

Spring Boot Actuator

Developing and Managing an application are the two most important aspects of the application's life cycle. It is very crucial to know what's going on beneath the application. Also when we push the application into production, managing it gradually becomes critically important. Therefore, it is always recommended to monitor the application both while at the development phase and at the production phase.

For the same use case, Spring Boot provides an actuator dependency that can be used to monitor and manage your Spring Boot application, By `/actuator` and `/actuator/health` endpoints you can achieve the purpose of monitoring.

- With the help of Spring Boot, we can achieve the above objectives.
- Spring Boot's 'Actuator' dependency is used to monitor and manage the Spring web application.
- We can use it to monitor and manage the application with the help of HTTP endpoints or with the JMX.

Advantages of Actuator the Application

- 1 It increases customer satisfaction.
- 2 It reduces downtime.
- 3 It boosts productivity.
- 4 It improves Cybersecurity Management.
- 5 It increases the conversion rate.

1. Configuration for Actuator

In order to use hibernate validators, these configurations are necessary in your Spring Boot project.

1.1 Dependency for Actuator

To use the 'Actuator' add the following dependency in your application's project settings file.

Dependency configuration for both Maven and Gradle build system.

Maven -> pom.xml

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

1.2 Application Properties configuration for Actuator

There more configurations available for Actuator, few of them are listed:

- You can also change the default endpoint by adding the following in the application.properties file.

management.endpoints.web.base-path=/details

- Including IDs/Endpoints

By default, all IDs are set to false except for 'health'. To include an ID, use the following property in the

application.properties file.

```
management.endpoint.<id>.enabled
```

Example -> `management.endpoint.metrics.enabled=true`

- List down all IDs that you want to include which are separated by a comma.

```
management.endpoints.web.exposure.include=metrics,info
```

- Include only metrics and info IDs and will exclude all others ('health' too).

To add/include all ID information about your application, you can do it in the application.properties file by simply adding the following –

```
management.endpoints.web.exposure.include=*
```

- Excluding IDs/Endpoints

To exclude an ID or endpoint, use the following property and list out the respective IDs separated by a comma in the application.properties file.

```
management.endpoints.web.exposure.exclude
```

Example ->

```
management.endpoints.web.exposure.exclude=info
```

Folder structure for projects

The below image demonstrates the picture of how your project must look

2. Implementing the project

2.1 Entity

UserEntity.java (Entity class representing the model data) is explained below:

- This class acts as a simple java bean whose

properties are returned as JSON response by the REST API's get() method.

- 'Lombok' library is used to generate GETTER/SETTER methods automatically at runtime using '@Data' annotation.
- '@RequiredArgsConstructor' annotation is used to generate a zero-argument constructor and if final or '@NonNull' fields are present, then respective arguments constructor is created.
- To add the 'Lombok' library in your application, add the following dependency in your application's project build.
- '@Component' annotation is used so that this bean automatically gets registered in Spring's application context.

```
package gfg;
```

```
import lombok.Data;
```

```
import lombok.RequiredArgsConstructor;
```

```
import org.springframework.stereotype.Component;
```

```
@Component
```

```
@Data
```

```
@RequiredArgsConstructor
```

```
public class UserEntity {
```

```
    String id = "1";
```

```
    String name = "Darshan.G.Pawar";
```

```
    String userName = "@drash";
```

```
    String email = "drash@geek";
```

```
    String pincode = "422-009";
```

```
}
```

2.2 Controller

RESTfulController.java (A REST API controller) for defining APIs and testing the program.

This controller's get() method uses the UserEntity bean to return JSON response. UserEntiy bean is outsourced through '@Autowired' annotation which was registered in Spring's application context.

```
package gfg;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/get")
public class RESTfulController {

    @Autowired
    UserEntity entity;

    @GetMapping("/data")
    public UserEntity getEntity(){
        return entity;
    }
}
```

3. Testing Actuator APIs

3.1 Controller APIs

Here, the JSON Formatter Chrome extension is used to automatically parse the JSON body. Further, it will be

required to work with 'Actuator'.

3.2 Working with Spring Boot Actuator APIs

To access the 'Actuator' services, you will have to use the HTTP endpoint as it becomes reliable to work with.

3.2.1 /actuator

It's simple just hit the default endpoint '/actuator', ensure that your Application is running.

Example:

You can also change the default endpoint by adding the following in the application.properties file.

```
management.endpoints.web.base-path=/details
```

3.2.2 /actuator/health

You can click on these above links and see the respective information. Additionally, you can activate other Actuator IDs and use them after '/actuator' to see more information. For example, 'health' ID is activated by default. Therefore you can click the link in the image or directly use 'http://localhost:8080/actuator/health'.

'UP' means the application's health is good. There are a total of 25 IDs out of which the commonly used are listed out here –

ID	Description
beans	Displays a complete list of all the Spring beans in your application.

By default, all IDs are set to false except for 'health'. To include an ID, use the following property in the application.properties file.

```
management.endpoint.<id>.enabled
```

Example -> management.endpoint.metrics.enabled=true

OR, you can just list down all IDs that you want to include which are separated by a comma.

```
management.endpoints.web.exposure.include=metrics,info
```

This will include only metrics and info IDs and will exclude all others ('health' too). To add/include all ID information about your application, you can do it in the application.properties file by simply adding the following

—

```
management.endpoints.web.exposure.include=*
```

Output: All the IDs or the Endpoint are now enabled

Excluding IDs/Endpoints

To exclude an ID or endpoint, use the following property and list out the respective IDs separated by a comma in the application.properties file.

```
management.endpoints.web.exposure.exclude
```

Example ->

```
management.endpoints.web.exposure.exclude=info
```

Use '*' in place of IDs in property to exclude all the IDs or endpoints.

Notes:

- 1 Before setting the management.endpoints.web.exposure.include,

ensure that the exposed actuators do not contain sensitive information.

- 2 They should be secured by placing them behind a firewall or are secured by something like Spring Security.

Day 1: Spring Boot Actuator

1. Monitoring and Managing Spring Boot Applications

- **Spring Boot Actuator Overview:** Provides production-ready features to monitor and manage Spring Boot applications. It includes several built-in endpoints for health checks, metrics, application info, etc.
- **Enabling Actuator:**
 - Add the dependency to pom.xml or build.gradle.xml

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>
```
- **Accessing Built-in Endpoints:**
 - **/actuator:** Base path for all Actuator endpoints.
 - **Health:** /actuator/health - Displays application health status.
 - **Info:** /actuator/info - Displays custom information.
 - **Metrics:** /actuator/metrics - Shows various application metrics.
 - **Environment:** /actuator/env - Exposes the application environment properties.
- **Securing Actuator Endpoints:**
 - Configure security to restrict access to sensitive endpoints.
 - Example:yaml

```
management:  
  endpoints:  
    web:  
      exposure:  
        include: "health", "info"  
  security:  
    roles: "ACTUATOR"
```

-

2. Using Built-in Endpoints

- **Health Endpoint:**

- Shows the application health status.
- Can be customized by implementing HealthIndicator.java
- @Component
- public class CustomHealthIndicator implements HealthIndicator {
- @Override
- public Health health() {
- if (check()) {
- return Health.up().build();
- }
- return Health.down().withDetail("Error", "Service is down").build();
- }
- }
- }
-

- **Info Endpoint:**

- Displays arbitrary application information.
- Configure in application.properties:properties
- info.app.name=My Spring Boot Application
- info.app.version=1.0.0
-

- **Metrics Endpoint:**

- Provides various metrics such as memory usage, JVM statistics.
- Access specific metrics:plaintext
- /actuator/metrics/jvm.memory.used
-

3. Customizing Actuator Endpoints

- **Custom Endpoints:**

- Create a custom endpoint by extending AbstractEndpoint.java

