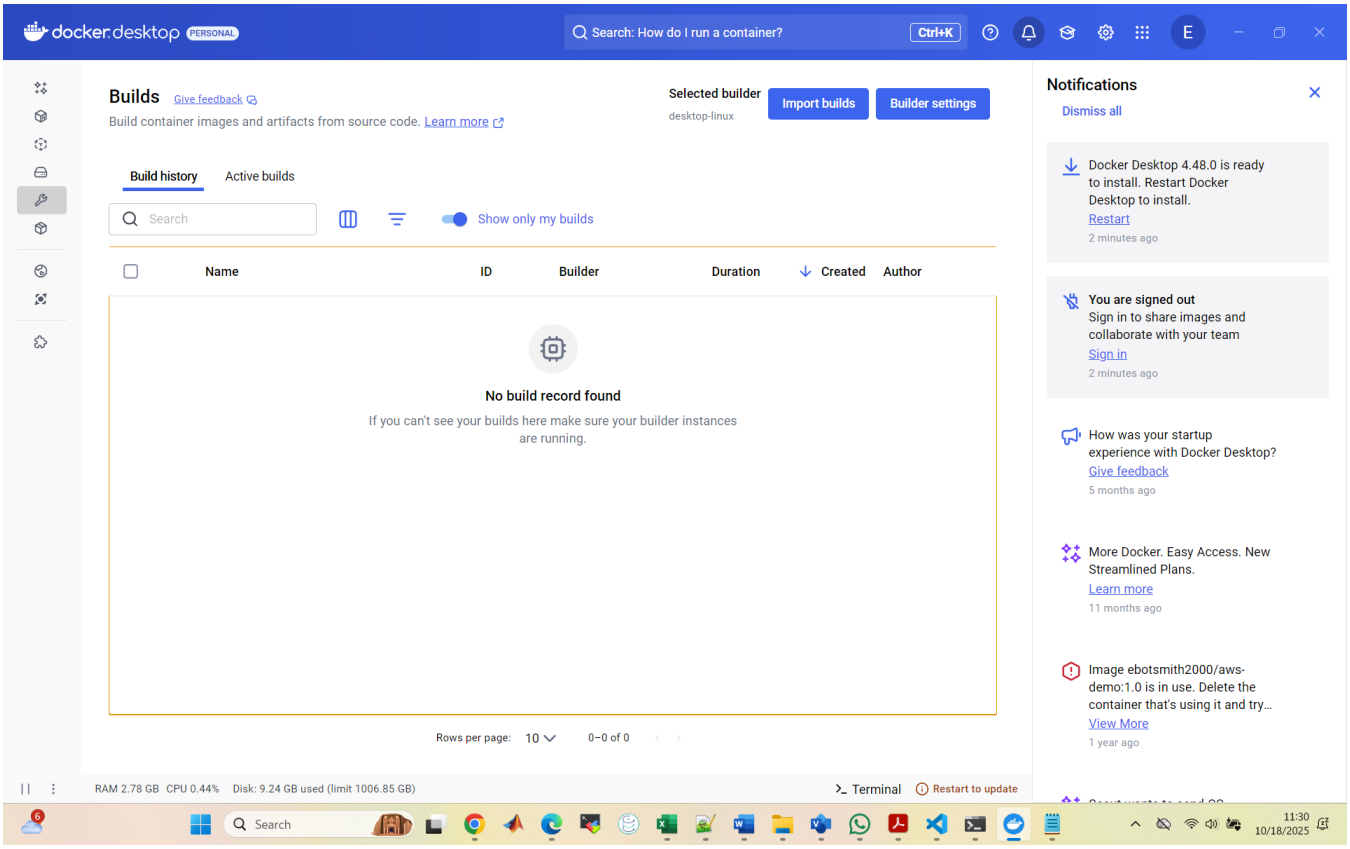
```
FROM eclipse-temurin:17

# Maintainer or author
LABEL maintainer="ebotsmith@gmail.com"

# Set the working directory inside the container
WORKDIR /app

# Copy the application's JAR file into the container
COPY /target/springboot-docker-demo-0.0.1-SNAPSHOT.jar /app/springboot-docker-demo.jar

# Define the ENTRYPOINT for the container. This command will be executed when the container starts
# Arguments provided to 'docker run' will be appended to this ENTRYPOINT
ENTRYPOINT ["java", "-jar", "/app/springboot-docker-demo.jar"]
```



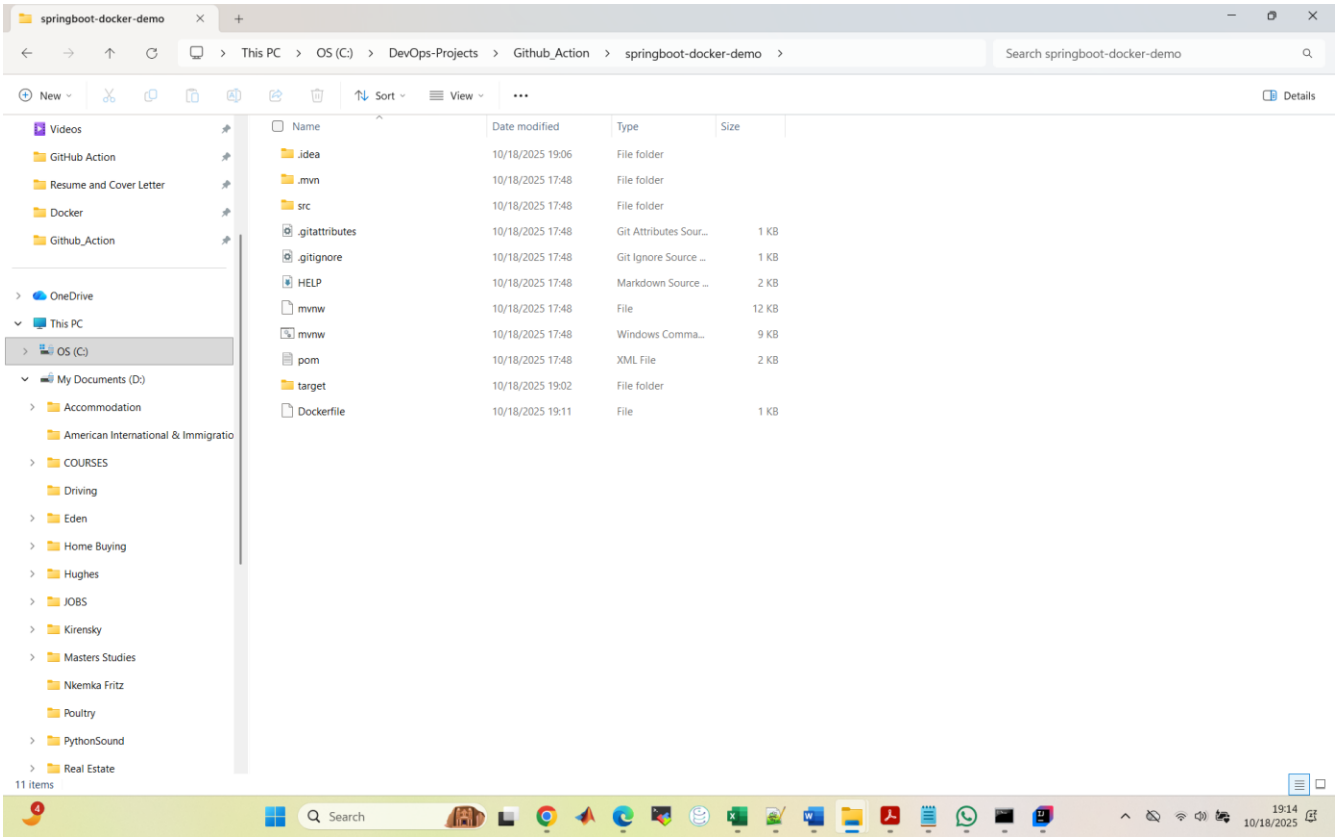
Now, let us save the file.

STEP 5: Build Docker Image from Dockerfile

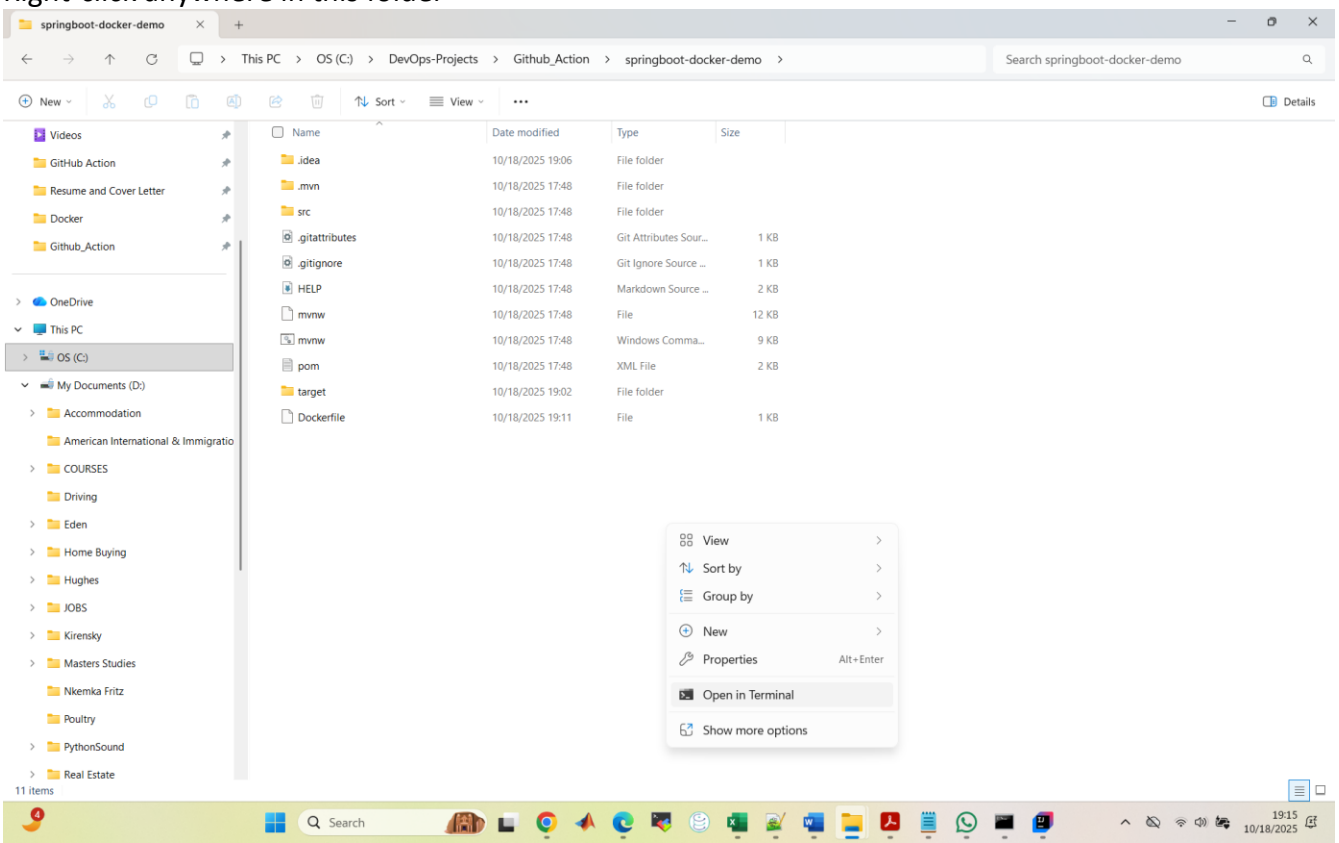
We will now build a Docker Image from the docker file. Before doing this, make sure your Docker desktop is up and running on your local machine.



