

how to dockerize a Spring Boot application

Dockerizing a Spring Boot application is an essential skill for software developers, as it enables the efficient deployment and management of applications in a containerized environment.

Packaging your Spring Boot app into a Docker container can ensure consistent behavior across different platforms, simplify scaling, and optimize development.

This guide will walk you through the steps to dockerize a Spring Boot application, including creating a Dockerfile, configuring the necessary settings, and building and running the Docker container. Following these steps can enhance your Spring Boot application's performance and maintainability.

To execute these actions, you'll need:

- [JDK 11 for compiling the project](#)
- [Docker Engine to create Spring Boot containers](#)
- [Any cmd bash to run some commands \(Git Bash is optional\)](#)
- [Apache Maven to run some commands to compile the project](#)
- [Your IDE \(I recommend IntelliJ IDEA\)](#)

want to work with the latest tech?

Join EPAM Anywhere to revolutionize your project and get the recognition you deserve.

Download the repository

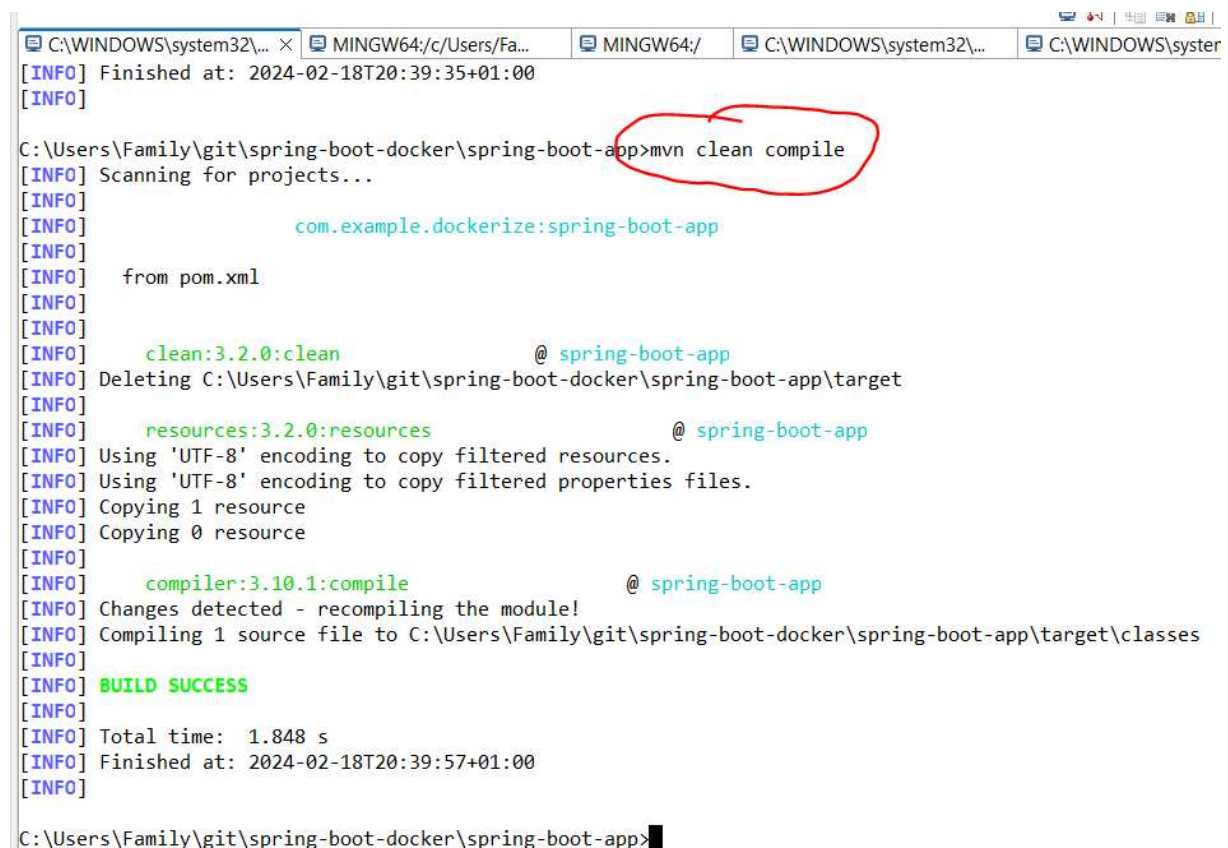
First, we will download [my pre-made repository on Github](#) with all the required dependencies to save some time.