```

    @Endpoint(id = "custom")
    public class CustomEndpoint {
        @ReadOperation
        public String custom() {
            return "Custom Endpoint";
        }
    }

```

- Register the endpoint:java

```

    @Configuration
    public class CustomEndpointConfig {
        @Bean
        public CustomEndpoint customEndpoint() {
            return new CustomEndpoint();
        }
    }

```

- **Configuring Endpoint Exposure:**

- Control which endpoints are exposed via application.properties:properties
- management.endpoints.web.exposure.include=health,info,custom

- **Extending Built-in Endpoints:**

- Add custom data to existing endpoints (e.g., health indicators).

Spring Boot Testing

There are so many different testing approaches in Spring Boot used for deploying the application server. Testing in Spring Boot is an important feature of software development which ensures that your

application behaves as per the requirement and meets all the testing criteria. Spring Boot provides a robust testing framework that supports **Unit Testing**, **Integration Testing**, and **End-to-End Testing**.

1. Unit Tests:

- The Focus is on testing individual components or units of the code in isolation.
- Use tools like JUnit and Mockito for the writing unit tests.

2. Integration Tests:

- Test the interactions between the multiple components or modules.
- Ensures that different parts of the application work together perfectly.
- Typically involves testing repositories, services, and controllers.

3. End-to-End (E2E) Tests:

- Test the entire application from the end to end simulating real user scenarios.
- Involve testing application's behavior through its external interfaces.
- Use tools like Selenium for the web applications.

Testing Annotations in Spring Boot

@SpringBootTest:

- Indicates that the annotated class is the Spring Boot test.
- The Loads the complete application context.

```
@SpringBootTest
public class MySpringBootTest {
    // Test methods go here
}
```

@RunWith(SpringRunner.class):

- The Specifies the class to run the tests.
- The SpringRunner is the alias for SpringJUnit4ClassRunner.

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class MySpringBootTest {
    // Test methods go here
}
```

@MockBean:

- The Mocks a Spring Bean for testing purposes.
- Useful for the isolating the unit of the code being tested.

```
@SpringBootTest
public class MyServiceTest {
    @Autowired
    private MyService myService;
    @MockBean
    private ExternalService externalService;
    // Test methods go here
}
```

@Test:

- The Denotes a test method.
- Executed when running the test class.

```
@Test
public void myUnitTest() {
    // Test logic goes here
}
```

Maven Dependencies for Testing

To get started with the testing in the Spring Boot project you need to include the following dependencies in the your pom.xml file:

```
<dependencies>
  <!-- Spring Boot Starter Test -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

JUnit Testing Dependency

The Spring Boot uses JUnit as the default testing framework. The spring-boot-starter-test dependency already includes the JUnit so you don't need to add it explicitly.

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>
```

1. @SpringBootTest (Integration Testing)

The **@SpringBootTest** is a core annotation in Spring Boot for the integration testing. It can be used to specify the configuration of ApplicationContext for the

your tests.

Let's consider a simple controller class:

```
@RestController
public class MyController {
    @GetMapping("/hello")
    public String hello() {
        return "Hello, World!";
    }
}
```

Now, let's write a test class:

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class MyControllerTest {
    @Autowired
    private MockMvc mockMvc;
    @Test
    public void testHelloEndpoint() throws Exception {
        mockMvc.perform(get("/hello"))
            .andExpect(status().isOk())
            .andExpect(content().string("Hello, World!"));
    }
}
```

Explanation:

- `@RunWith(SpringRunner.class)`: The Specifies the runner that JUnit should use to run the tests. The `SpringRunner` is the new name for the `SpringJUnit4ClassRunner`.
- `@SpringBootTest`: Used to indicate that the

annotated class is the Spring Boot test class.

Property File Configuration:

For testing, you might want to have a separate the application-test.properties file. The Spring Boot automatically picks up properties from this file during the tests.

2. @TestConfiguration (Test Configuration)

The @TestConfiguration is used to the specify additional configuration for the tests. It is often used to define @Bean methods for the test-specific beans.

```
@TestConfiguration
public class TestConfig {
    @Bean
    public MyService myService() {
        return new MyService();
    }
}
```

3. @MockBean (Mocking)

The @MockBean is used to mock a bean of the specific type, making it convenient to test components that depend on bean.

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class MyServiceTest {
    @Autowired
    private MyService myService;
    @MockBean
    private ExternalService externalService;
    @Test
    public void testServiceMethod() {
```



```

        // Define behavior for the mocked external service
        when(externalService.getData()).thenReturn("Mocked
Data");
        // Now test the method from MyService that uses
externalService
        String result = myService.processData();
        assertEquals("Processed: Mocked Data", result);
    }
}

```

4. @WebMvcTest (Unit Testing)

The @WebMvcTest is used for the testing the controllers in the Spring MVC application. It focuses only on MVC components.

```

@RunWith(SpringRunner.class)
@WebMvcTest(MyController.class)
public class MyControllerWebMvcTest {
    @Autowired
    private MockMvc mockMvc;
    @Test
    public void testHelloEndpoint() throws Exception {
        mockMvc.perform(get("/hello"))
            .andExpect(status().isOk())
            .andExpect(content().string("Hello, World!"));
    }
}

```

5. @DataJpaTest (Integration Testing)

The @DataJpaTest is used for the testing JPA repositories. It focuses only on the JPA components.

```

@RunWith(SpringRunner.class)
@DataJpaTest
public class MyRepositoryTest {
    @Autowired

```

```

private MyRepository myRepository;
@Test
public void testRepositoryMethod() {
    MyEntity entity = new MyEntity();
    entity.setData("Test Data");
    myRepository.save(entity);
    MyEntity savedEntity =
myRepository.findById(entity.getId()).orElse(null);
    assertNotNull(savedEntity);
    assertEquals("Test Data", savedEntity.getData());
}
}

```

Additional Annotations

- `@AutoConfigureMockMvc`: Used with the `@SpringBootTest` to automatically configure a `MockMvc` instance.
- `@DirtiesContext`: Indicates that the `ApplicationContext` associated with test is dirty and should be closed after the test.
- `@Transactional`: Used to indicate that a test-managed transaction should be used.

Conclusion

The Spring Boot's testing support combined with the popular testing libraries like JUnit and Mockito enables the developers to create comprehensive test suites for their applications. By using the appropriate testing annotations and strategies you can ensure the reliability and correctness of the your Spring Boot applications.

Spring Boot Testing

Introduction to Spring Boot Testing

Testing is an essential part of the software development process, ensuring that code functions as expected and helping to catch bugs early. Spring Boot provides a robust testing framework that simplifies writing both unit and integration tests.

Testing Strategies

1 Unit Testing:

- Tests individual components or classes in isolation.
- Fast and focuses on small parts of the application.
- Commonly uses mocking to simulate dependencies.

2 Integration Testing:

- Tests the interaction between components or modules.
- Ensures that different parts of the system work together as expected.
- Typically involves starting the application context and using real dependencies.