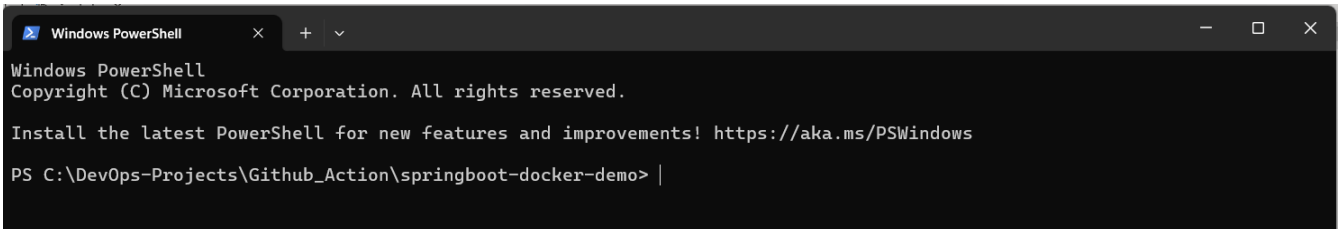
Head back to the project folder in your local machine



Right-click anywhere in this folder

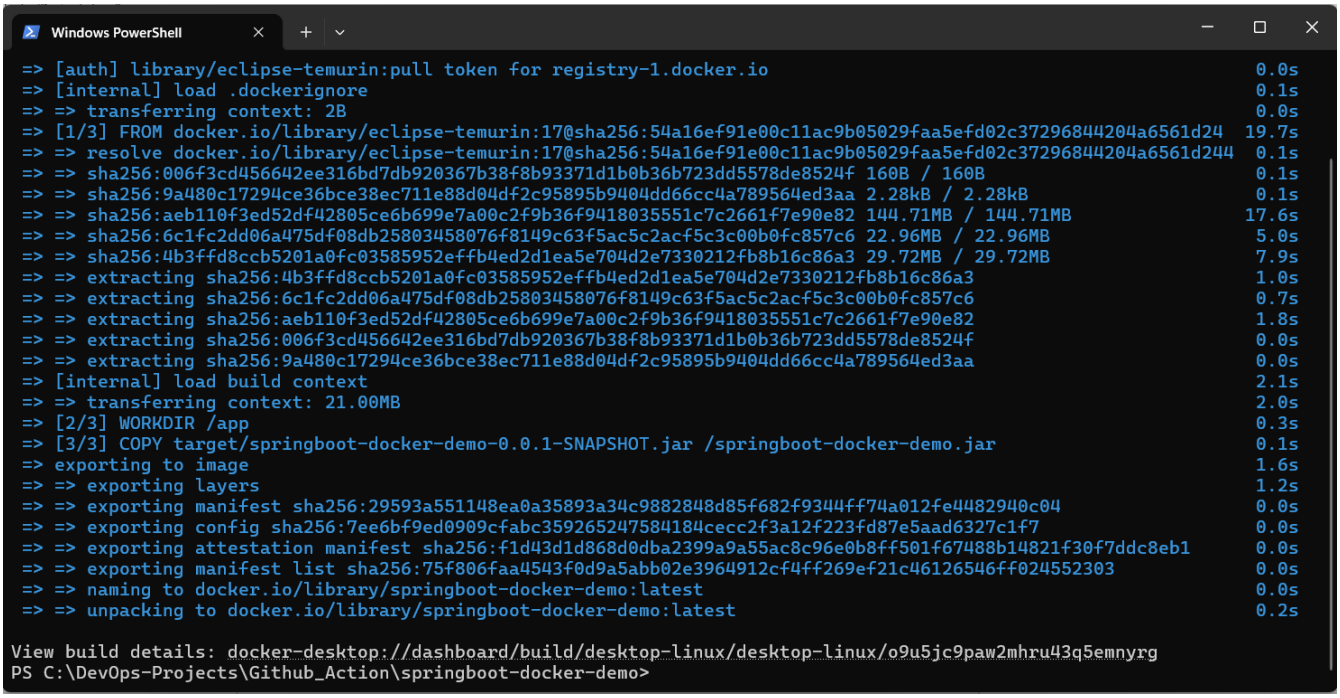


Select "**Open in Terminal**"

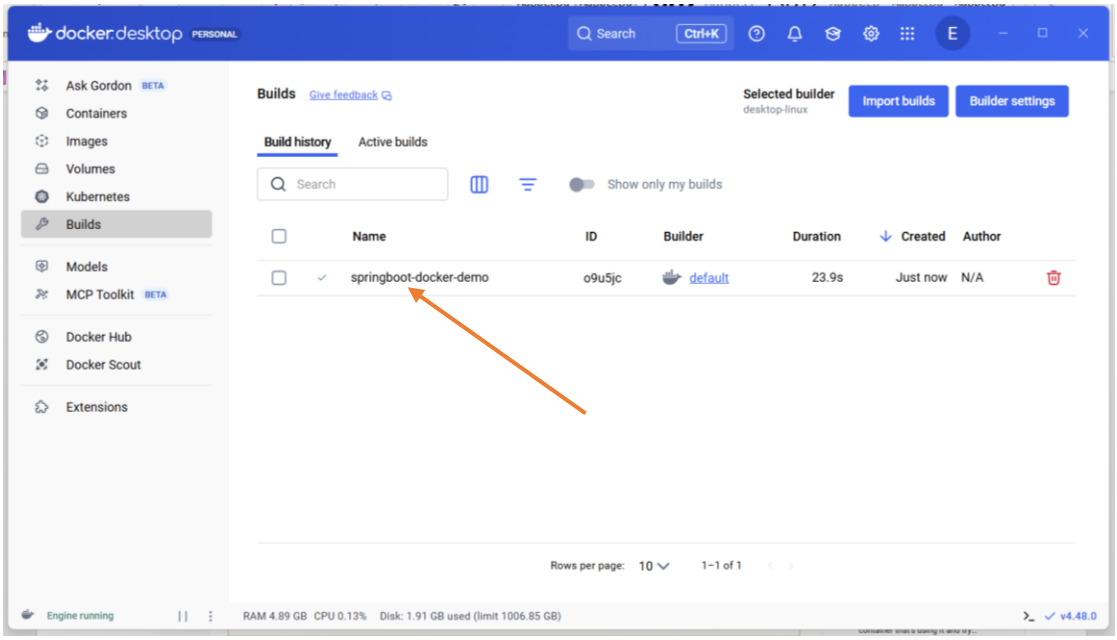


Run the command to build the Docker image with the name "**springboot-docker-demo**"

```
docker build -t springboot-docker-demo .
```