Check the project

Open the downloaded project in your IntelliJ IDEA and run the command **mvn clean compile** to check if the project is working.

mvn clean compile

You should see the following message in your cmd.



```
C:\WINDOWS\system32\... × MINGW64:/c/Users/Fa... MINGW64:/ C:\WINDOWS\system32\... C:\WINDOWS\system32\...
[INFO] Finished at: 2024-02-18T20:39:35+01:00
[INFO]

C:\Users\Family\git\spring-boot-docker\spring-boot-app>mvn clean compile
[INFO] Scanning for projects...
[INFO]
[INFO] com.example.dockerize:spring-boot-app
[INFO]
[INFO] from pom.xml
[INFO]
[INFO] clean:3.2.0:clean @ spring-boot-app
[INFO] Deleting C:\Users\Family\git\spring-boot-docker\spring-boot-app\target
[INFO]
[INFO] resources:3.2.0:resources @ spring-boot-app
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Using 'UTF-8' encoding to copy filtered properties files.
[INFO] Copying 1 resource
[INFO] Copying 0 resource
[INFO]
[INFO] compiler:3.10.1:compile @ spring-boot-app
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to C:\Users\Family\git\spring-boot-docker\spring-boot-app\target\classes
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 1.848 s
[INFO] Finished at: 2024-02-18T20:39:57+01:00
[INFO]

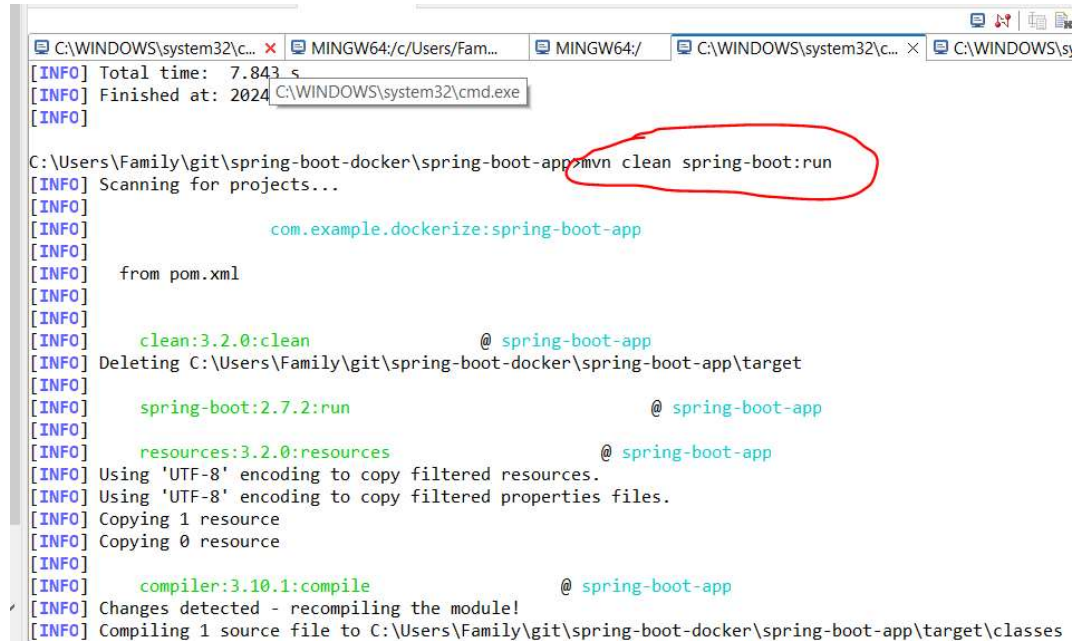
C:\Users\Family\git\spring-boot-docker\spring-boot-app>
```