3 End-to-End Testing:

- Tests the complete application flow from start to finish.
- Involves running the application in a production-like environment.
- Simulates user interactions to verify the entire system.

Spring Boot Test Utilities

- **@SpringBootTest**: Annotation to load the full application context.
- **@WebMvcTest**: Annotation to test Spring MVC controllers.
- **@DataJpaTest**: Annotation to test JPA repositories.
- **@MockBean**: Annotation to add mock beans to the application context.

Writing Unit Tests with JUnit and Mockito

JUnit

JUnit is a widely-used testing framework for Java. It provides annotations and assertions to define and verify test cases.

Common Annotations:

- **@Test**: Marks a method as a test case.
- **@BeforeEach**: Runs before each test case.
- **@AfterEach**: Runs after each test case.
- **@BeforeAll**: Runs once before all test cases.
- **@AfterAll**: Runs once after all test cases.

Example:

```
java
```

```

@SpringBootTest
public class MyServiceTests {
    @Autowired
    private MyService myService;

    @Test
    public void testService() {
        String result = myService.getValue();
        assertEquals("ExpectedValue", result);
    }
}

```

Mockito

Mockito is a popular framework for creating mock objects. It allows simulating the behavior of dependencies in isolation.

Common Annotations:

- `@Mock`: Creates a mock instance.
- `@InjectMocks`: Injects mock instances into the tested object.

Example:

java

```

@ExtendWith(MockitoExtension.class)
public class MyServiceTests {
    @Mock
    private MyRepository myRepository;

    @InjectMocks
    private MyService myService;

    @Test
    public void testService() {
        when(myRepository.findById(1L)).thenReturn(Optional.of(new
MyEntity("TestValue")));
        String result = myService.getValue(1L);
        assertEquals("TestValue", result);
    }
}

```

Integration Testing with Spring Boot

Integration tests verify that different parts of the application work together correctly. Spring Boot provides annotations and utilities to simplify integration testing.

@SpringBootTest

This annotation is used to load the full application context for testing.

Example:

```
@SpringBootTest
public class ApplicationTests {
    @Autowired
    private MockMvc mockMvc;

    @Test
    public void testEndpoint() throws Exception {
        mockMvc.perform(get("/api/endpoint"))
            .andExpect(status().isOk())
            .andExpect(content().string("ExpectedResponse"));
    }
}
```

@WebMvcTest

This annotation is used to test Spring MVC controllers without starting the full application context. It focuses on the web layer only.

Example:

```
@WebMvcTest(MyController.class)
public class MyControllerTests {
    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private MyService myService;

    @Test
    public void testGet() throws Exception {
        when(myService.getValue()).thenReturn("TestValue");
        mockMvc.perform(get("/api/value"))
            .andExpect(status().isOk())
            .andExpect(content().string("TestValue"));
    }
}
```

@DataJpaTest

This annotation is used to test JPA repositories. It configures an in-memory database and scans for @Entity classes and Spring Data JPA repositories.

Example:

```
@DataJpaTest
public class MyRepositoryTests {
    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private MyRepository myRepository;

    @Test
    public void testFindByName() {
        MyEntity entity = new MyEntity();
        entity.setName("TestName");
        entityManager.persist(entity);

        MyEntity found = myRepository.findByName("TestName");
        assertEquals("TestName", found.getName());
    }
}
```

Summary

- **Spring Boot Testing Overview:** Provides tools and annotations to simplify both unit and integration testing.
- **Unit Testing:** Use JUnit and Mockito to test individual components in isolation.
- **Integration Testing:** Use @SpringBootTest, @WebMvcTest, and @DataJpaTest to test the interaction between components.
- **Best Practices:** Isolate unit tests, use mock objects for dependencies, and verify interactions and outcomes. For integration tests, focus on the interactions between components and the correctness of the application context.

Microservices with Spring Boot:

Microservices is an architectural approach to build a

collection of logic, data layers, and loosely coupled applications. Every microservices deals with one business function end-to-end independently from other microservices. Microservices present simple and understandable APIs to communicate with each other through lightweight common protocols such as HTTP. With the increasing demand for Microservices Architecture Patterns in the industry the popularity of **Spring Boot** is also increasing because when it comes to **Microservices Development**, **Spring Boot** is the first choice of every developer. Spring Boot is a microservice-based framework that makes a production-ready application in significantly less time. By using Spring Boot, you can make your microservices smaller and it will run faster. For this reason, Spring Boot has become the standard for Java microservices. In this article, we will create a simple Microservice using **Spring Boot**.

Step-by-Step Guide

Step 1: Create a New Spring Boot Project in Spring Initializr

To create a new Spring Boot project, please refer to [How to Create a Spring Boot Project in Spring Initializr and Run it in IntelliJ IDEA](#). For this project choose the following things

- Project: Maven
- Language: Java
- Packaging: Jar
- Java: 17

Please choose the following dependencies while creating the project.

- Spring Boot DevTools
- Spring Data JPA
- MySQL Driver
- Spring Web

Generate the project and run it in IntelliJ IDEA by referring to the above article.

Note: We have used the **MySQL** database in this project.

Step 2: Create Schema in MySQL Workbench and Put Some Sample Data

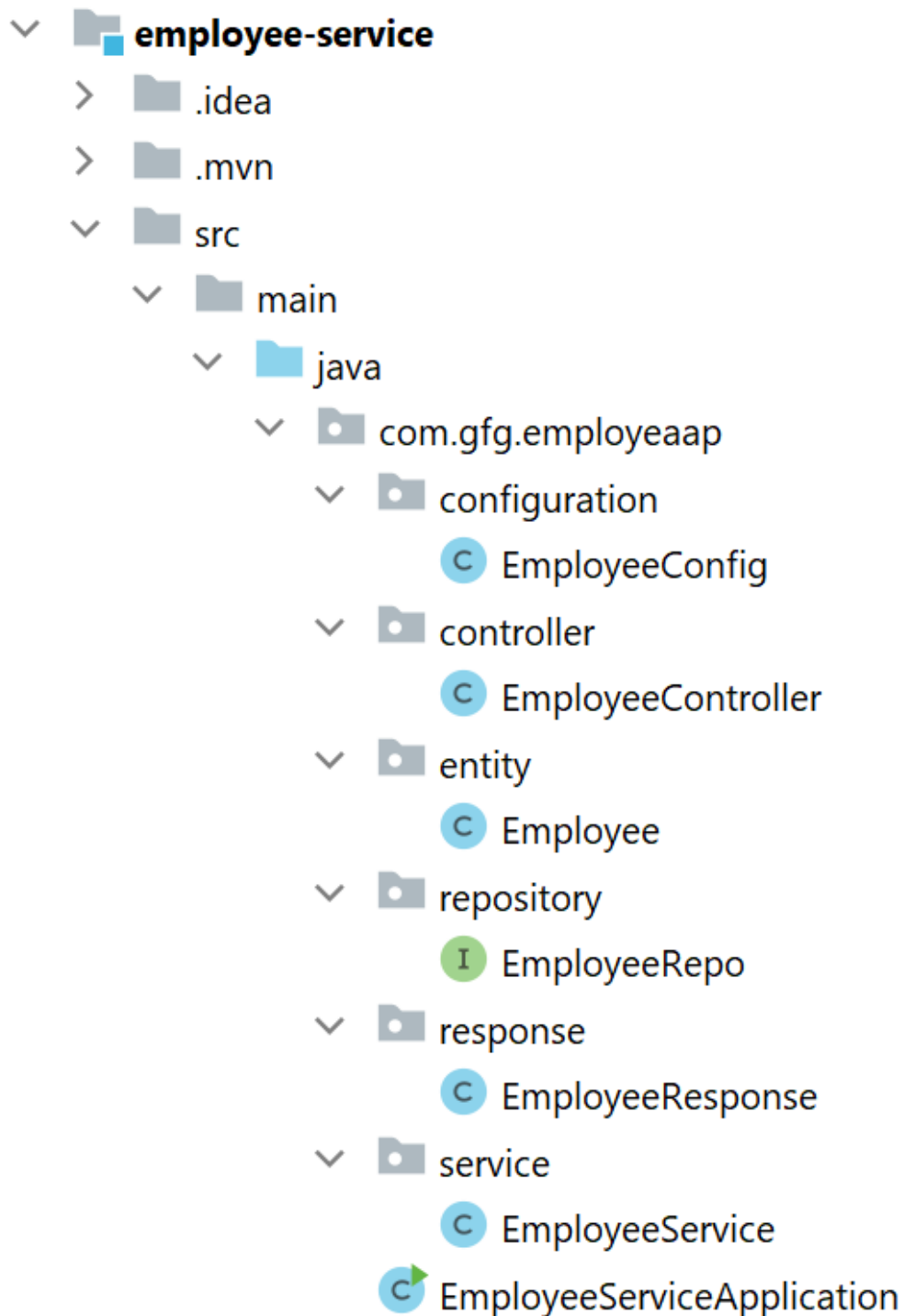
Go to your **MySQL Workbench** and create a schema named **gfgmicroservicesdemo** and inside that create a table called **employee** and put some sample data as shown in the below image. Here we have created 4 columns and put some sample data.