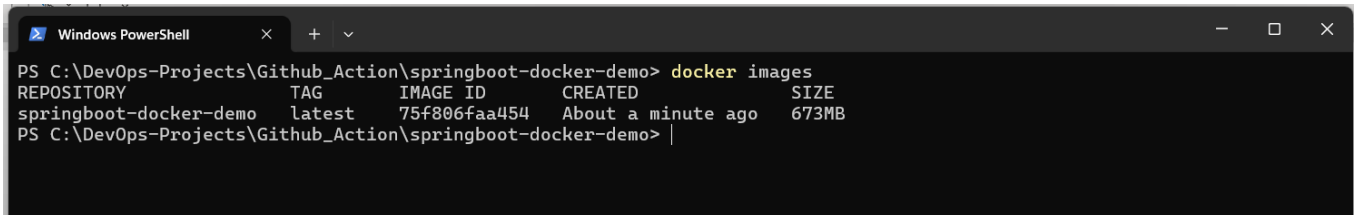


Head back to the Docker desktop



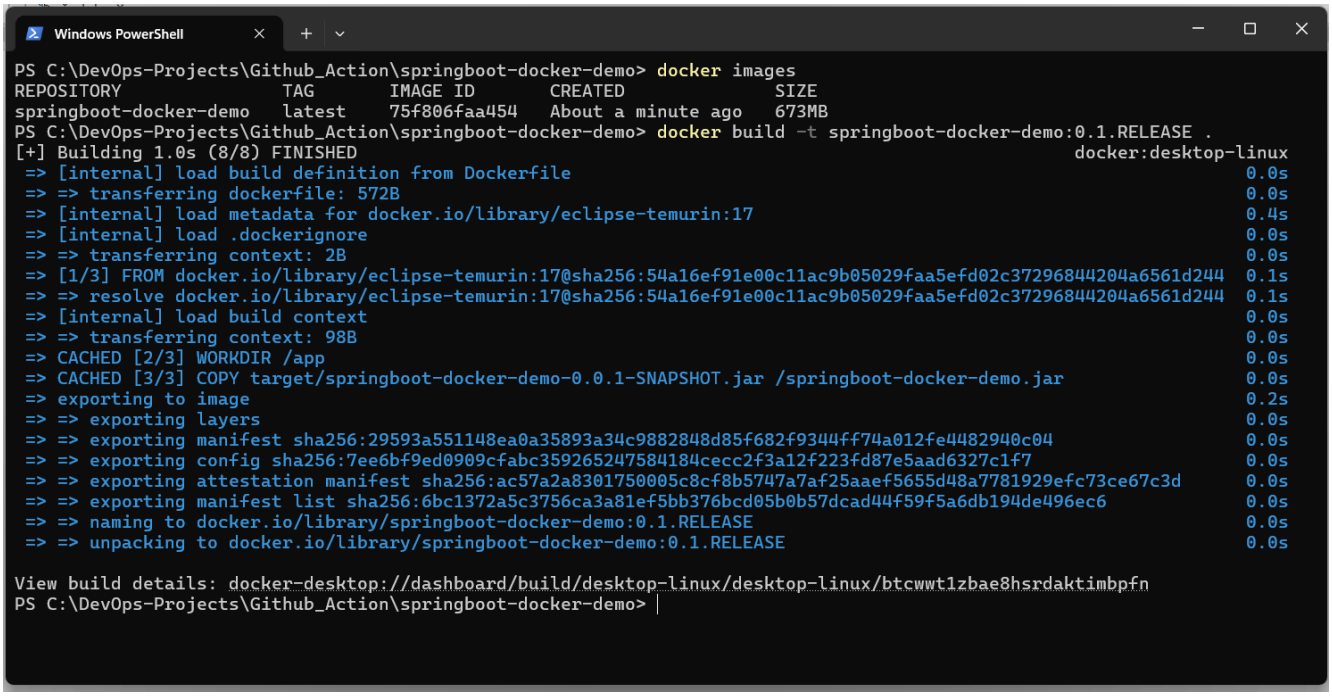
You can see the image has been built. Let us verify using the command:

```
docker images
```



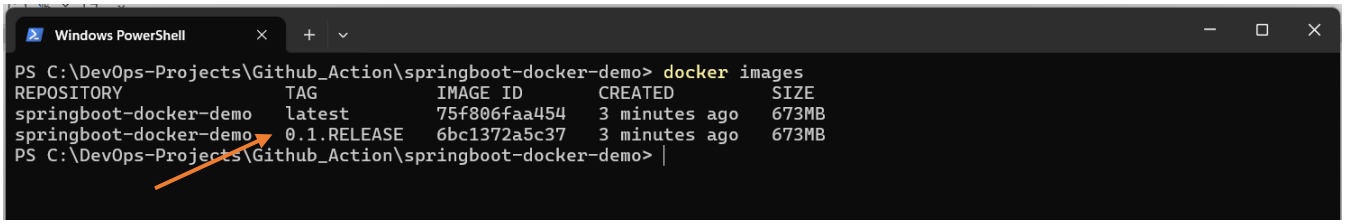
You can see our image. We can also add a tag to the image. We can add a tag name “0.1.RELEASE” by using the command:

```
docker build -t springboot-docker-demo:0.1.RELEASE .
```



Let us check the list of docker images again using the command:

```
docker images
```



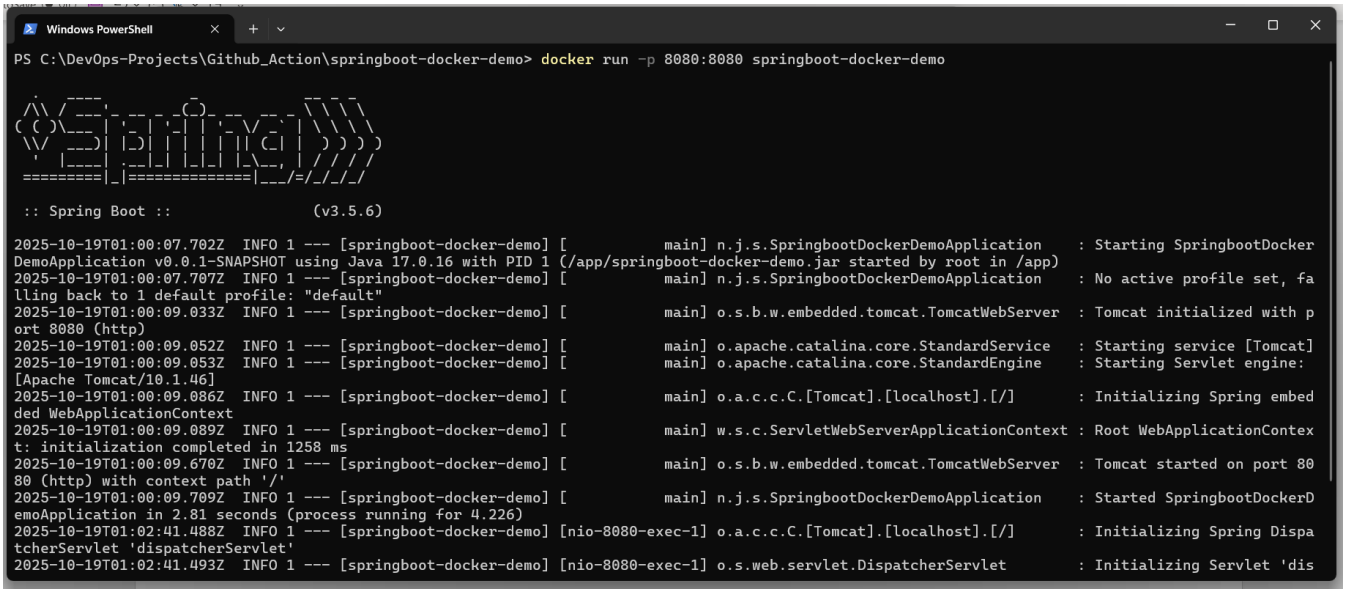
You can see the tag.

STEP 6: Run Docker Image in a Docker Container

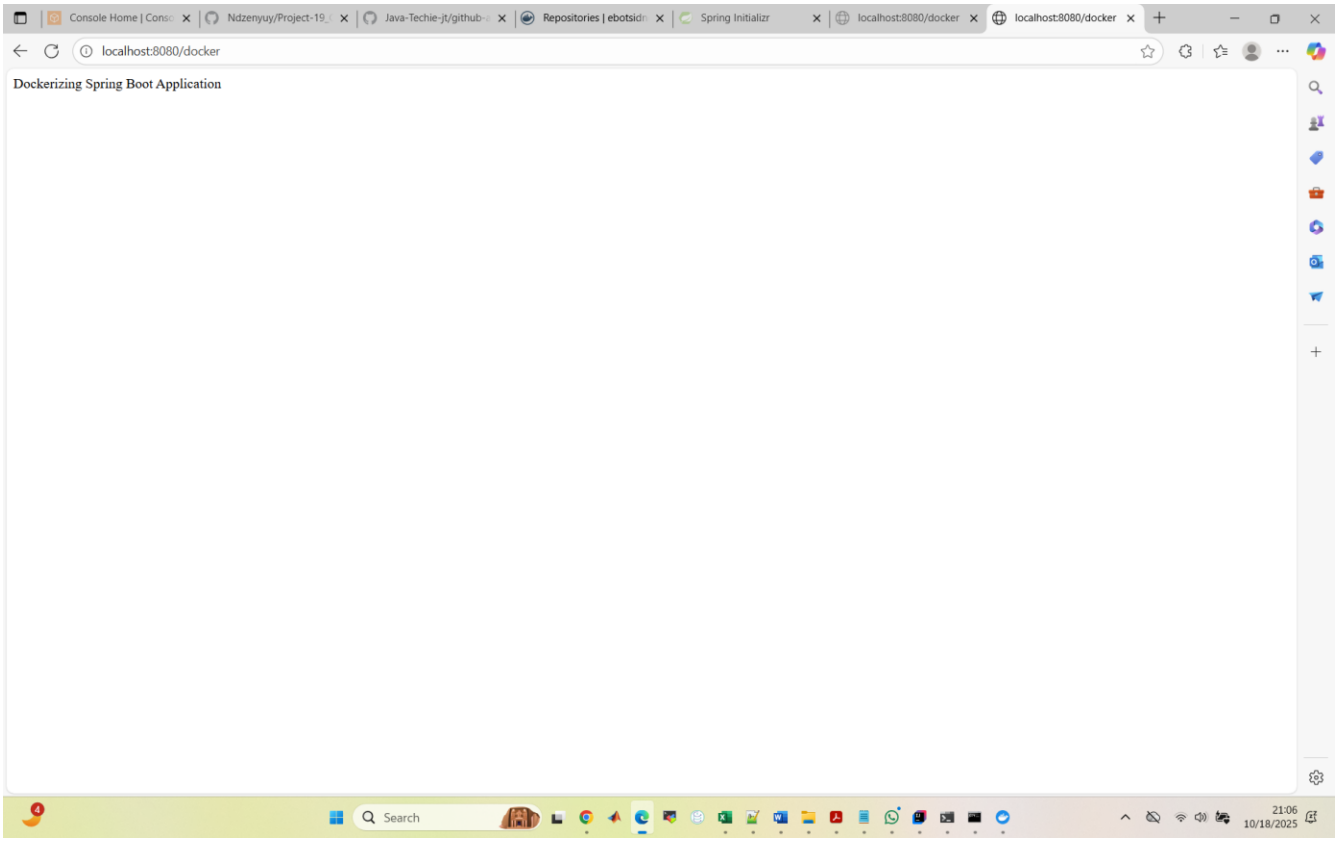
We will run the Docker Image in a Docker container. This will be done using the command:

```
docker run -p 8080:8080 springboot-docker-demo
```