Check the application

Before we dockerize a Spring Boot app, we'll need to check if our Spring Boot application is also running. Let's run the next command in the internal terminal in IntelliJ (run it from your cmd or Git Bash in Windows):

mvn clean spring-boot:run

You should see the following message in your cmd.

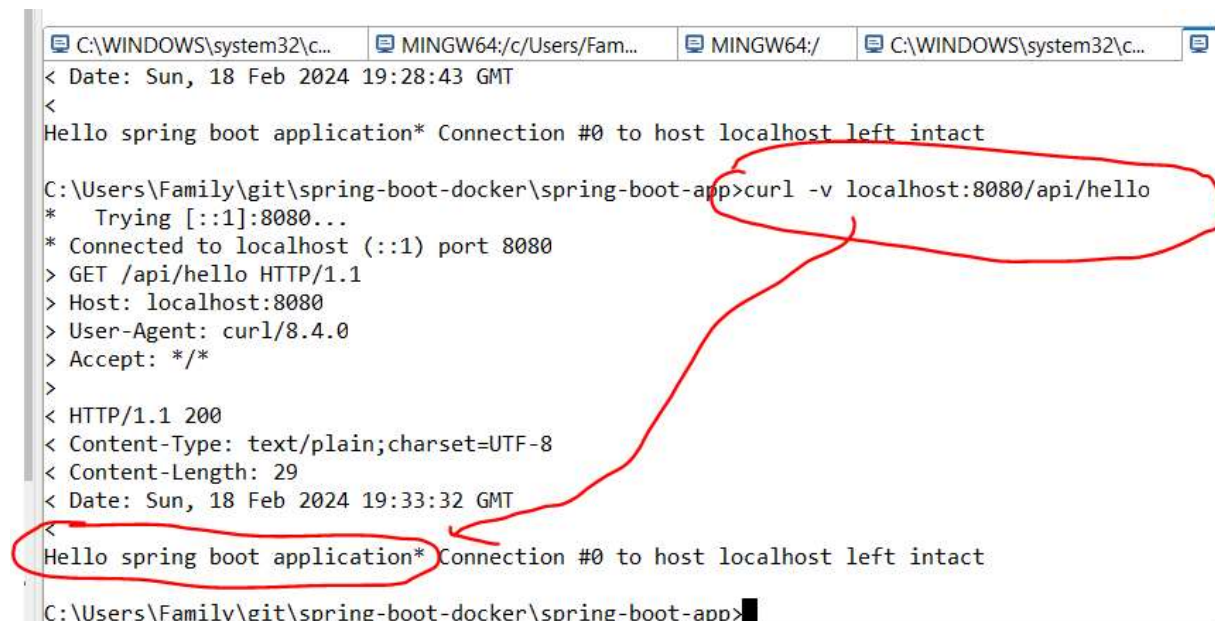


```
C:\WINDOWS\system32\cmd.exe
[INFO] Total time: 7.843 s
[INFO] Finished at: 2024 C:\WINDOWS\system32\cmd.exe
[INFO]

C:\Users\Family\git\spring-boot-docker\spring-boot-app>mvn clean spring-boot:run
[INFO] Scanning for projects...
[INFO]
[INFO]               com.example.dockerize:spring-boot-app
[INFO]
[INFO] from pom.xml
[INFO]
[INFO]      clean:3.2.0:clean @ spring-boot-app
[INFO] Deleting C:\Users\Family\git\spring-boot-docker\spring-boot-app\target
[INFO]
[INFO]      spring-boot:2.7.2:run @ spring-boot-app
[INFO]
[INFO]      resources:3.2.0:resources @ spring-boot-app
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Using 'UTF-8' encoding to copy filtered properties files.
[INFO] Copying 1 resource
[INFO] Copying 0 resource
[INFO]
[INFO]      compiler:3.10.1:compile @ spring-boot-app
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to C:\Users\Family\git\spring-boot-docker\spring-boot-app\target\classes
```