- 1 id
- 2 name
- 3 email
- 4 age

The screenshot shows a database management interface. On the left, a 'Navigator' pane displays a tree view of the database schema. The 'gfgmicroservicesdemo' database is expanded, showing 'Tables' with 'employee' selected. Below the tree, the 'Table: employee' is detailed with its columns: 'id' (int AI PK), 'name' (varchar(45)), 'email' (varchar(45)), and 'age' (varchar(45)). The main pane shows a SQL query: '1 • SELECT * FROM gfgmicroservicesdemo.employee;'. Below the query, a 'Result Grid' displays the data for the 'employee' table. The grid has four columns: 'id', 'name', 'email', and 'age'. It contains two rows of data: one for 'Amiya' (id: 1, email: ar@gmail, age: 25) and one for 'Asish' (id: 2, email: asis@gmail, age: 30). The 'id' column is highlighted in blue for the first row.

id	name	email	age
1	Amiya	ar@gmail	25
2	Asish	asis@gmail	30

Now we are going to fetch Employee Data from Employee Table in our Spring Boot project. To do it refer to the following steps. Before moving to IntelliJ IDEA let's have a look at the complete project structure for our Microservices.



Step 3: Make Changes in Your application.properties File

Now make the following changes in your [application.properties](#) file.

```
spring.datasource.url=jdbc:mysql://localhost:3306/
```

```
gfgmicroservicesdemo
```

```
spring.datasource.username=put your username here
```

```
spring.datasource.password=put your password here
```

Step 4: Create Your Entity/Model Class

Go to the **src > main > java > entity** and create a class Employee and put the below code. This is our model class.

```
package com.gfg.employeeaap.entity;

import jakarta.persistence.*;

@Entity
@Table(name = "employee")
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private int id;

    @Column(name = "name")
    private String name;

    @Column(name = "email")
    private String email;

    @Column(name = "age")
    private String age;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }
}
```

```

    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getAge() {
        return age;
    }

    public void setAge(String age) {
        this.age = age;
    }
}

```

Step 5: Create Your Repository Interface

Go to the **src > main > java > repository** and create an interface **EmployeeRepo** and put the below code. This is our repository where we write code for all the database-related stuff.

```

package com.gfg.employeaap.repository;

import com.gfg.employeaap.entity.Employee;
import org.springframework.data.jpa.repository.JpaRepository;

public interface EmployeeRepo extends JpaRepository<Employee,

```

```
Integer> {
```

```
}
```

Note: Please refer to this article to know more about [JpaRepository](#).

Step 6: Create an EmployeeResponse Class

Go to the **src > main > java > response** and create a class EmployeeResponse and put the below code.

```
package com.gfg.employeaap.response;
```

```
public class EmployeeResponse {
```

```
    private int id;
```

```
    private String name;
```

```
    private String email;
```

```
    private String age;
```

```
    public int getId() {
```

```
        return id;
```

```
    }
```

```
    public void setId(int id) {
```

```
        this.id = id;
```

```
    }
```

```
    public String getName() {
```

```
        return name;
```

```
    }
```

```
    public void setName(String name) {
```

```
        this.name = name;
```

```
    }
```

```
    public String getEmail() {
```

```
        return email;
```

```
    }
```

```

    public void setEmail(String email) {
        this.email = email;
    }

    public String getAge() {
        return age;
    }

    public void setAge(String age) {
        this.age = age;
    }
}

```

Step 7: Create Your Service Class

Go to the **src > main > java > service** and create a class `EmployeeService` and put the below code. This is our service class where we write our business logic.

```

package com.gfg.employeeaap.service;

import com.gfg.employeeaap.entity.Employee;
import com.gfg.employeeaap.repository.EmployeeRepo;
import com.gfg.employeeaap.response.EmployeeResponse;
import org.modelmapper.ModelMapper;
import org.springframework.beans.factory.annotation.Autowired;

import java.util.Optional;

public class EmployeeService {

    @Autowired
    private EmployeeRepo employeeRepo;

    @Autowired
    private ModelMapper mapper;

    public EmployeeResponse getEmployeeById(int id) {
        Optional<Employee> employee = employeeRepo.findById(id);
    }
}

```

```

        EmployeeResponse employeeResponse = mapper.map(employee,
EmployeeResponse.class);
        return employeeResponse;
    }
}

```

Step 8: Create an Employee Controller

Go to the **src > main > java > controller** and create a class `EmployeeController` and put the below code. Here we are going to create an endpoint “**/employees/{id}**” to find an employee using id.

```

package com.gfg.employeeaap.controller;

import com.gfg.employeeaap.response.EmployeeResponse;
import com.gfg.employeeaap.service.EmployeeService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class EmployeeController {

    @Autowired
    private EmployeeService employeeService;

    @GetMapping("/employees/{id}")
    private ResponseEntity<EmployeeResponse>
getEmployeeDetails(@PathVariable("id") int id) {
        EmployeeResponse employee =
employeeService.getEmployeeById(id);
        return ResponseEntity.status(HttpStatus.OK).body(employee);
    }
}

```

Step 9: Create a Configuration Class

Go to the **src > main > java > configuration** and create a class `EmployeeConfig` and put the below code.

```
package com.gfg.employeeaap.configuration;

import com.gfg.employeeaap.service.EmployeeService;
import org.modelmapper.ModelMapper;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class EmployeeConfig {

    @Bean
    public EmployeeService employeeBean() {
        return new EmployeeService();
    }

    @Bean
    public ModelMapper modelMapperBean() {
        return new ModelMapper();
    }
}
```