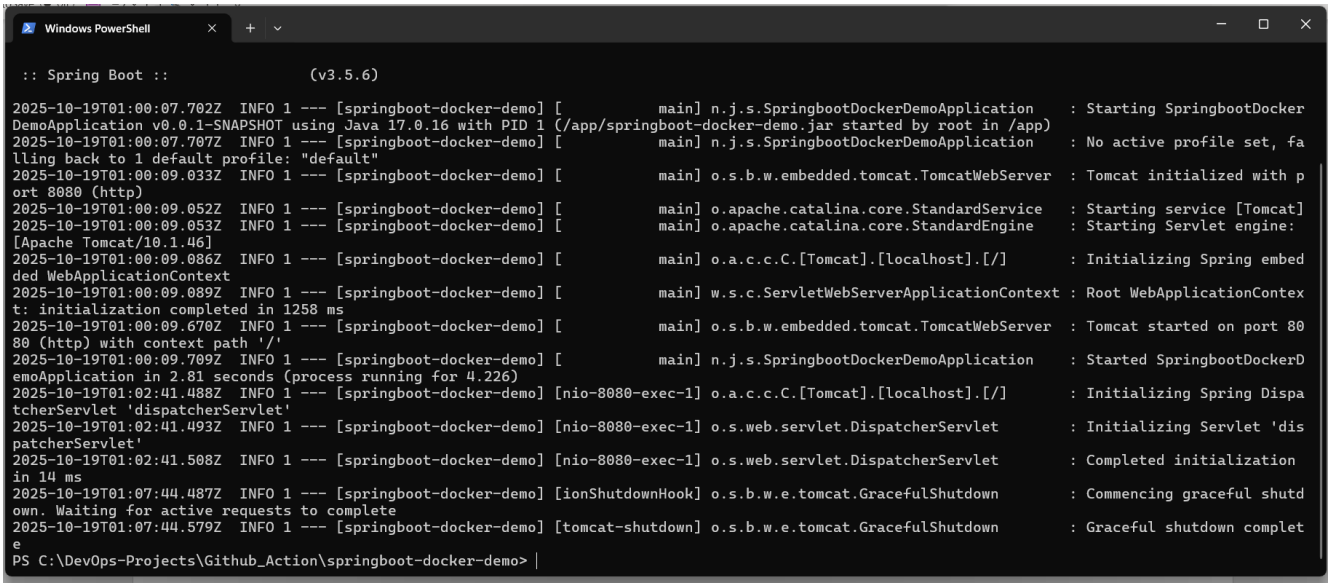
The -p is used to map the operating system port 8080 to the docker container port 8080. The image name is "**springboot-docker-demo**"



Now, let us go to our browser and test using **localhost:8080/docker**

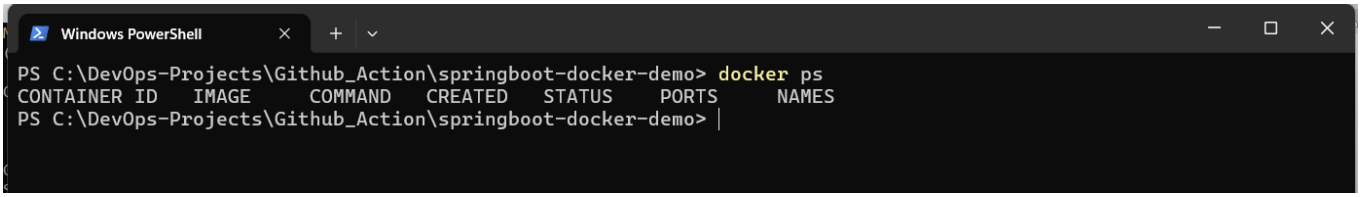


The application is running on our container. You can go back to the command mode in the terminal using "**ctrl + c**". This will also stop the docker container



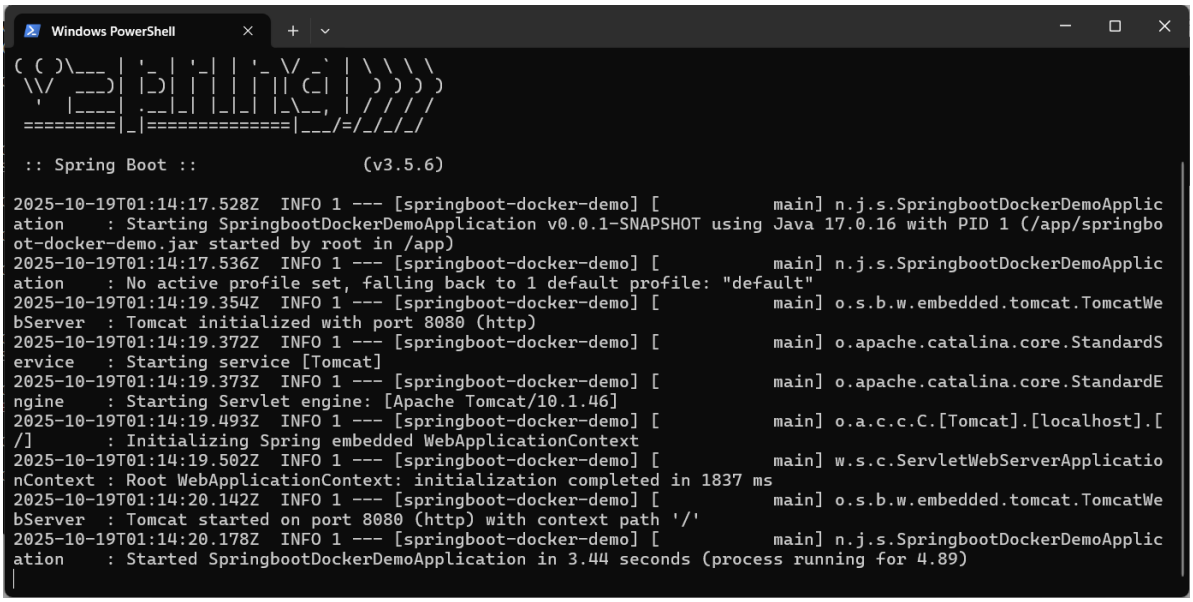
Run the command to check if any container is running:

```
docker ps
```

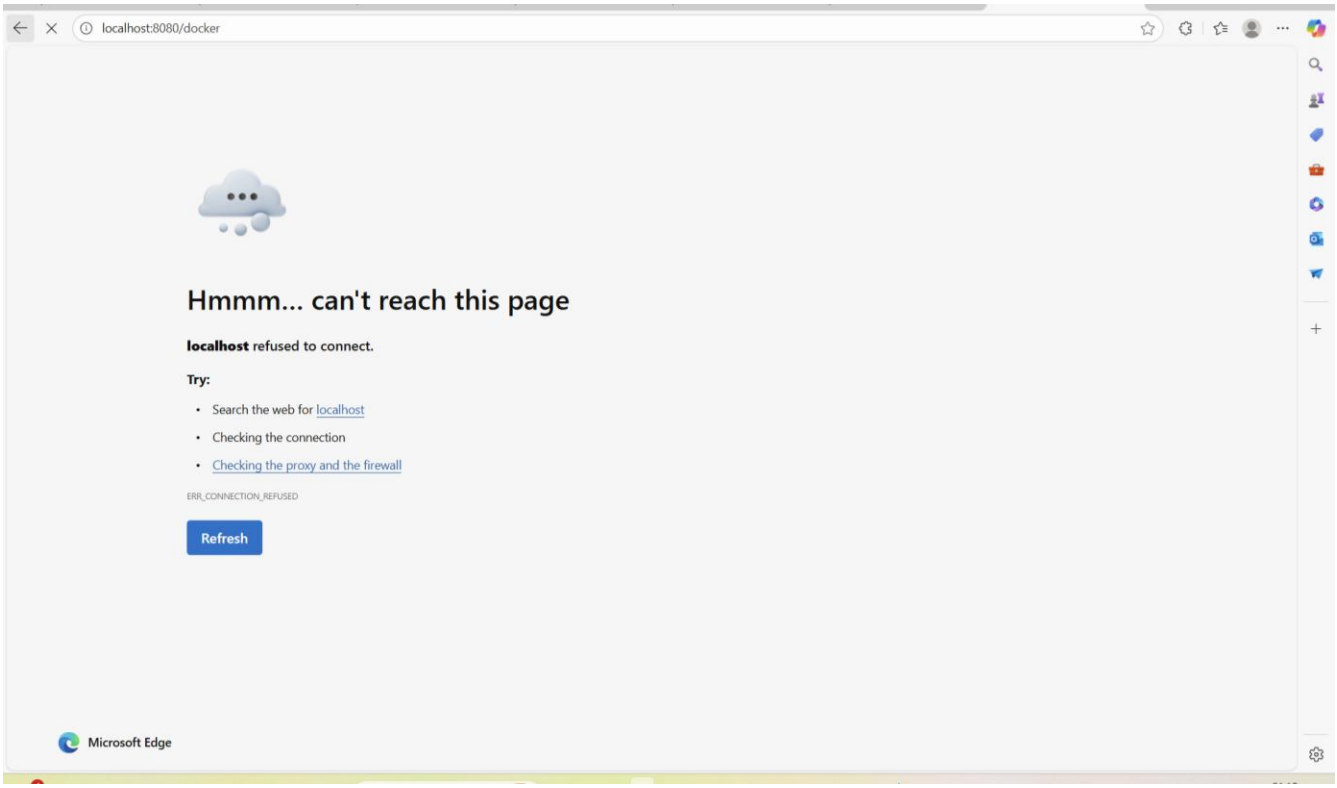


You can see that no container is running. Let us try to run the container on another port now. We will use port 8081. So, our command will be