curl -v localhost:8080/api/hello

You should see the following message in your cmd to validate that the app is already running and returning the proper message in this route:



```
C:\WINDOWS\system32\cmd.exe
< Date: Sun, 18 Feb 2024 19:28:43 GMT
<
Hello spring boot application* Connection #0 to host localhost left intact

C:\Users\Family\git\spring-boot-docker\spring-boot-app>curl -v localhost:8080/api/hello
* Trying [::1]:8080...
* Connected to localhost (::1) port 8080
> GET /api/hello HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/8.4.0
> Accept: */*
>
< HTTP/1.1 200
< Content-Type: text/plain;charset=UTF-8
< Content-Length: 29
< Date: Sun, 18 Feb 2024 19:33:32 GMT
<
Hello spring boot application* Connection #0 to host localhost left intact

C:\Users\Family\git\spring-boot-docker\spring-boot-app>
```

Add configuration to dockerize the Spring Boot application

The next step of dockerizing a Spring Boot app is adding the configuration to the Dockerfile for Spring Boot app.

To clarify, Docker is a platform that combines your application, its dependencies, and even an OS into an image that can run on any platform.

Now we will create a Dockerfile to add all the configurations to dockerize our Spring Boot application.

Create a Dockerfile and add the next configurations (Docker will read as a pipeline to apply them):

```
FROM openjdk:11
```

```
VOLUME /tmp
```

```
EXPOSE 8080
```

```
ARG JAR_FILE=target/spring-boot-docker.jar
```

```
ADD ${JAR_FILE} app.jar
```

```
ENTRYPOINT ["java","-jar","/app.jar"]
```

Now I will explain a little bit about these commands:

- **FROM** creates a layer from an existing Spring Boot Docker image that exists locally or in any container registry. **openjdk:11** will be the one to use.
- **VOLUME** creates a specific space to persist some data in your container. The **tmp** folder will store information.
- **EXPOSE** informs Docker that the container listens to the specified network ports at runtime. This is the port to access the Spring Boot container and will be used to run the container.
- **ARG** defines a variable that can be passed to the application at runtime. For example, we pass the location of the final jar file within the target folder and save it in a **JAR_FILE** variable. You can also pass more arguments like credentials, keys, and environment variables with their respective values.
- **ADD** copies new files, directories or remote file URLs from the source and adds them to the filesystem of the image at the provided path. In our case we add the Spring Boot application to the Docker image from the source path (the **JAR_FILE** variable) to a destination named **app.jar**.
- **ENTRYPOINT** specifies the command that Docker will use to run our app. In this case it will pass the common command to run a jar — **java -jar <name of the jar>** — so in this case it is **java -jar app.jar** to our **ENTRYPOINT** option (remember that we renamed the spring-boot-docker.jar file to **app.jar**).

That's all for the configuration.

