



SPRING BOOT BASICS

Guide to web application development using Spring
Boot covers auto-configuration, managing it with
Spring profiles, and deployment by Docker

Vlad Bogdantsev

<https://github.com/vladproduction>

CONTENTS

Preview	1
1) Booting from the Web	2
2) Understanding the Project.....	4
3) Understanding Auto-Configuration	5
4) Configuration in Spring Boot	7
5) Spring Profiles in Boot	8
6) Building Spring Boot Applications	10
7) Containerizing Spring Boot Applications	11
Summary	14

PREVIEW

This document provides a concise guide to rapid web application development using Spring Boot. It covers leveraging Spring Boot's auto-configuration for simplified setup, customizing configurations both during startup and runtime, effectively managing configurations with Spring profiles, and streamlining deployment by building Docker images.

Topics covered one by one:

- * Booting from the Web;
- * Understanding the Project;
- * Understanding Auto-Configuration;
- * Configuration in Spring Boot;
- * Spring Profiles in Boot;
- * Building Spring Boot Applications;
- * Containerizing Spring Boot Applications;

1) BOOTING FROM THE WEB

Spring Boot simplifies the process of creating stand-alone, production-grade Spring-based applications. One of the key features is its ability to run web applications effortlessly.

Steps to create one-by-one

1. Setup Your Environment:

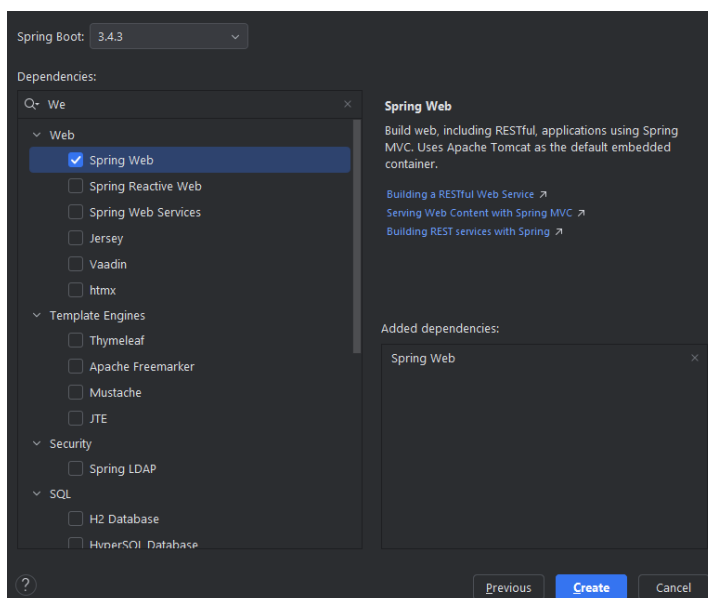
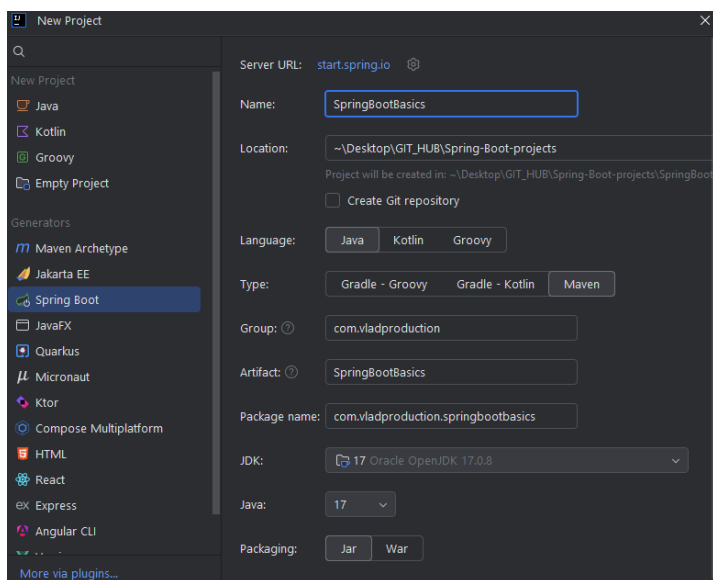
Ensure you have Java Development Kit (JDK) 11 or higher installed.