Note: You may refer to these two articles

- [Spring @Configuration Annotation with Example](#)
- [Spring @Bean Annotation with Example](#)

Before running the Microservice below is the complete **pom.xml** file. Please cross-verify if you have missed some dependencies

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```



```
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                    https://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>3.0.2</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>com.gfg.employeaap</groupId>
<artifactId>employee-service</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>employee-service</name>
<description>Employee Service</description>
<properties>
  <java.version>17</java.version>
</properties>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
  </dependency>
  <dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <scope>runtime</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
```

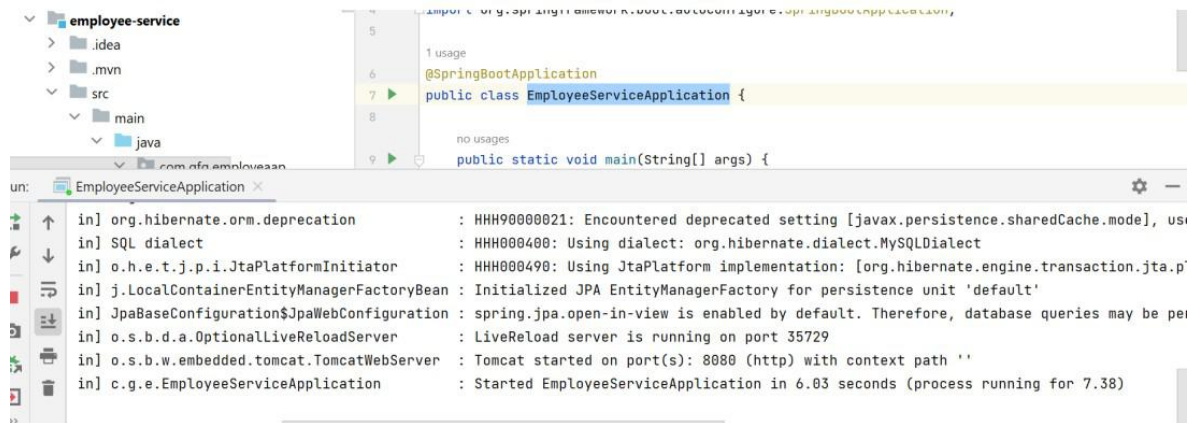
```
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.modelmapper</groupId>
        <artifactId>modelmapper</artifactId>
        <version>3.1.1</version>
    </dependency>
</dependencies>

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

</project>
```

Step 10: Run Your Employee Microservice

To run your Employee Microservice **src > main > java > EmployeeServiceApplication** and click on the Run button. If everything goes well then you may see the following screen in your console. Please refer to the below image.



Step 11: Test Your Endpoint in Postman

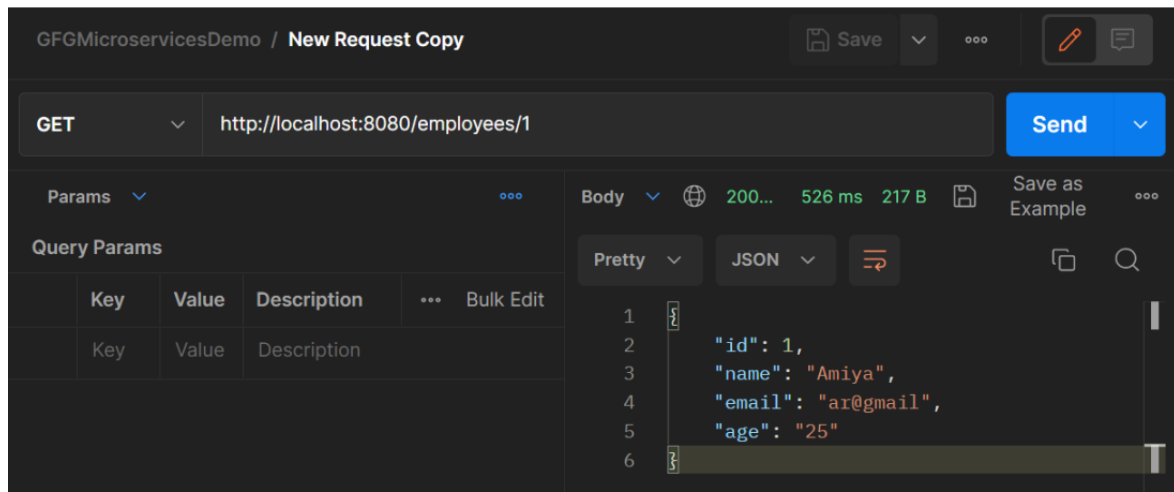
Now open Postman and hit the following URL:

GET: <http://localhost:8080/employees/1>

And you can see the following response:

```
{
  "id": 1,
  "name": "Amiya",
  "email": "ar@gmail",
  "age": "25"
}
```

Please refer to the below image:



This is how we have built our **Employee Microservice** with the help of Java and Spring Boot. And you can also design all your endpoints and write all the business logic, database logic, etc in the corresponding file.

Day 3: Microservices with Spring Boot

Introduction to Microservices Architecture

Microservices Principles

1 Single Responsibility:

- Each microservice is designed to perform a specific business function.
- Promotes high cohesion and low coupling.

2 Decentralization:

- Services are independently deployable and scalable.
- Allows using different technologies and databases as needed.

3 Scalability:

- Individual services can be scaled horizontally as demand increases.
- Better resource utilization.

Advantages

- **Independent Deployment:** Services can be deployed independently, reducing the risk and effort of deployment.
- **Technology Diversity:** Different technologies can be used for different services.
- **Fault Isolation:** Failure in one service does not necessarily impact others.

- **Scalability:** Services can be scaled independently based on their needs.

Challenges

- **Complexity:** Managing multiple services introduces complexity.
- **Inter-Service Communication:** Requires reliable communication mechanisms.
- **Data Consistency:** Maintaining data consistency across services can be challenging.

Building Microservices with Spring Boot

Creating a Microservice

- **Project Setup:**
 - Create a new Spring Boot project using Spring Initializr with dependencies like Web, Actuator, and specific starters.

• xml

```
<dependency>
```

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

- **Defining REST Controllers:**

- Create REST endpoints for the microservice.

• java

```
@RestController
```

- ```
@RequestMapping("/api")
```
- ```
public class MyServiceController {
```
- 
- ```
@GetMapping("/data")
```
- ```
public ResponseEntity<String> getData() {
```
- ```
    return ResponseEntity.ok("Service Data");
```
- ```
}
```
- ```
}
```
-

Service Discovery

- **Eureka Server:**

- Add dependencies for Eureka Server.

- xml

```
<dependency>
```

- ```
<groupId>org.springframework.cloud</groupId>
```
- ```
<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
```
- ```
</dependency>
```
- 

- Application properties:

- properties

```
eureka.client.register-with-eureka=false
```

- ```
eureka.client.fetch-registry=false
```
- ```
server.port=8761
```
- 

- Main application class:

- java

```
@SpringBootApplication
```

- ```
@EnableEurekaServer
```
- ```
public class EurekaServerApplication {
```
- ```
    public static void main(String[] args) {
```
- ```
 SpringApplication.run(EurekaServerApplication.class, args);
```
- ```
    }
```
- ```
}
```
- 

- **Eureka Client:**

- Add dependencies for Eureka Client.