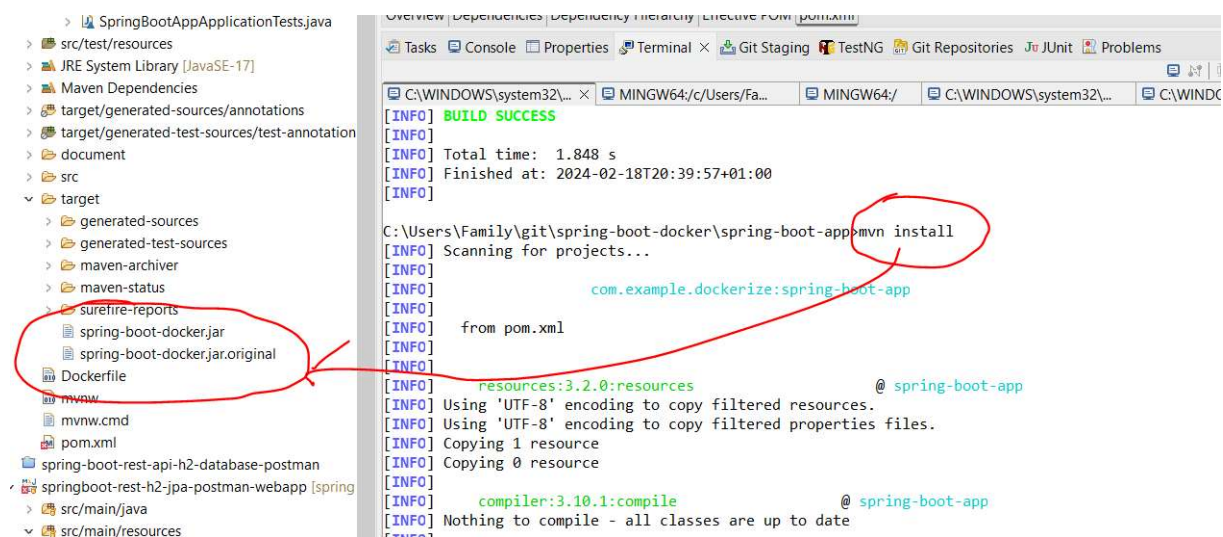
related:

[creating a streaming pipeline in apache beam](#) [read more](#)

Generate a .jar file

Now we need to generate our .jar file running the **mvn install** command in your IntelliJ terminal. This will generate a .jar file with all the classes from our application.

When the process is finished, go to the target folder from your project and see the next **spring-boot-docker.jar** file.

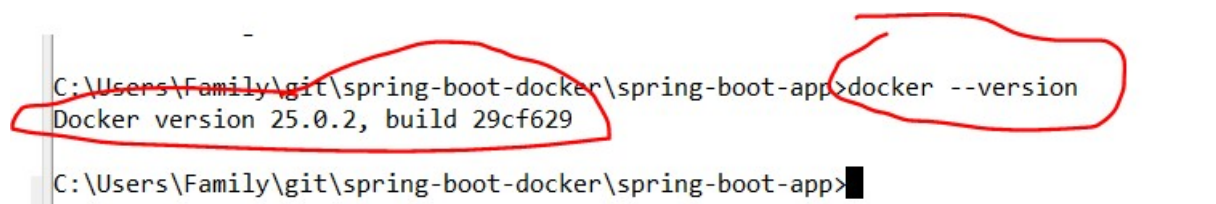


Build a Spring Boot Docker image

Great, we're done with the preparations, and it's time to build our image using Docker Engine.

Make sure you have Docker Engine installed. Run the `docker ps` or `docker info` command on the terminal screen to check it. If the command is not found, you may need to install Docker first. In this case, please follow [this link](#) and find the installer for your OS.

docker --version



docker-compose --version

```
C:\Program Files\Docker\Docker>docker-compose --version
Docker Compose version v2.24.3-desktop.1
C:\Program Files\Docker\Docker>
```

"C:\Program Files\Docker\Docker\DockerCli.exe" -SwitchDaemon

```
C:\Program Files\Docker\Docker>"C:\Program Files\Docker\Docker\DockerCli.exe" -SwitchDaemon
switching to windows engine: request failed and retry attempts exhausted: Post "http://ipc/engine/switch": open
\\.\pipe\dockerBackendApiServer: The system cannot find the file specified.
C:\Program Files\Docker\Docker>
```

Run your Docker Engine. Find the folder with the Dockerfile of your Spring Boot project in the terminal and execute the following command (make sure to end the command with a space and a dot):

docker build -t spring-boot-docker:spring-docker .