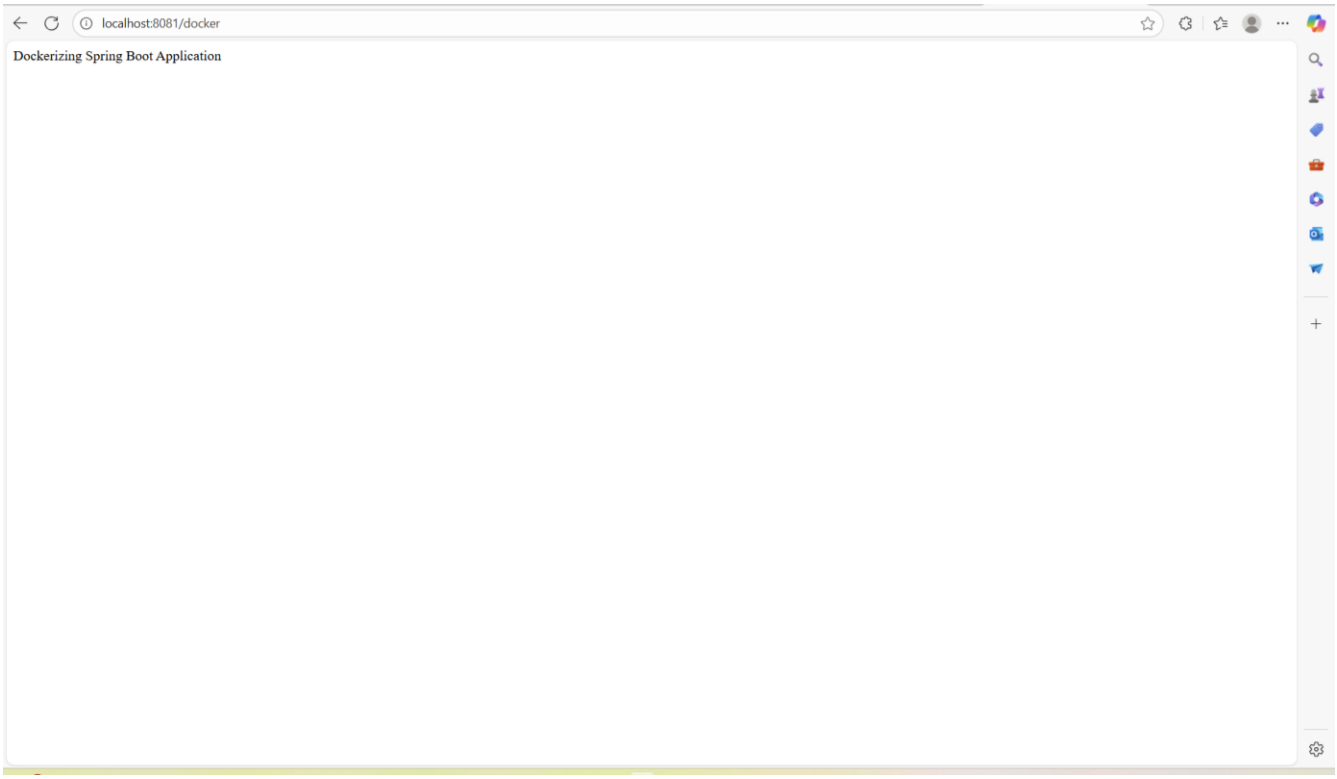
```
docker run -p 8081:8080 springboot-docker-demo
```



Let us try to test our application on port 8080 again using localhost:8080/docker



You can see that the application is not running on port 8080 anymore. Let us try to test it on port 8081 now using `localhost:8081/docker`

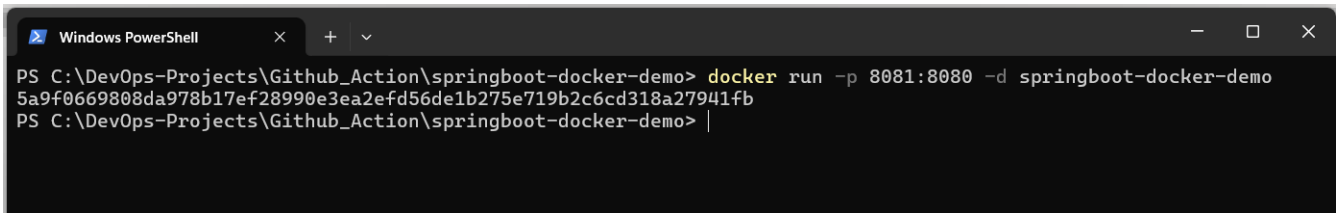


You can see that the application is now running on Port 8081.

STEP 7: Run the Docker container in Detached mode

Running the docker container in detached mode means running it in the background. To do this we run the command:

```
docker run -p 8081:8080 -d springboot-docker-demo
```

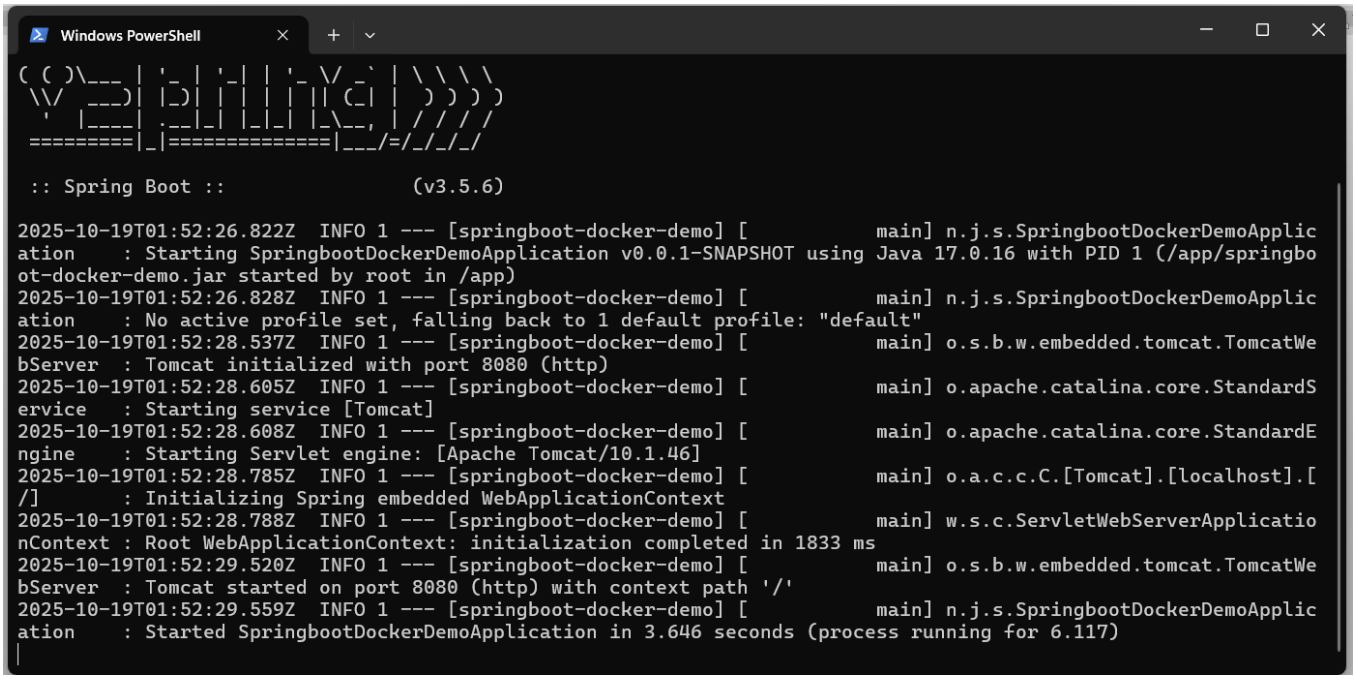


You can see the container's ID

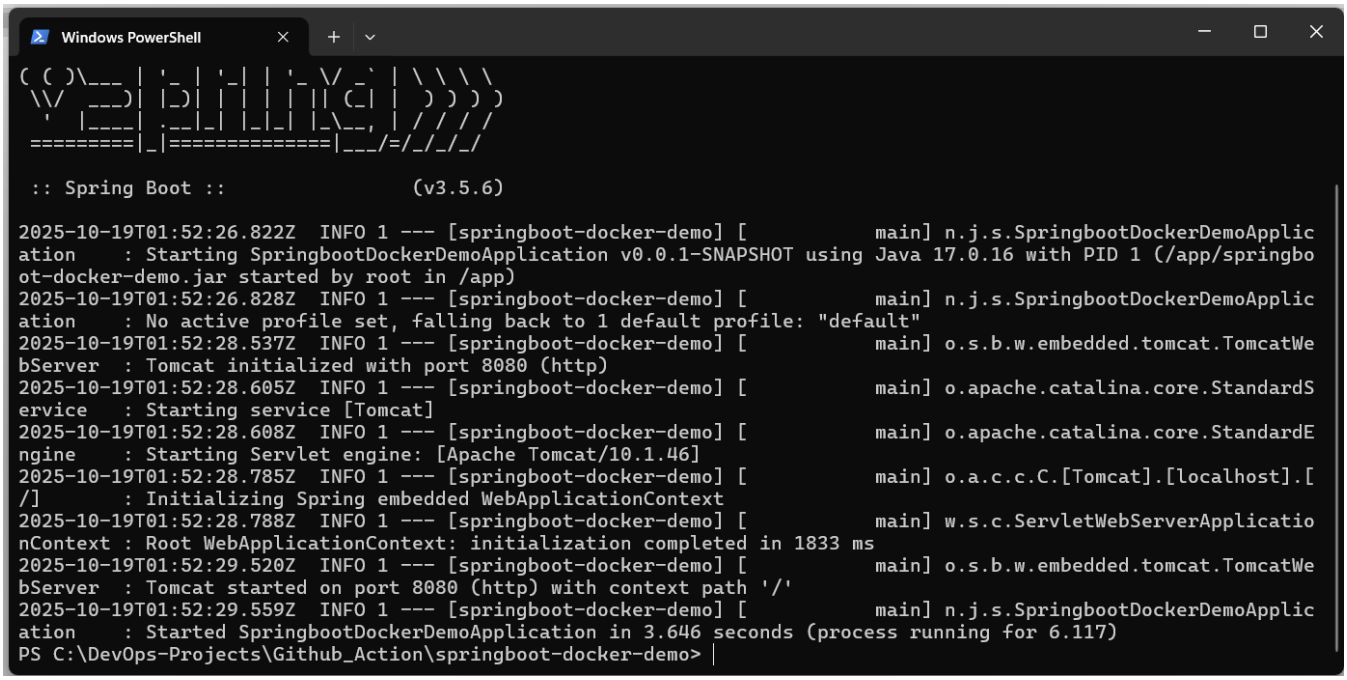
We can now check the logs of the docker container using the command:

```
docker logs -f 5a9f
```

5a9f is the first four digits of the docker container's ID

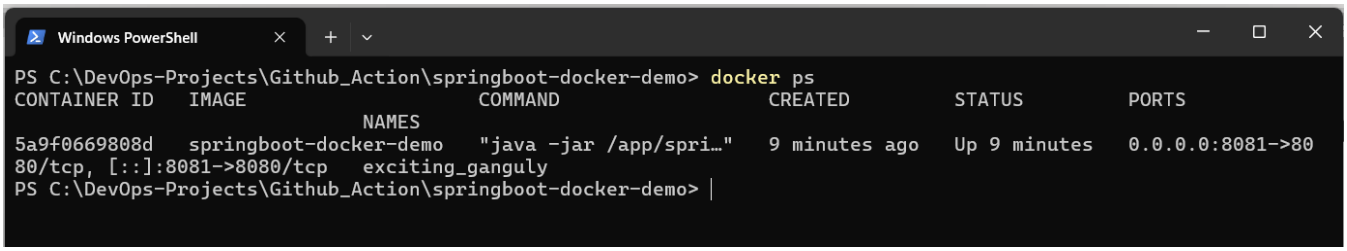


This is the log of the container. If you use "**ctrl + c**" now, you will exit to command mode but the container will still be running in the background