- xml

```
<dependency>
```

- ```
<groupId>org.springframework.cloud</groupId>
```
- ```
<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
```
- ```
</dependency>
```

- - Application properties:
- properties


```
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
```
- - Main application class:
- java


```
@SpringBootApplication
@EnableEurekaClient
public class MyServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyServiceApplication.class, args);
    }
}
```

Inter-Service Communication

REST

- **Using RestTemplate:**
 - Configure RestTemplate bean.
- java


```
@Configuration
public class RestTemplateConfig {
    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```
- - Use RestTemplate for inter-service calls.
- java

```

@Service
public class MyServiceClient {
    @Autowired
    private RestTemplate restTemplate;

    public String getData() {
        return restTemplate.getForObject("http://other-service/api/data",
String.class);
    }
}

```

- **Using WebClient:**

- Add dependencies for WebClient.
- xml

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>

```

- Configure WebClient bean.
- java

```

@Configuration
public class WebClientConfig {
    @Bean
    public WebClient.Builder webClientBuilder() {
        return WebClient.builder();
    }
}

```

- Use WebClient for inter-service calls.
- java

```

@Service
public class MyServiceClient {
    @Autowired

```


- private WebClient.Builder webClientBuilder;
-
- public String getData() {
- return webClientBuilder.build()
- .get()
- .uri("http://other-service/api/data")
- .retrieve()
- .bodyToMono(String.class)
- .block();
- }
- }
-

gRPC

- **Setup:**
 - Add dependencies for gRPC.
- xml


```

<dependency>
•   <groupId>io.grpc</groupId>
•   <artifactId>grpc-netty-shaded</artifactId>
•   <version>1.37.0</version>
• </dependency>
• <dependency>
•   <groupId>io.grpc</groupId>
•   <artifactId>grpc-protobuf</artifactId>
•   <version>1.37.0</version>
• </dependency>
• <dependency>
•   <groupId>io.grpc</groupId>
•   <artifactId>grpc-stub</artifactId>
•   <version>1.37.0</version>
• </dependency>
      
```
- **Define gRPC Service:**
 - Create a .proto file for the service definition.
- protobuf


```

syntax = "proto3";
      
```

-
- option java_package = "com.example.grpc";
- option java_multiple_files = true;
-
- service MyService {
- rpc getData(MyRequest) returns (MyResponse);
- }
-
- message MyRequest {
- string id = 1;
- }
-
- message MyResponse {
- string data = 1;
- }
-
-
- ○ Generate gRPC classes using the protoc compiler.

- **Implement gRPC Service:**

java

@GrpcService

- public class MyServiceImpl extends
- MyServiceGrpc.MyServiceImplBase {
-
- @Override
- public void getData(MyRequest request,
- StreamObserver<MyResponse> responseObserver) {
- MyResponse response =
- MyResponse.newBuilder().setData("Service Data").build();
- responseObserver.onNext(response);
- responseObserver.onCompleted();
- }
- }
-

- **gRPC Client:**

java

@Service

- public class MyGrpcClient {

-
- private final MyServiceGrpc.MyServiceBlockingStub stub;
-
- public MyGrpcClient(ManagedChannel channel) {
- stub = MyServiceGrpc.newBlockingStub(channel);
- }
-
- public String getData(String id) {
- MyRequest request = MyRequest.newBuilder().setId(id).build();
- MyResponse response = stub.getData(request);
- return response.getData();
- }
- }
-

Messaging

- **Using Spring Cloud Stream with RabbitMQ:**
 - Add dependencies for Spring Cloud Stream and RabbitMQ.
- xml


```

      <dependency>
      •     <groupId>org.springframework.cloud</groupId>
      •     <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
      •     </dependency>
      
```
- **Configure Message Channels:**

```

      java

      @Configuration

      •     public class MessagingConfig {
      •         @Bean
      •         public MessageChannel outputChannel() {
      •             return MessageChannels.direct().get();
      •         }
      •
      •         @Bean
      •         public MessageChannel inputChannel() {
      •             return MessageChannels.direct().get();
      •         }
      
```

- }
-

- **Producer:**

java

@Service

- public class MessageProducer {
- @Autowired
- private MessageChannel outputChannel;
-
- public void sendMessage(String message) {
-
- outputChannel.send(MessageBuilder.withPayload(message).build());
- }
- }
-

- **Consumer:**

java

@Service

- public class MessageConsumer {
-
- @StreamListener(target = Sink.INPUT)
- public void consumeMessage(String message) {
- System.out.println("Received message: " + message);
- }
- }
-

- **Application properties:**

properties

spring.cloud.stream.bindings.output.destination=myQueue

- spring.cloud.stream.bindings.input.destination=myQueue
-

Summary

- **Introduction to Microservices Architecture:** Principles, advantages, and challenges of microservices.
- **Building Microservices with Spring Boot:** Setting up microservices, defining REST controllers, and configuring service discovery with Eureka.
- **Inter-Service Communication:** Using REST (RestTemplate and WebClient), gRPC, and messaging with Spring Cloud Stream and RabbitMQ.

Spring Security Introduction

Spring Security is a framework which provides various security features like: authentication, authorization to create secure Java Enterprise Applications.

It is a sub-project of Spring framework which was started in 2003 by Ben Alex. Later on, in 2004, It was released under the Apache License as Spring Security 2.0.0.

It overcomes all the problems that come during creating non spring security applications and manage new server environment for the application.

This framework targets two major areas of application are authentication and authorization. Authentication is the process of knowing and identifying the user that wants to access.

Authorization is the process to allow authority to perform actions in the application.

We can apply authorization to authorize web request, methods and access to individual domain.

Technologies that support Spring Security Integration

Spring Security framework supports wide range of authentication models. These models either provided by third parties or framework itself. Spring Security supports integration

with all of these technologies.

- HTTP BASIC authentication headers
- HTTP Digest authentication headers
- HTTP X.509 client certificate exchange
- LDAP (Lighweight Directory Access Protocol)
- Form-based authentication
- OpenID authentication
- Automatic remember-me authentication
- Kerberos
- JOSSO (Java Open Source Single Sign-On)
- AppFuse
- AndroMDA
- Mule ESB
- DWR(Direct Web Request)

The beauty of this framework is its flexible authentication nature to integrate with any software solution. Sometimes, developers want to integrate it with a legacy system that does not follow any security standard, there Spring Security works nicely.

Advantages

Spring Security has numerous advantages. Some of that are given below.

- Comprehensive support for authentication and authorization.
- Protection against common tasks
- Servlet API integration
- Integration with Spring MVC
- Portability
- CSRF protection
- Java Configuration support

Spring Security History

In late 2003, a project **Acegi Security System for Spring** started with the intention to develop a Spring-based security system. So, a simple security system was implemented but not released officially. Developers used that code internally for their solutions and by 2004 about 20 developers were using that.

Initially, authentication module was not part of the project, around a year after, module was added and complete project was reconfigure to support more technologies.

After some time this project became a subproject of Spring framework and released as 1.0.0 in 2006.

in 2007, project is renamed to Spring Security and widely accepted. Currently, it is recognized and supported by developers open community world wide.