The “**build**” command will build an image according to the instructions we passed to the Dockerfile, and the -t flag is used to add a tag for our image.

In a few minutes, you will see that the image was successfully created:

```
#7 exporting to image
#7 sha256:e8c613e07b0b7ff33893b694f7759a10d42e180f2b4dc349fb57dc6b71dcab00
#7 exporting layers done
#7 writing image sha256:f4ff5be8f76bc5def149a92841e37b3a2e3e25754df18a903379485af4285edd done
#7 naming to docker.io/library/spring-boot-docker:spring-docker done
#7 DONE 0.0s

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
```

Run the **docker image ls** command to check if the image exists. You should be able to see your image in the repository.

```
$ docker image ls
REPOSITORY          TAG          IMAGE ID      CREATED        SIZE
spring-boot-docker  spring-docker  f4ff5be8f76b  21 minutes ago  672MB
```

How to change the base image

Changing the base image in a Dockerfile is a straightforward process that can significantly impact your container's functionality, size, and security. To change the base image, follow these steps:

1. Choose a new base image

Before making any changes, research and select a suitable base image that meets your application's requirements. Consider factors such as the operating system, size, security, and compatibility with your application. Official images from the Docker Hub or other trusted sources are recommended.

In our example, we will choose [eclipse-temurin:11](#) as a base image, a lightweight Java image.

2. Update the Dockerfile

Open your Dockerfile in a text editor. Locate the ***FROM*** instruction at the file's beginning, specifying the current base image. Replace the existing image with the new one you've chosen. For example, if you want to change the base image from `openjdk:11` to `eclipse-temurin:11`, update the ***FROM*** instruction as follows:

```
FROM eclipse-temurin:11
```

3. Adjust dependencies and configurations

Depending on the differences between the old and new base images, you may need to update the dependencies, configurations, or commands in your Dockerfile. Ensure all required packages, libraries, and tools are installed and configured correctly for the new base image.

This `eclipse-temurin:11` has the necessary dependencies to run your application.

4. Test your changes

After updating the Dockerfile, rebuild your Docker image using the `docker build` command:

```
docker build -t your-image-name .
```

Replace `your-image-name` with a suitable name for your image. Remember the period at the end of the command, which indicates the build context (usually the current directory).

Or you can use **commit** option to create a new image with the new changes that you add.

related:

[using GraalVM & AWS Lambda in Java for cold start problems](#)read more

Run the Spring Boot Docker image in a container

Now, we'll run our image in a container, but first, let's make sure we won't get an error trying to point our container port to the localhost one.

Run the next command:

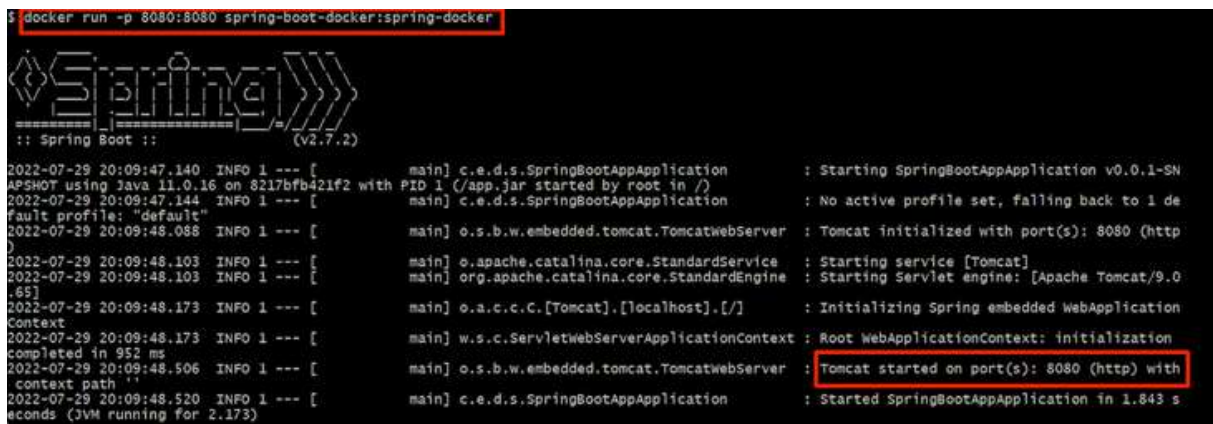
```
docker run -p 8080:8080 spring-boot-docker:spring-docker .
```