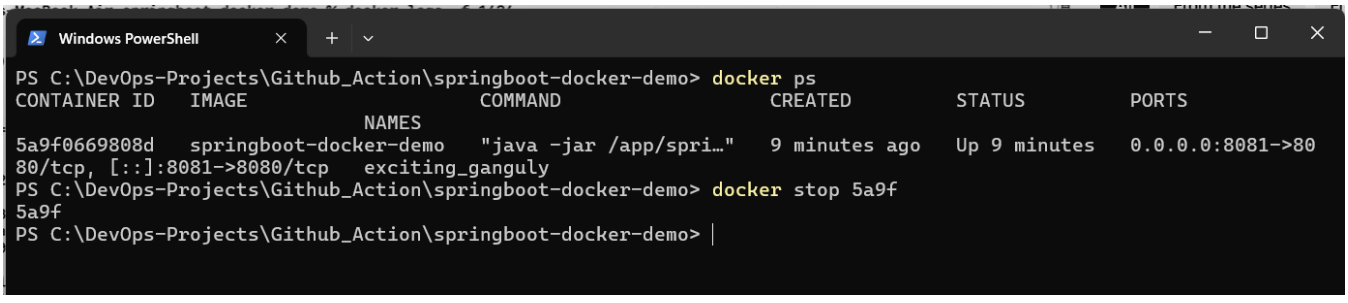
Verify using the command:

```
docker ps
```



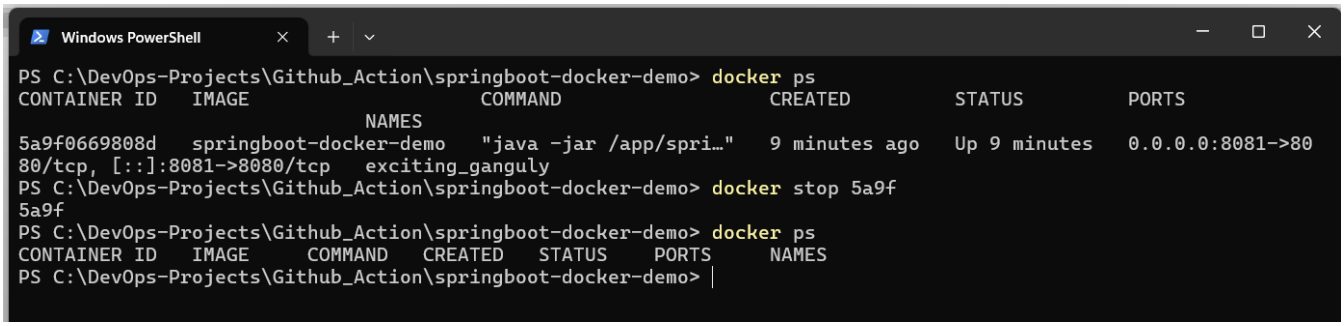
You can see that the container is still running in the background. To stop the container from running completely, you use the command:

```
docker stop 5a9f
```



If you run the command to check the containers now, that is

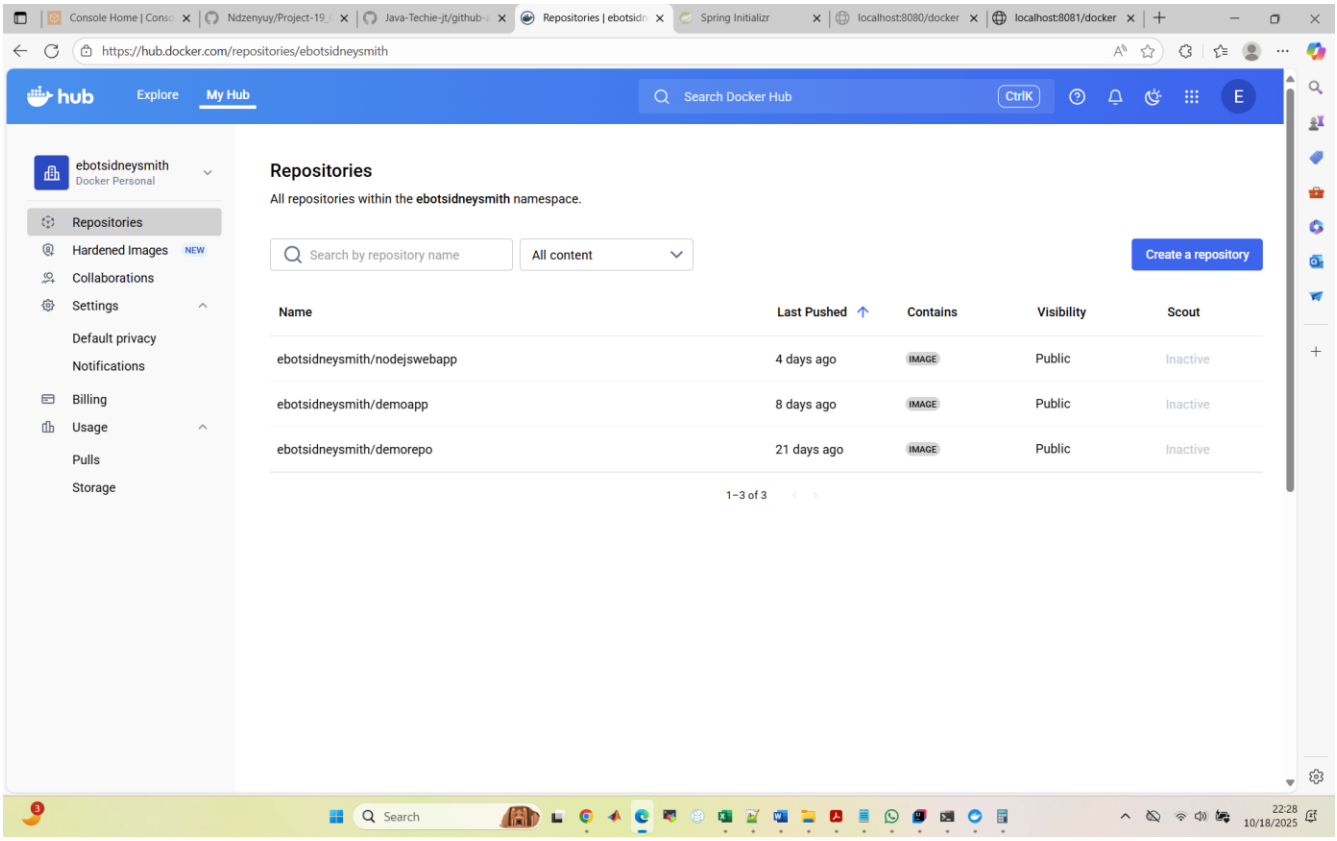
```
docker ps
```



You can see no container is running.

STEP 8: Push Docker Image from local Machine to Docker Hub

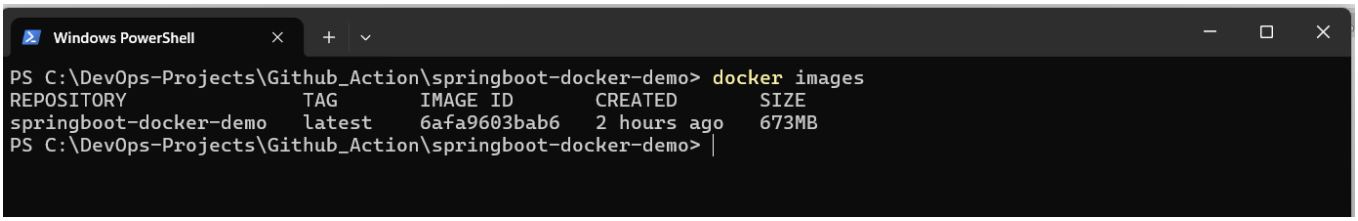
Finally, we will push the Docker image to the Docker hub. To do this you have to first create a account in Docker hub. I already have an account in Docker hub. So, I will skip that step.



Whenever you push a docker image to a docker hub, a new repository is created in your Docker hub to hold the image.

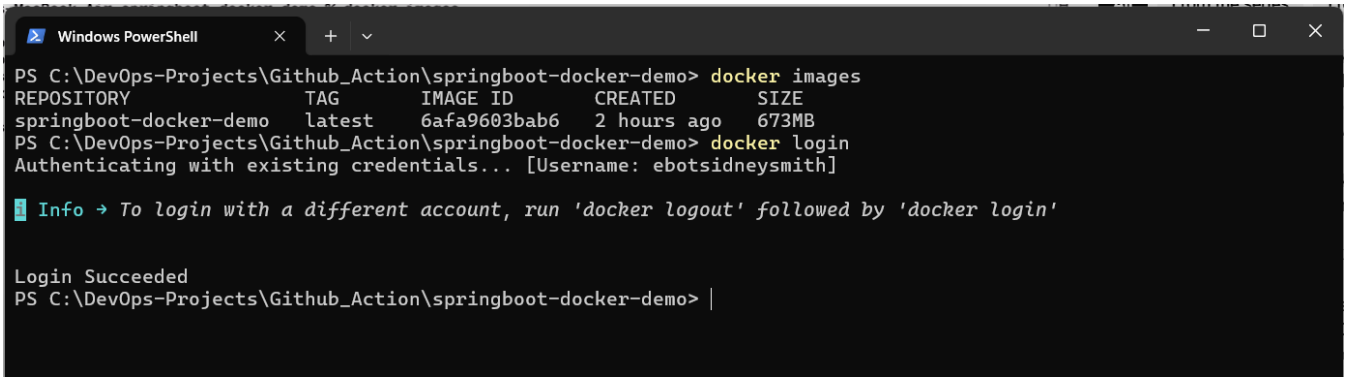
Let us check the docker images using the command:

```
docker images
```



We have one docker image. Now, let us push this image to the docker hub. To do this, we have to login to docker in our terminal. We can do this using the command:

```
docker login
```



I have logged into docker. Now, let us first associate our docker image with the Docker hub using the command:

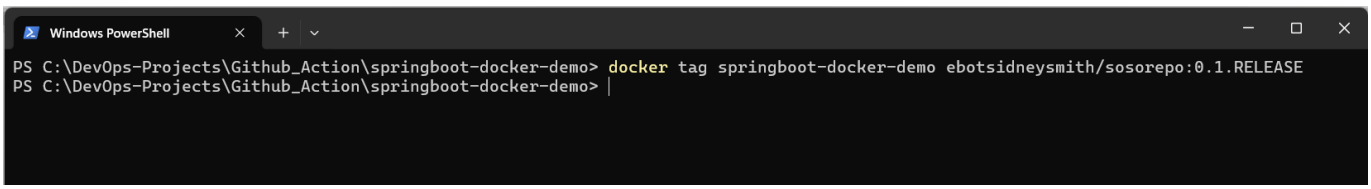
```
docker tag springboot-docker-demo ebotsidneysmith/sosorepo:0.1.RELEASE
```

springboot-docker-demo is the image name

ebotsidneysmith is my docker account ID

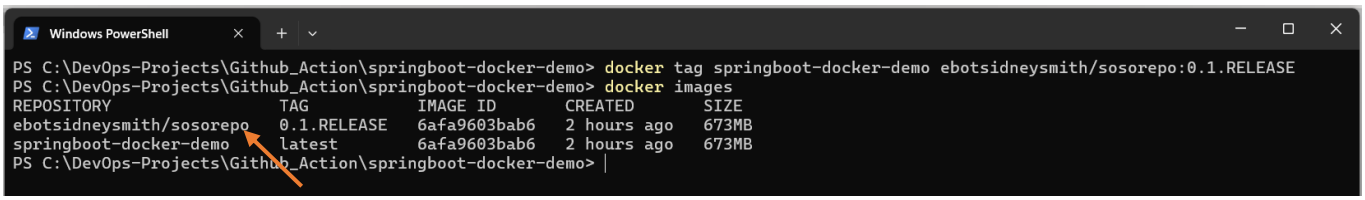
sosorepo is the repository to be created in Docker hub to hold the image

0.1.RELEASE is our image tag



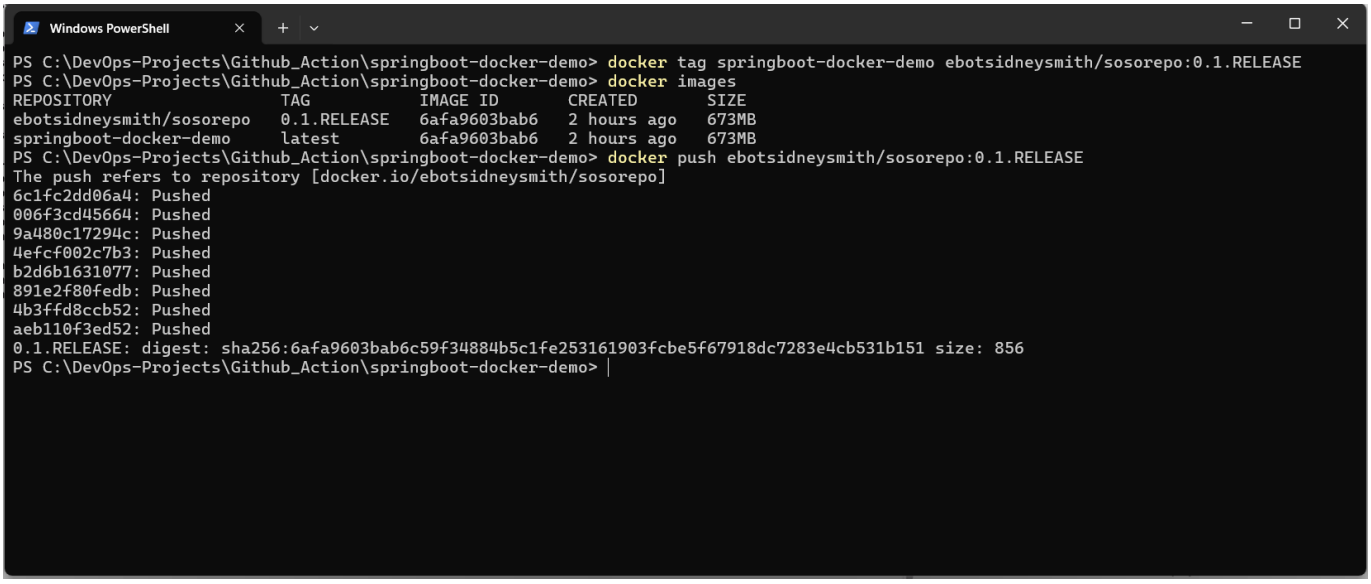
Run the command:

```
docker images
```

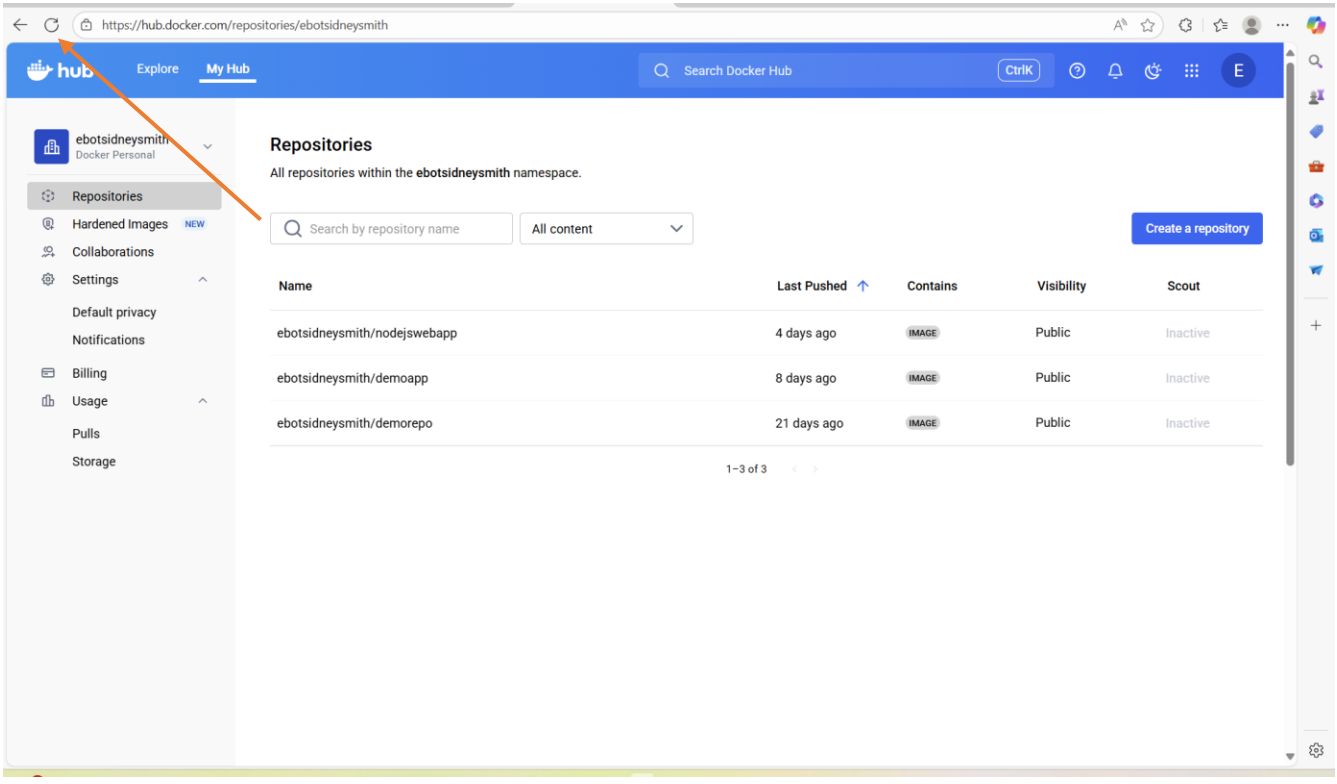


You can see that the repository ebotsidneysmith/sosorepo. Then push the image to the docker hub using the command:

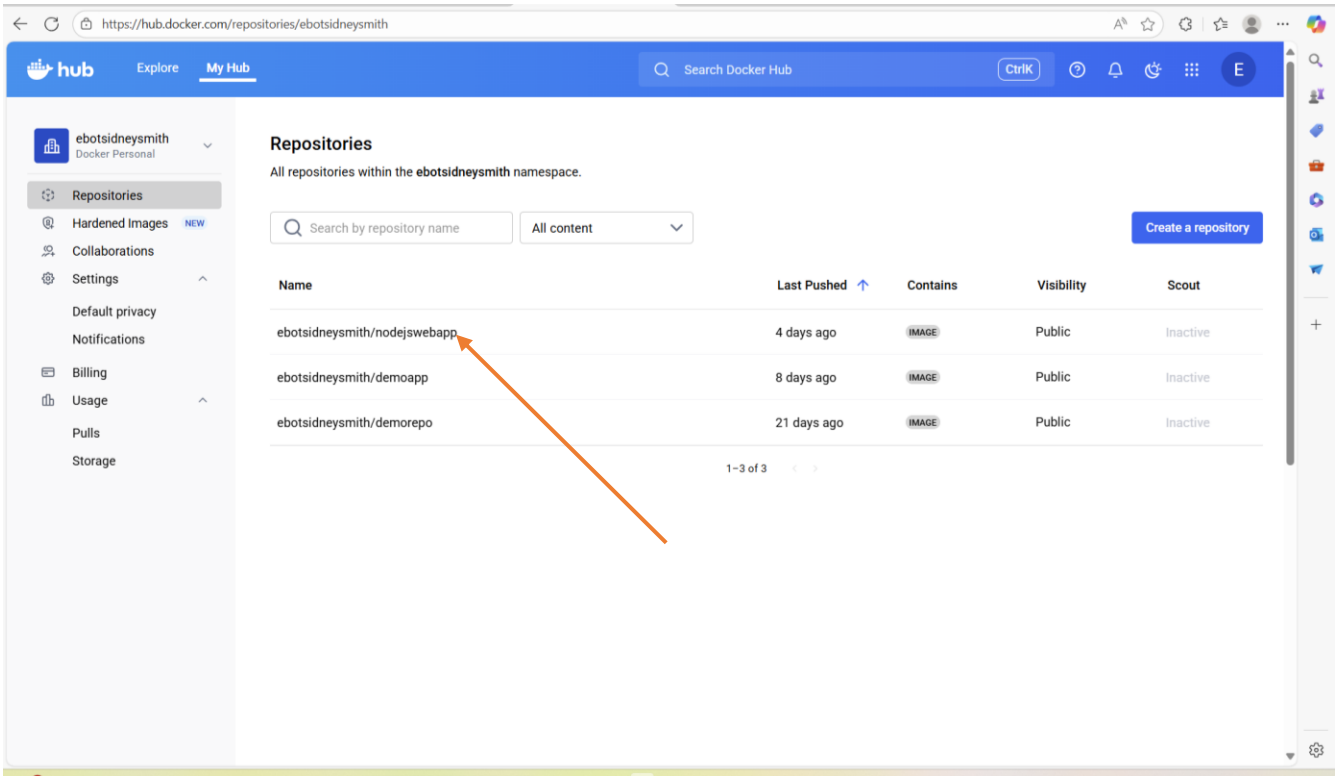
```
docker push ebotsidneysmith/sosorepo:0.1.RELEASE
```