Spring Security Features:

- LDAP (Lightweight Directory Access Protocol)
- Single sign-on
- JAAS (Java Authentication and Authorization Service) LoginModule
- Basic Access Authentication
- Digest Access Authentication
- Remember-me
- Web Form Authentication
- Authorization
- Software Localization
- HTTP Authorization

LDAP (Lightweight Directory Access Protocol)

It is an open application protocol for maintaining and accessing distributed directory information services over an Internet Protocol.

Single sign-on

This feature allows a user to access multiple applications with the help of single account(user name and password).

JAAS (Java Authentication and Authorization Service) LoginModule

It is a Pluggable Authentication Module implemented in Java. Spring Security supports it for its authentication process.

Basic Access Authentication

Spring Security supports Basic Access Authentication that is used to provide user name and password while making request over the network.

Digest Access Authentication

This feature allows us to make authentication process more secure than Basic Access Authentication. It asks to the browser to confirm the identity of the user before sending sensitive data over the network.

Remember-me

Spring Security supports this feature with the help of HTTP Cookies. It remember to the user and avoid login again from the same machine until the user logout.

Web Form Authentication

In this process, web form collect and authenticate user credentials from the web browser. Spring Security supports it

while we want to implement web form authentication.

Authorization

Spring Security provides the this feature to authorize the user before accessing resources. It allows developers to define access policies against the resources.

Software Localization

This feature allows us to make application user interface in any language.

HTTP Authorization

Spring provides this feature for HTTP authorization of web request URLs using Apache Ant paths or regular expressions.

Features added in Spring Security 5.0:

OAuth 2.0 Login

This feature provides the facility to the user to login into the application by using their existing account at GitHub or Google. This feature is implemented by using the Authorization Code Grant that is specified in the OAuth 2.0 Authorization Framework.

Reactive Support

From version Spring Security 5.0, it provides reactive programming and reactive web runtime support and even, we can integrate with Spring WebFlux.

Modernized Password Encoding

Spring Security 5.0 introduced new Password encoder **DelegatingPasswordEncoder** which is more modernize and solve all the problems of previous encoder **NoOpPasswordEncoder**.

Introduction to Spring Security

Overview of Spring Security

Spring Security is a powerful and customizable authentication and access control framework for Java applications. It provides comprehensive security services for Java EE-based enterprise software applications.

Key Features

- **Authentication:** Verifying the identity of a user or system.
- **Authorization:** Determining whether an authenticated user has access to a specific resource.
- **Protection against common vulnerabilities:** Such as session fixation, clickjacking, cross-site request forgery (CSRF), and more.
- **Integration:** Easily integrates with Spring-based applications.

Core Components

- **SecurityContext:** Holds the security information of the currently authenticated user.
- **Authentication:** Represents the token for an authentication request or for an authenticated principal.
- **GrantedAuthority:** Represents an authority granted to an Authentication object.
- **UserDetailsService:** Used to retrieve user-related data.

Configuring Spring Security in a Spring Boot Application

Dependencies

Add the Spring Security starter dependency to your pom.xml:

xml

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

Basic Configuration

By default, Spring Security secures all HTTP endpoints with basic authentication. You can configure it using a `SecurityConfig` class.

`@EnableWebSecurity`

```
public class SecurityConfig extends WebSecurityConfigurerAdapter {
```

```
    @Override
```

```
    protected void configure(HttpSecurity http) throws Exception {
```

```
        http
```

```
            .authorizeRequests()
```

```
                .anyRequest().authenticated()
```

```
                .and()
```

```
            .formLogin()
```

```
                .and()
```

```
            .httpBasic();
```

```
    }
```

```
}
```

Customizing Authentication

You can customize the authentication mechanism using the `configure(AuthenticationManagerBuilder)` method.

`@Override`

```
protected void configure(AuthenticationManagerBuilder auth) throws  
Exception {
```

```
    auth.inMemoryAuthentication()
```

```
        .withUser("user").password("{noop}password").roles("USER")
```

```
        .and()
```

```
        .withUser("admin").password("{noop}admin").roles("ADMIN");
```

```
}
```

In-Memory and JDBC-Based Authentication

In-Memory Authentication

In-memory authentication is useful for testing and small applications.

`@Override`

```
protected void configure(AuthenticationManagerBuilder auth) throws  
Exception {
```

```

    auth.inMemoryAuthentication()
        .withUser("user").password("{noop}password").roles("USER")
        .and()
        .withUser("admin").password("{noop}admin").roles("ADMIN");
}

```

JDBC-Based Authentication

For production applications, you would typically use JDBC-based authentication, where user details are stored in a database.

Dependencies:

xml

```

<dependency>

    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>

```

Database Schema:

sql

```

CREATE TABLE users (

    username VARCHAR(50) NOT NULL PRIMARY KEY,
    password VARCHAR(100) NOT NULL,
    enabled BOOLEAN NOT NULL
);

CREATE TABLE authorities (
    username VARCHAR(50) NOT NULL,
    authority VARCHAR(50) NOT NULL,
    FOREIGN KEY (username) REFERENCES users(username)
);

```

Configuration:

@Override

```
protected void configure(AuthenticationManagerBuilder auth) throws  
Exception {  
    auth.jdbcAuthentication()  
        .dataSource(dataSource)  
        .usersByUsernameQuery("select username, password, enabled from  
users where username=?")  
        .authoritiesByUsernameQuery("select username, authority from  
authorities where username=?");  
}
```

Application Properties:

```
spring.datasource.url=jdbc:h2:mem:testdb  
  
spring.datasource.driverClassName=org.h2.Driver  
spring.datasource.username=sa  
spring.datasource.password=password  
spring.h2.console.enabled=true
```

Day 2: Securing RESTful Services

Configuring Security for REST APIs

Securing REST APIs involves configuring Spring Security to handle HTTP requests in a secure manner.

Basic Configuration for REST APIs

@Override

```
protected void configure(HttpSecurity http) throws Exception {  
    http  
        .csrf().disable() // Disable CSRF for APIs  
        .authorizeRequests()
```

```

        .antMatchers("/api/public/**").permitAll()
        .anyRequest().authenticated()
        .and()
        .httpBasic();
    }

```

Role-Based Access Control

Role-based access control restricts access to certain parts of the application based on user roles.

@Override

```

protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
            .antMatchers("/api/admin/**").hasRole("ADMIN")
            .antMatchers("/api/user/**").hasRole("USER")
            .anyRequest().authenticated()
            .and()
            .httpBasic();
}

```

Using JWT (JSON Web Tokens) for Authentication

JWT is a compact, URL-safe means of representing claims to be transferred between two parties.