You may add the **-d** flag before **-p** to avoid seeing any logs and run the container in the background mode.

The flag **-p** creates a firewall rule that maps the previously exposed container port:8080 to the :8080 port on your machine.

If everything is done right, our container should be running.

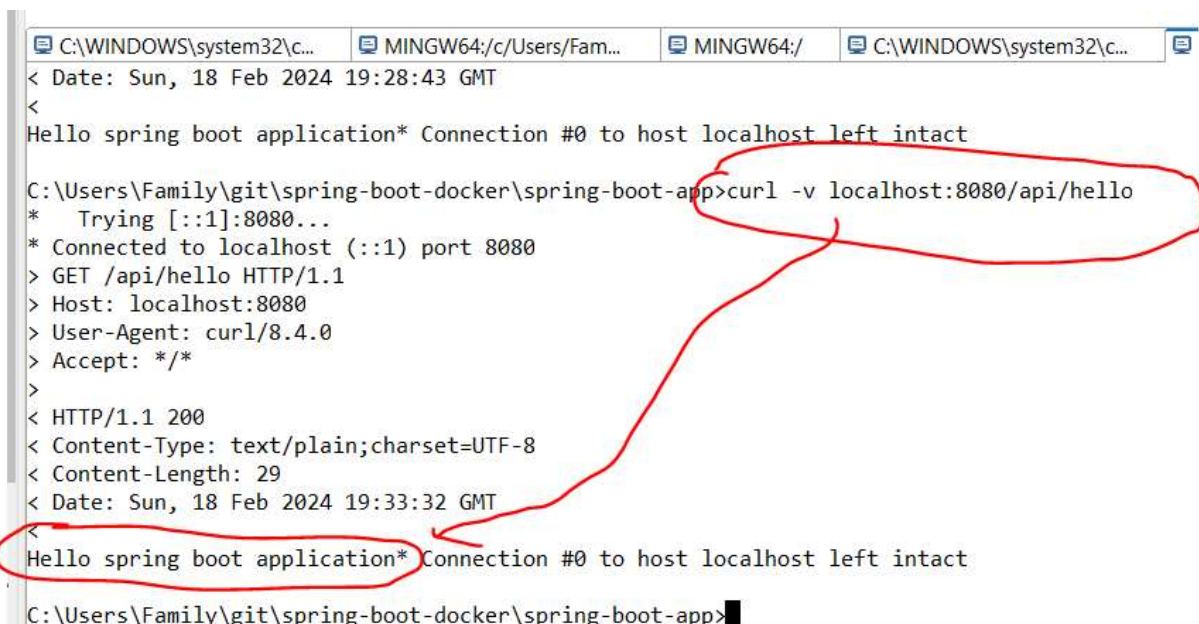
```
$ docker run -p 8080:8080 spring-boot-docker:spring-docker
```



```
Spring
:: Spring Boot ::
(v2.7.2)
2022-07-29 20:09:47.140 INFO 1 --- [main] c.e.d.s.SpringBootApplication : Starting SpringBootApplication v0.0.1-SNAPSHOT using Java 11.0.16 on 8217bfb421f2 with PID 1 (/app.jar started by root in /)
2022-07-29 20:09:47.144 INFO 1 --- [main] c.e.d.s.SpringBootApplication : No active profile set, falling back to 1 default profile: "default"
2022-07-29 20:09:48.088 INFO 1 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2022-07-29 20:09:48.103 INFO 1 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2022-07-29 20:09:48.103 INFO 1 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.65]
2022-07-29 20:09:48.173 INFO 1 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplication
2022-07-29 20:09:48.173 INFO 1 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 952 ms
2022-07-29 20:09:48.506 INFO 1 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2022-07-29 20:09:48.520 INFO 1 --- [main] c.e.d.s.SpringBootApplication : Started SpringBootApplication in 1.843 seconds (JVM running for 2.173)
```