Choose an Integrated Development Environment (IDE), like IntelliJ IDEA or Eclipse.

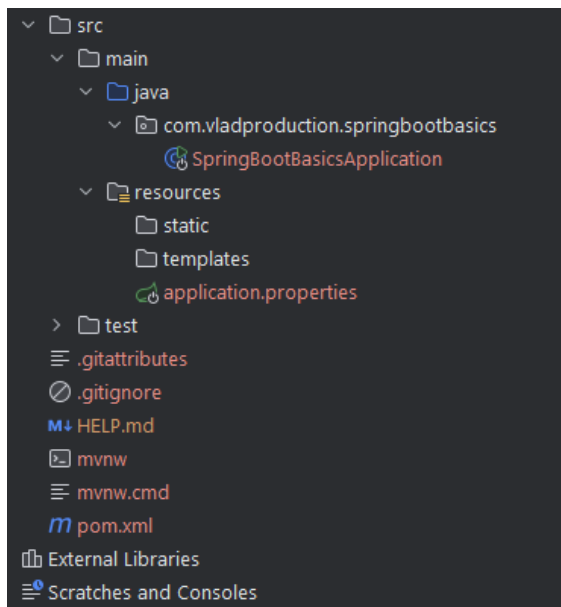
2. Creating a Spring Boot Project:

Use Spring Initializer (<https://start.spring.io/>) to scaffold a new project.

Select dependencies like "Spring Web" to get the web module.

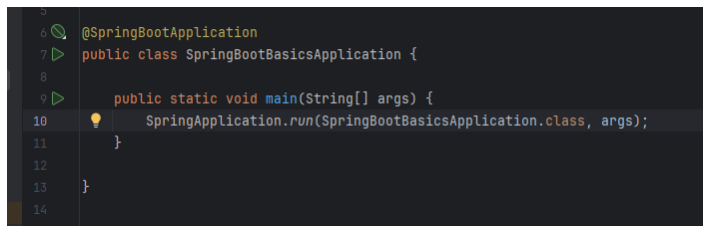


3. Project Structure (the basic structure of the project can look like):



4. SpringBootBasicApplication.java:

This is the entry point of your Spring Boot application.



5. Creating a Simple REST Controller:

Create a REST controller to serve HTTP requests.



6. Running the Application:

Use the command line or your IDE to run SpringBootBasicApplication.java.

By default, the application will run on localhost:8080.

7. Testing the Endpoint:

Open a web browser or a tool like Postman.

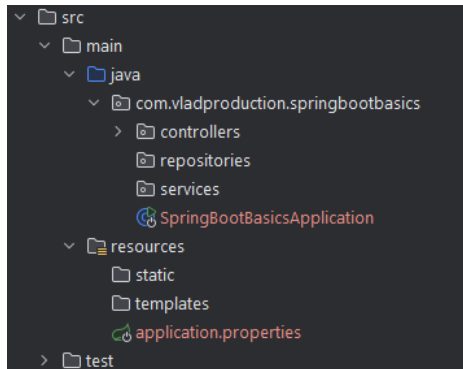
Navigate to <http://localhost:8080/hello>, and you should see "Hello, Spring Boot!"

2) UNDERSTANDING THE PROJECT

In a Spring Boot application, understanding the project structure and key components is essential for developing effective applications. Let's break down the important parts of a Spring Boot project.

Key Components of a Spring Boot Project

1. Project Structure:



2. Main Application Class (SpringBootBasicApplication.java):

It contains the `@SpringBootApplication` annotation which is a combination of:

`@Configuration`: Indicates that the class can be used by the Spring IoC container as a source of bean definitions.

`@EnableAutoConfiguration`: Tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.

`@ComponentScan`: Enables scanning for components (e.g., controllers, services) in the specified package.

3. Controller Package:

This is where you define REST controllers. Each controller is annotated with `@RestController`, and you map HTTP requests to handler methods using annotations like `@GetMapping`, `@PostMapping`, etc.

4. Resources:

The resources directory contains configuration files and static resources.

`application.properties`: This file is used to define application-specific configurations such as server port, database URLs, etc.

5. Dependency Management:

Spring Boot uses Maven or Gradle for dependency management. You will find a file named `pom.xml` (for Maven) or `build.gradle` (for Gradle) in the project root.