Let us head to our Docker hub now



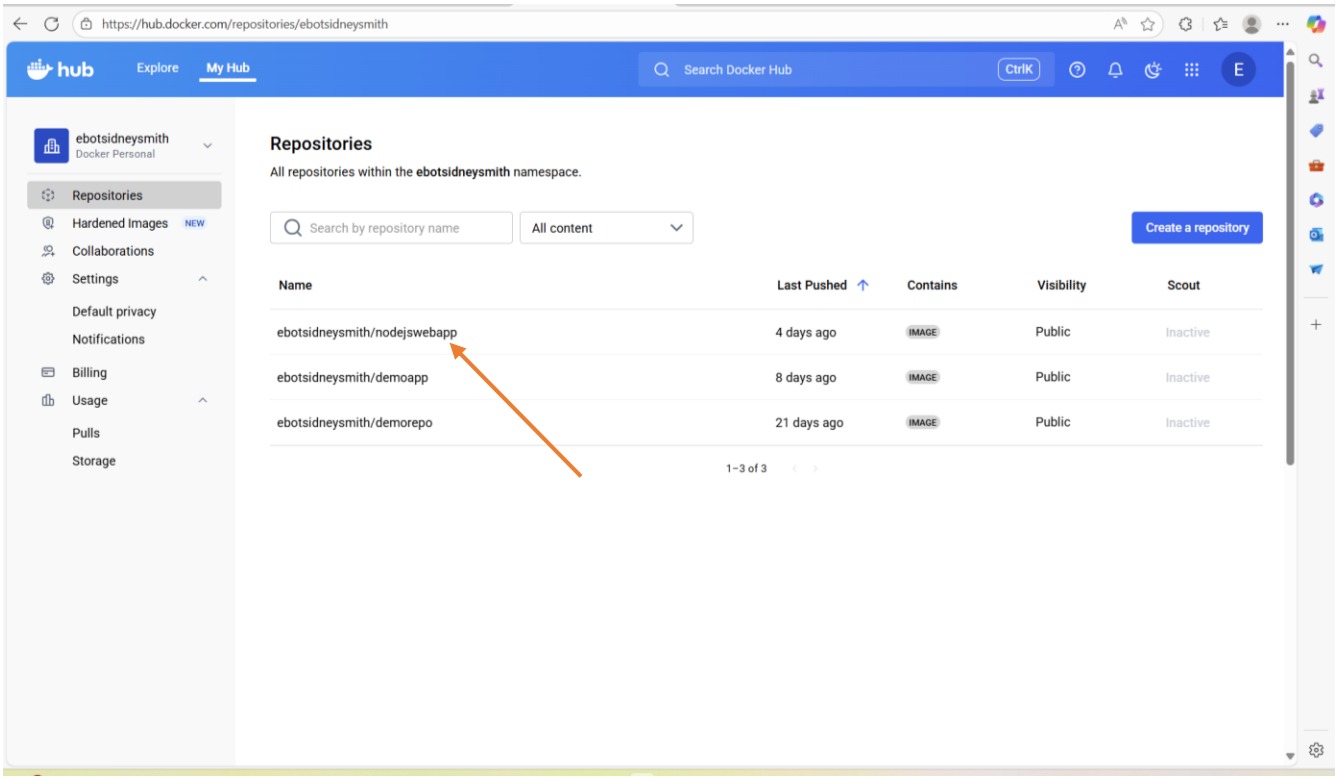
Refresh the page



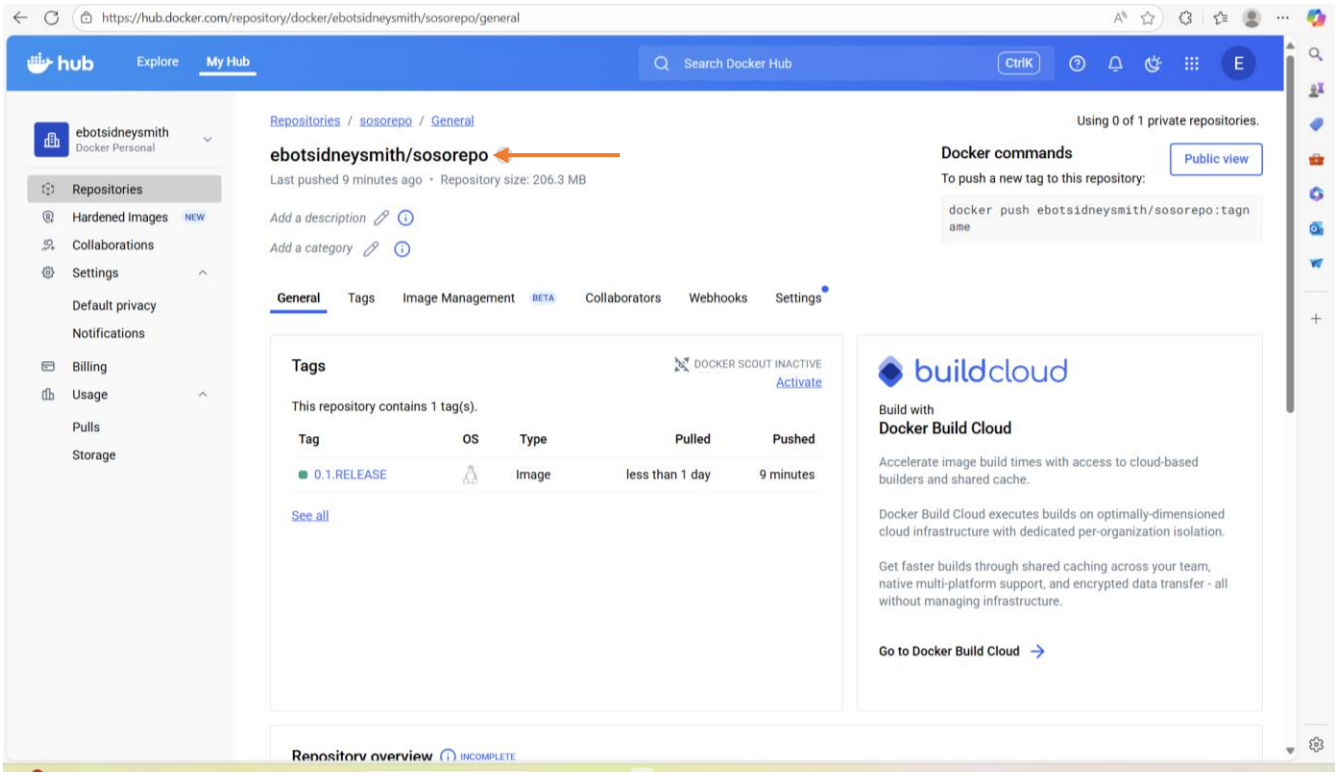
You can see our repository now.

STEP 9: Pull an Image from a Docker Hub

We can also pull an image from a docker hub and deploy it on different environments. This will be demonstrated here.

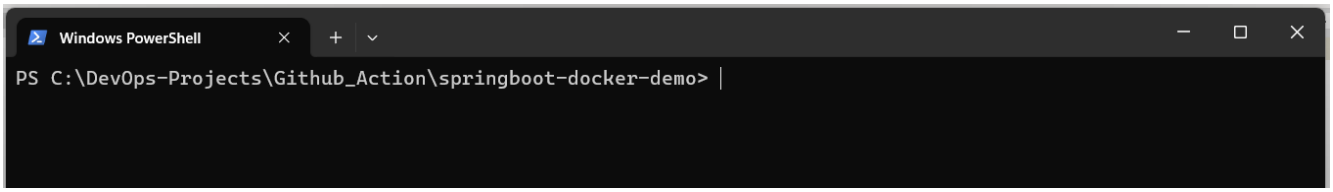


Click on the docker repository



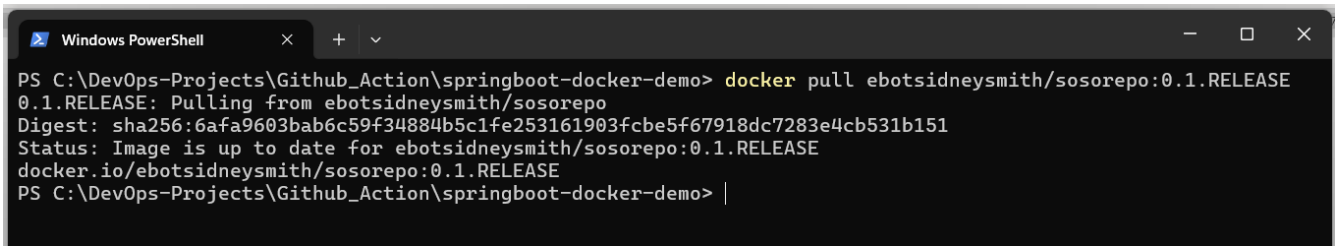
Copy the docker ID and the repository: **ebotsidneysmith/sosorepo**

Head back to our terminal



Then run the command:

```
docker pull ebotsidneysmith/sosorepo:0.1.RELEASE
```



We have successfully pulled the docker image from the docker hub. Let us check using the command:

```
docker images
```