Then, run the **curl -v localhost:8080/api/hello** command in your terminal to test if we can access the application's endpoint.

You must see the same response as at the beginning of the tutorial.



```
C:\WINDOWS\system32\cmd.exe MINGW64:/c/Users/Fam... MINGW64:/ C:\WINDOWS\system32\cmd.exe
< Date: Sun, 18 Feb 2024 19:28:43 GMT
<
Hello spring boot application* Connection #0 to host localhost left intact

C:\Users\Family\git\spring-boot-docker\spring-boot-app>curl -v localhost:8080/api/hello
* Trying [::1]:8080...
* Connected to localhost (::1) port 8080
> GET /api/hello HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/8.4.0
> Accept: */*
>
< HTTP/1.1 200
< Content-Type: text/plain; charset=UTF-8
< Content-Length: 29
< Date: Sun, 18 Feb 2024 19:33:32 GMT
<
Hello spring boot application* Connection #0 to host localhost left intact

C:\Users\Family\git\spring-boot-docker\spring-boot-app>
```

Does Spring Boot 2.3 support buildpacks?

Spring Boot 2.3 introduced native support for Cloud Native Buildpacks, simplifying creating Docker images for your Spring Boot applications. Buildpacks automatically detect and configure the required dependencies, runtime, and application settings, eliminating the need to write a custom Dockerfile.

Here's what you need to know about Buildpacks support in Spring Boot 2.3 when dockerizing your app.

With Buildpacks support, you can create a Docker image for your Spring Boot app using the following command:

```
mvn spring-boot:build-image -Dspring-boot.build-image.imageName=your-image-name
```

Replace ``your-image-name`` with a suitable name for your image. This command will package your application and create a Docker image using the Paketo Buildpacks.

Now you have already dockerized your app without any Dockerfile, and the process was easy and fast.

In summary, Spring Boot 2.3's Buildpacks support simplifies dockerizing your app by automating the creation of Docker images and optimizing the build process. Using Buildpacks, you can focus on developing your application while benefiting from a streamlined containerization process.

That's it! I hope you learned how to dockerize a Spring Boot application.

In conclusion, dockerizing a Spring Boot application greatly simplifies the deployment process and ensures a uniform and consistent environment across various application lifecycle stages. Docker lends agility, flexibility, and peace of mind, from stringent version control to seamless scaling up.

We've seen how easy it is to encapsulate an application and its dependencies into a Docker image, thus reducing potential inconsistencies and conflicts significantly. This walk-through showcases the method and the beauty of working with some of the latest cutting-edge technology.

Working at EPAM Anywhere translates into being part of an innovative, tech-forward organization that encourages utilizing the latest technology in routine work. It promotes constant learning, growth, and adaptation in a rapidly evolving tech world.

The company's global, flexible, and trendsetting work culture frequently exposes one to the latest and most impactful technologies like Docker, Spring Boot, etc.

Whether you are kick-starting your career or looking to refine your expertise further, applying for [remote Java developer jobs](#) at EPAM Anywhere will let you get your hands on tech tools, constantly challenging you to evolve and learn. You'll always be at the forefront of technological development, making you invaluable in today's dynamic tech market.