Example of dependencies in `pom.xml`:

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

6. Spring Boot Starter:

Starters are a set of convenient dependency descriptors you can include in your application. For example, `spring-boot-starter-web` includes dependencies for Spring MVC, REST, and Tomcat.

7. Profiles:

Spring Profiles allow you to have different configurations for different environments (development, testing, production) within the `application.properties`.

Example: A Simple Application Configuration

Here's a brief look at what your `application.properties` might contain:

```
spring.application.name=SpringBootBasics

server.port=8080

spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=secret
```

3) UNDERSTANDING AUTO-CONFIGURATION

Auto-configuration is one of the most powerful features of Spring Boot. It attempts to automatically configure your Spring application based on the dependencies you have added. This reduces the need for a lot of manual configuration.

How Auto-Configuration Works

1. Spring Boot Starter:

When you include a starter dependency (e.g., `spring-boot-starter-web`), Spring Boot provides a set of configurations that it considers essential for that particular module.

2. Conditional Annotations:

Auto-configuration classes are annotated with `@Conditional`, `@ConditionalOnClass`, and `@ConditionalOnMissingBean` among other annotations to control whether a specific configuration should occur or not.

`@Configuration`

`@ConditionalOnClass(DataSource.class)`

```
public class DataSourceAutoConfiguration {
    // Configuration logic for DataSource
}
```

3. Spring Factories:

Spring Boot uses a file named `spring.factories` located in the `META-INF` folder of the jar files. This file lists all auto-configuration classes.

When your application starts, Spring Boot reads this file and applies the necessary configuration automatically.

4. Debugging Auto-Configuration:

You can enable debugging for auto-configuration by setting the following property in the application.properties file:

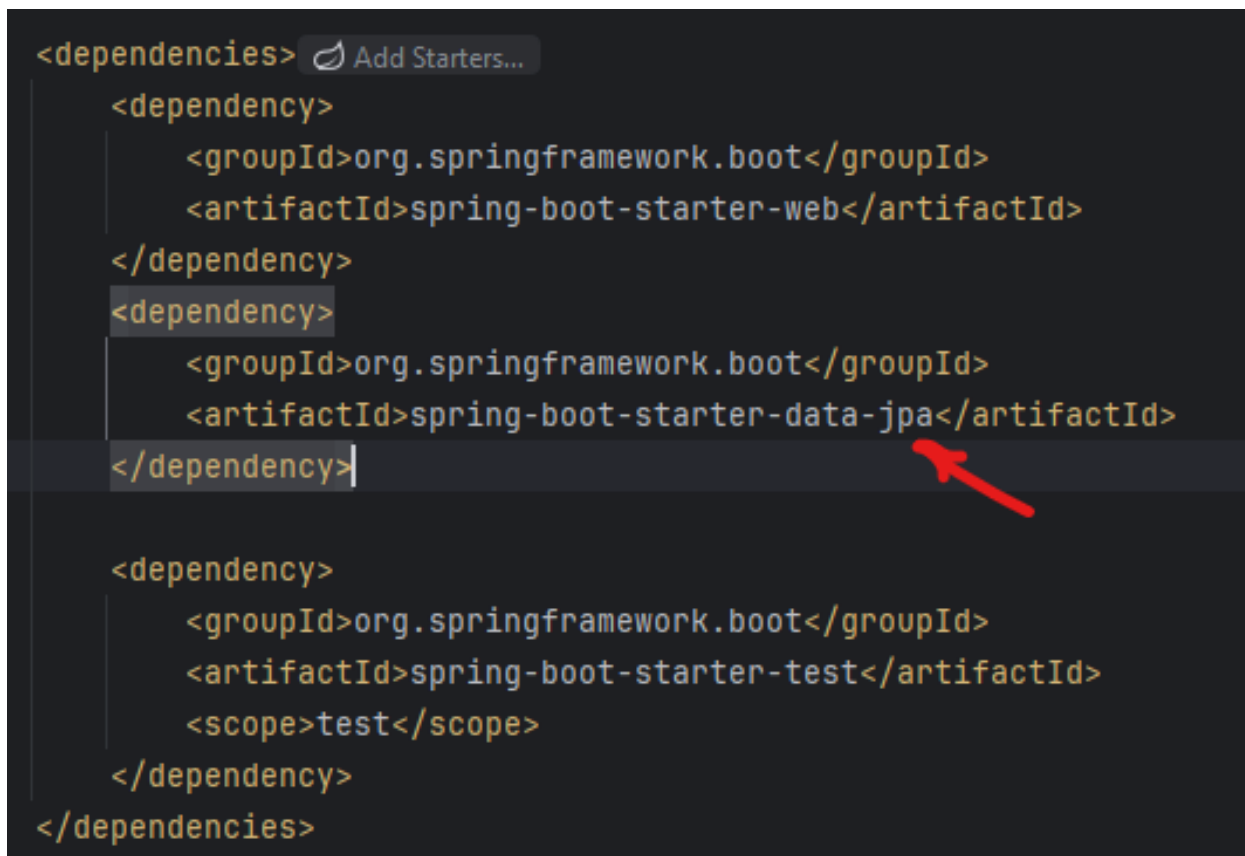
debug=true

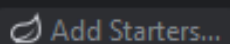
This will log the auto-configuration report in the console, showing which configurations were applied and which were not, along with the reasons.

5. Auto-configuration capabilities

The @SpringBootApplication annotation already includes auto-configuration capabilities, meaning that Spring Boot will automatically set up various configurations based on what dependencies are in your classpath.

For example, if you add the Spring Data JPA dependency in your pom.xml:



```
<dependencies> 
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

With this dependency, Spring Boot will automatically configure:

An instance of EntityManager for JPA if a compatible JPA provider (like Hibernate) is on the classpath. H2 or other database configurations if present.

Auto-configuration dramatically simplifies the setup process, helping developers focus on application logic instead of boilerplate configuration.

4) CONFIGURATION IN SPRING BOOT

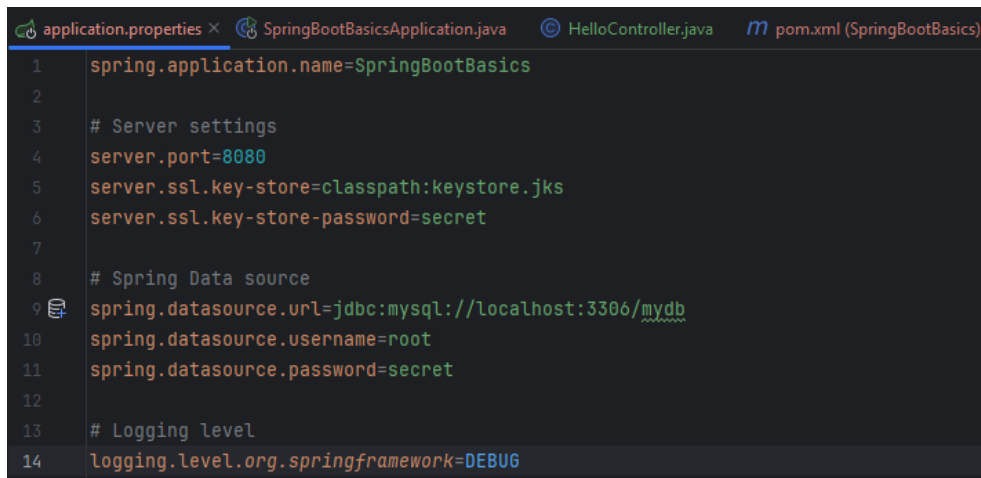
Spring Boot provides various ways to configure your application. The configuration options range from simple property files to more complex structures like YAML files and programmatic configuration. Understanding these options helps you customize your application settings effectively.

Key Configuration Methods

1. Application Properties File:

The simplest form of configuration is the application.properties file located in the src/main/resources directory.

Example:



```
1 spring.application.name=SpringBootBasics
2
3 # Server settings
4 server.port=8080
5 server.ssl.key-store=classpath:keystore.jks
6 server.ssl.key-store-password=secret
7
8 # Spring Data source
9 spring.datasource.url=jdbc:mysql://localhost:3306/mydb
10 spring.datasource.username=root
11 spring.datasource.password=secret
12
13 # Logging level
14 logging.level.org.springframework=DEBUG
```

2. YAML Configuration:

You can also use YAML files (application.yml) for configuration. YAML is more readable for hierarchical data structures.

Example of application.yml:

server:

port: 8080

spring:

application:

name: MySpringBootApplication

logging:

level:

root: INFO

3. Environment Variables:

Spring Boot allows you to configure properties using environment variables. For instance, you can define a variable like SERVER_PORT and refer to it in your application.properties:

server.port=\${SERVER_PORT:8080} # Defaults to 8080 if SERVER_PORT isn't set

4. Command Line Arguments:

You can pass configurations as command line arguments when running your application:

java -jar myapp.jar --server.port=9090

5. Profiles:

Spring Boot supports different profiles (e.g., development, production) for different configurations. You can define properties in specific profile configuration files like `application-dev.properties` or `application-prod.properties`.

Activate a profile using:

`spring.profiles.active=dev`

6. Custom Configuration Classes:

You can create custom configuration classes using `@Configuration`, where you define beans specific to your application. For example:

```

7  @Configuration new *
8  public class MyCustomConfig {
9
10     @Bean new *
11     public MyService myService() {
12         return new MyService();
13     }
14 }
```

Spring Boot's configuration options provide flexibility to adapt your application settings to different environments while promoting maintainability and readability.

5) SPRING PROFILES IN BOOT

Spring Profiles allow you to define different configurations for different environments, such as development, testing, and production. This feature is especially useful for managing environment-specific configurations without changing the whole application.

How to Use Spring Profiles

1. Defining Profiles:

You can define profiles in your `application.properties` or `application.yml` using the naming convention `application-{profile}.properties` or `application-{profile}.yml`.

For example:

`application-dev.properties` for the development profile.

`application-prod.properties` for the production profile.

2. Activating a Profile (can activate a specific profile in several ways):

- Via `application.properties`:

`spring.profiles.active=dev`

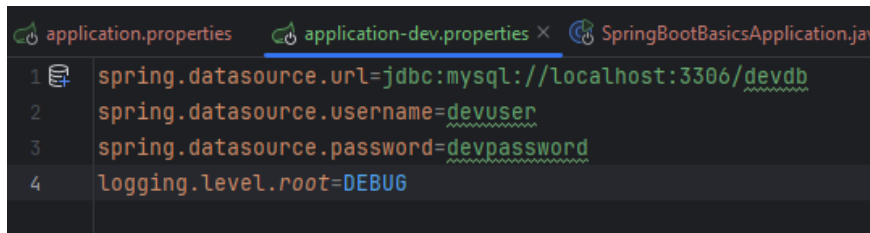
- Via Command Line (bash):

`java -jar myapp.jar --spring.profiles.active=prod`

- Via Environment Variables: Set the environment variable (bash):

`SPRING_PROFILES_ACTIVE=prod`

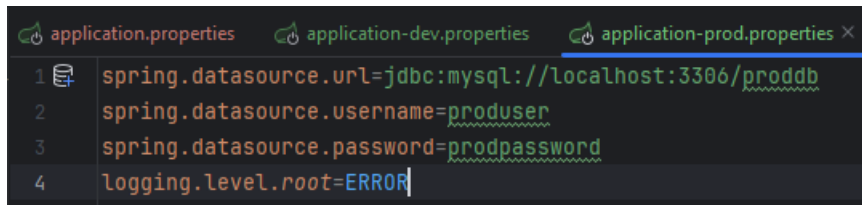
3. Example Profile-Specific Configuration Files:



```

1 spring.datasource.url=jdbc:mysql://localhost:3306/devdb
2 spring.datasource.username=devuser
3 spring.datasource.password=devpassword
4 logging.level.root=DEBUG

```



```

1 spring.datasource.url=jdbc:mysql://localhost:3306/proddb
2 spring.datasource.username=produser
3 spring.datasource.password=prodpassword
4 logging.level.root=ERROR

```

4. Conditional Beans Based on Profiles:

You can define beans that will only be created for specific profiles using the `@Profile` annotation.

Example:



```

10 @Configuration new *
11 @Profile("dev")
12 public class DevDataBaseConfig {
13
14     @Bean new *
15     public DataSource devDataSource() {
16         //config for development
17         return new DriverManagerDataSource();
18     }
19
20 }
21
10 @Configuration new *
11 @Profile("prod")
12 public class ProdDataBaseConfig {
13
14     @Bean new *
15     public DataSource prodDataSource() {
16         //Configuration for production
17         return new DriverManagerDataSource();
18     }
19
20 }
21

```

5. Combining Profiles:

You can also combine multiple profiles. For example, you may have a staging profile that incorporates settings from both dev and prod.

In `application.properties` `spring.profiles.active=dev,staging`

Spring Profiles significantly enhance the way applications are configured and manage different environments efficiently. By defining separate configurations per profile, you minimize the risk of misconfigurations during deployment.

6) BUILDING SPRING BOOT APPLICATIONS

Building Spring Boot applications involves packaging your application into a deployable format, typically as a JAR (Java Archive) or WAR (Web Application Archive). Spring Boot makes it simple to bundle all necessary components and dependencies.

Steps to Build a Spring Boot Application

1. Build Tool Setup:

Most Spring Boot applications use either Maven or Gradle. Ensure that your project has a proper build configuration (pom.xml for Maven or build.gradle for Gradle).

2. Maven Build:

In your pom.xml, make sure to include the Spring Boot Maven plugin:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

To build your project, you can run the following command in the terminal (bash):

```
mvn clean package
```

3. Gradle Build:

In your build.gradle, apply the Spring Boot plugin (groovy):

```
plugins {
    id 'org.springframework.boot' version '2.x.x' // Replace with your Spring Boot version
    id 'io.spring.dependency-management' version '1.0.11.RELEASE'
    id 'java'
}
```

To build your project (bash):

```
./gradlew build
```

This will generate the JAR file in the build/libs directory.

4. Executing the Application:

After building, you can run the application using the following command (bash):

```
java -jar target/your-application-name.jar
```

Ensure you replace your-application-name.jar with the actual name of the JAR file you built.

5. Creating a WAR File (Optional):

If you want to deploy your application as a WAR file (typically for traditional servlet containers), modify your `pom.xml` or `build.gradle` as follows:

For Maven (xml):

```
<packaging>war</packaging>
```

For Gradle (groovy):

apply plugin: 'war'

Build the WAR using the same build commands as above, and the WAR file will be found in the respective output directory.

6. Running from IDE:

Most IDEs allow you to run Spring Boot applications directly from the IDE. Simply run the main method or use the built-in run configurations.

Example Project Structure Post-Build

After building your application, the directory structure might look like this (for a JAR file):

your-app/

```
├── target/
│   └── your-application-name.jar
├── src/
│   └── ...
├── pom.xml    (if using Maven)
└── build.gradle (if using Gradle)
```

Building a Spring Boot application is straightforward due to the built-in support for packaging, whether as a JAR or a WAR file. The simplicity of running applications from the command line and IDEs facilitates rapid development and deployment.

7) CONTAINERIZING SPRING BOOT APPLICATIONS

Containerization allows you to encapsulate your Spring Boot application along with its dependencies, ensuring that it runs consistently across different computing environments. Docker is a widely used platform for containerization.

Steps to Containerize a Spring Boot Application

1. Install Docker:

Ensure you have Docker installed on your machine. You can download it from the Docker website.

2. Create a Dockerfile:

In the root of your Spring Boot project, create a file named `Dockerfile`. This file contains the instructions to build your Docker image. Here's an example Dockerfile for a JAR-based Spring Boot application:

```
# Use a base image with Java 17
FROM openjdk:17-jdk-slim

# Set the working directory in the container
WORKDIR /app

# Copy the JAR file into the container
COPY target/*.jar app.jar

# Expose the port your Spring Boot application runs on (typically 8080)
EXPOSE 8080

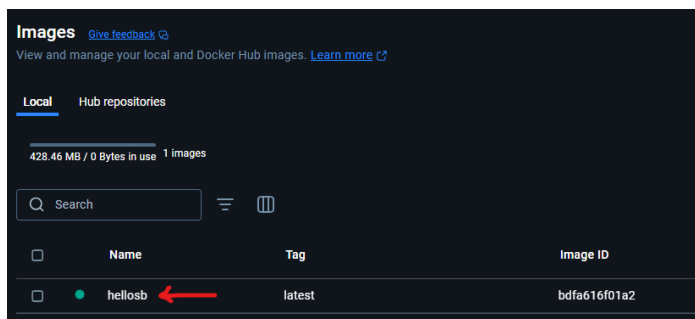
# Command to execute the application
ENTRYPOINT ["java", "-jar", "app.jar"]
```

3. Build the Docker Image:

Open a terminal in your project root directory (where the Dockerfile is located) and run:

```
docker build -t hellosb .
```

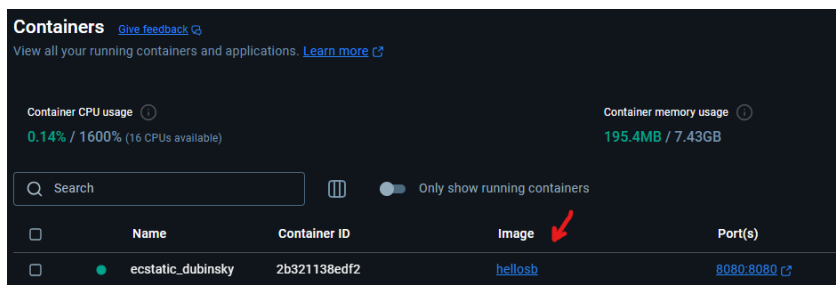
This command builds a Docker image using the specified Dockerfile, and you can replace your-image-name with your desired image name (hellosb).



4. Run the Docker Container:

Once the image is built successfully, you can run a container based on the image:

```
docker run -p 8080:8080 hellosb
```



This command maps port 8080 of the container to port 8080 on your host machine. Now your Spring Boot application will be accessible at <http://localhost:8080>.

Check browser: <http://localhost:8080/hello>

Hello, Spring Boot!

5. Manage Docker Containers:

You can list running containers using (bash): `docker ps`

To stop and remove a container (bash):

- `docker stop <container-id>` (our case it is: 2b321138ed)
- `docker rm <container-id>` (our case it is: 2b321138ed)

To remove an image (bash): `docker rmi your-image-name` (our case it is: hellosb)

6. Using Docker Compose (Optional):

If your application interacts with other services (like databases), you can manage multi-container applications using Docker Compose.

Place the `compose.yml` file: Put the `compose.yml` file in the same directory as your `Dockerfile` and your target directory (or in the root of your project).

Create a `docker-compose.yml` file:

```

1  version: '3.8'
2  services:
3    app:
4      build:
5        context: .
6        dockerfile: Dockerfile
7      ports:
8        - "8080:8080"
9      # If your application needs environment variables:
10     # environment:
11     #   - SPRING_PROFILES_ACTIVE=production
12     #   - DATABASE_URL=...

```

Build and run the application:

Open a terminal or command prompt in the directory containing the `compose.yml` file.

Run the following command: `docker-compose up --build`

`docker-compose up` starts the services defined in the `compose.yml` file.

`--build` ensures that the Docker image is built if it doesn't exist or if the `Dockerfile` has changed.

Stop the application:

To stop the running containers, press `Ctrl+C` in the terminal or run `docker-compose down`.

Containerization with Docker allows you to deploy your Spring Boot applications in a consistent environment across various platforms. By encapsulating your app and its dependencies, you simplify the deployment process and achieve better scalability.

SUMMARY

Topics we covered in our exploration of Spring Boot:

- 1) Booting from the Web: Set up a Spring Boot project and created a simple REST endpoint.
- 2) Understanding the Project: Familiarized ourselves with the project structure, including important components like the main application class, controllers, and resources.
- 3) Understanding Auto-Configuration: Explored how Spring Boot auto-configures application settings based on classpath dependencies, simplifying setup.
- 4) Configuration in Spring Boot: Discussed various ways to configure your application using `application.properties`, `application.yml`, environment variables, and Java configuration classes.
- 5) Spring Profiles in Boot: Utilized profiles to manage environment-specific configurations for development, testing, and production.
- 6) Building Spring Boot Applications: Covered how to build a Spring Boot application using Maven or Gradle and package it as a JAR or WAR file.
- 7) Containerizing Spring Boot Applications: Learned how to create a Docker image for a Spring Boot application, enabling consistent deployment across different environments.

These steps provide a solid foundation for developing, configuring, and deploying Spring Boot applications effectively.