Dependencies

```

<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt</artifactId>
  <version>0.9.1</version>
</dependency>

```

JWT Utility Class

```

public class JwtUtil {

    private String secret = "mySecretKey";

    public String generateToken(String username) {
        return Jwts.builder()
            .setSubject(username)
            .setIssuedAt(new Date(System.currentTimeMillis()))

```

```

        .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60 *
10))
        .signWith(SignatureAlgorithm.HS256, secret)
        .compact();
    }

    public String extractUsername(String token) {
        return Jwts.parser()
            .setSigningKey(secret)
            .parseClaimsJws(token)
            .getBody()
            .getSubject();
    }

    public boolean validateToken(String token, UserDetails userDetails) {
        final String username = extractUsername(token);
        return (username.equals(userDetails.getUsername()) && !
isTokenExpired(token));
    }

    private boolean isTokenExpired(String token) {
        return Jwts.parser()
            .setSigningKey(secret)
            .parseClaimsJws(token)
            .getBody()
            .getExpiration()
            .before(new Date());
    }
}

```

JWT Filter

```

public class JwtRequestFilter extends OncePerRequestFilter {

    @Autowired
    private JwtUtil jwtUtil;

    @Autowired
    private UserDetailsService userDetailsService;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
    HttpServletResponse response, FilterChain chain)
        throws ServletException, IOException {

```

```

        final String authorizationHeader = request.getHeader("Authorization");

        String username = null;
        String jwt = null;

        if (authorizationHeader != null && authorizationHeader.startsWith("Bearer
    ")) {
            jwt = authorizationHeader.substring(7);
            username = jwtUtil.extractUsername(jwt);
        }

        if (username != null &&
SecurityContextHolder.getContext().getAuthentication() == null) {
            UserDetails userDetails =
this.userDetailsService.loadUserByUsername(username);
            if (jwtUtil.validateToken(jwt, userDetails)) {
                UsernamePasswordAuthenticationToken
usernamePasswordAuthenticationToken = new
UsernamePasswordAuthenticationToken(
                    userDetails, null, userDetails.getAuthorities());
                usernamePasswordAuthenticationToken.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));

SecurityContextHolder.getContext().setAuthentication(usernamePasswordAut
henticationToken);
            }
        }
        chain.doFilter(request, response);
    }
}

```

Security Configuration

```

@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private JwtRequestFilter jwtRequestFilter;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .csrf().disable()

```



```

        .authorizeRequests()
        .antMatchers("/authenticate").permitAll()
        .anyRequest().authenticated()
        .and()
        .sessionManagement()
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS);
    http.addFilterBefore(jwtRequestFilter,
UsernamePasswordAuthenticationFilter.class);
}

@Override
protected void configure(AuthenticationManagerBuilder auth) throws
Exception {
    auth.userDetailsService(userDetailsService());
}

@Bean
@Override
public AuthenticationManager authenticationManagerBean() throws
Exception {
    return super.authenticationManagerBean();
}
}

```

Day 3: OAuth2 and OpenID Connect

Introduction to OAuth2

OAuth2 is an authorization framework that enables applications to obtain limited access to user accounts on an HTTP service. It works by delegating user authentication to the service that hosts the user account and authorizing third-party applications to access the user account.

Key Concepts

- **Resource Owner:** The user who authorizes an application to access their account.
- **Client:** The application requesting access to the user's account.
- **Resource Server:** The server hosting the protected resources.
- **Authorization Server:** The server that authenticates the user and issues access tokens.

Implementing OAuth2 in Spring Boot

Dependencies

Add the following dependencies to your pom.xml:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-oauth2-jose</artifactId>
</dependency>
```

Application Properties

Configure your application properties for OAuth2:

```
spring.security.oauth2.client.registration.google.client-id=your-client-id
spring.security.oauth2.client.registration.google.client-secret=your-client-secret
spring.security.oauth2.client.registration.google.scope=profile, email
spring.security.oauth2.client.registration.google.redirect-uri={baseUrl}/login/
oauth2/code/{registrationId}
```

Security Configuration

@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {

 @Override

 protected void configure(HttpSecurity http) throws Exception {

 http

 .authorizeRequests()

 .antMatchers("/", "/login**").permitAll()

 .anyRequest().authenticated()

 .and()

 .oauth2Login();

 }

}

Using Spring Security OAuth2 for Social Logins

Spring Security OAuth2 makes it easy to integrate with social login providers like Google, Facebook, etc.

Social Login Configuration

To configure social login, you need to register your application with the social login provider and obtain client credentials.

- **Google:**
 - Go to the Google Developers Console.
 - Create a new project and enable the "Google+ API".
 - Create OAuth 2.0 credentials and set the authorized redirect URI to `http://localhost:8080/login/oauth2/code/google`.
- **Facebook:**
 - Go to the [Facebook Developers](#).
 - Create a new app and configure the OAuth redirect URI.

Example Configuration for Google Login

Application Properties:

```
spring.security.oauth2.client.registration.google.client-id=your-google-client-id
spring.security.oauth2.client.registration.google.client-secret=your-google-client-secret
spring.security.oauth2.client.registration.google.scope=profile, email
spring.security.oauth2.client.registration.google.redirect-uri={baseUrl}/login/oauth2/code/{registrationId}
spring.security.oauth2.client.provider.google.authorization-uri=https://accounts.google.com/o/oauth2/auth
spring.security.oauth2.client.provider.google.token-uri=https://oauth2.googleapis.com/token
spring.security.oauth2.client.provider.google.user-info-uri=https://openidconnect.googleapis.com/v1/userinfo
```

Security Configuration:

@EnableWebSecurity

```
public class SecurityConfig extends WebSecurityConfigurerAdapter {
```

```
    @Override
```

```
    protected void configure(HttpSecurity http) throws Exception {
```

```
        http
```

```
            .authorizeRequests()
```

```
                .antMatchers("/", "/login**").permitAll()
```

```
                .anyRequest().authenticated()
```

```
                .and()
```

```
        .oauth2Login();  
    }  
}
```

Summary

- **Introduction to Spring Security:** Overview, configuration, and in-memory/JDBC-based authentication.
- **Securing RESTful Services:** Configuring security for REST APIs, role-based access control, and JWT-based authentication.
- **OAuth2 and OpenID Connect:** Introduction to OAuth2, implementing OAuth2 in Spring Boot, and using Spring Security OAuth2 for social logins.

How To Build And Deploy Spring Boot Application In Tomcat For DevOps ?

Spring Boot is a famous **Java framework for building stand-alone applications**. Now if we are going to build web applications using Spring Boot then it may not be useful to build stand-alone applications as we need to deploy the application on the web. Spring Boot by default packages the application in a **jar (Java ARchive)** file but to deploy the application using an external server, we need a **WAR (Web ARchive)** file. **Apache Tomcat** is a runtime environment for Java-based web applications and can be used to deploy Spring Boot Projects. In this article, we will be discussing how we can deploy a **Spring Boot Project** using a **Tomcat server**.

What is Spring Boot?

Spring Boot, developed by Pivotal Software is an **open-sourced Java framework** used for application development using [Java](#) and [Kotlin](#). This framework is designed to simplify the process of creating industry-grade applications with ease of development by reducing **boilerplate code** and **easy configuration** through [Spring Starters](#). Spring Boot also comes with an **embedded HTTP server**, to allow applications to be deployed as stand-alone applications.

What is Tomcat?

[Apache Tomcat](#) is an **open-sourced** implementation of **Jakarta Servlet**, **Java Server Pages (JSP)**, Java-based, and **Jakarta Expression Language (EL)** technologies with **Web socket technologies**. Tomcat works as a **servlet container** and provides a **runtime environment** for **Java-based web applications**. It manages the execution of **servlets** and **JSP pages**. Tomcat can be defined as a bridge between the **application** and the **web server**, handling **HTTP requests** and **responses**.

Steps to deploy Spring Boot Project using Tomcat

Step 1: Create Spring Boot Project

To create a Spring Boot project, go to start.spring.io and choose the following configurations,

Project: Maven

Language: Java

Spring Boot: 3.2.3

Packaging: Jar

Java: 17

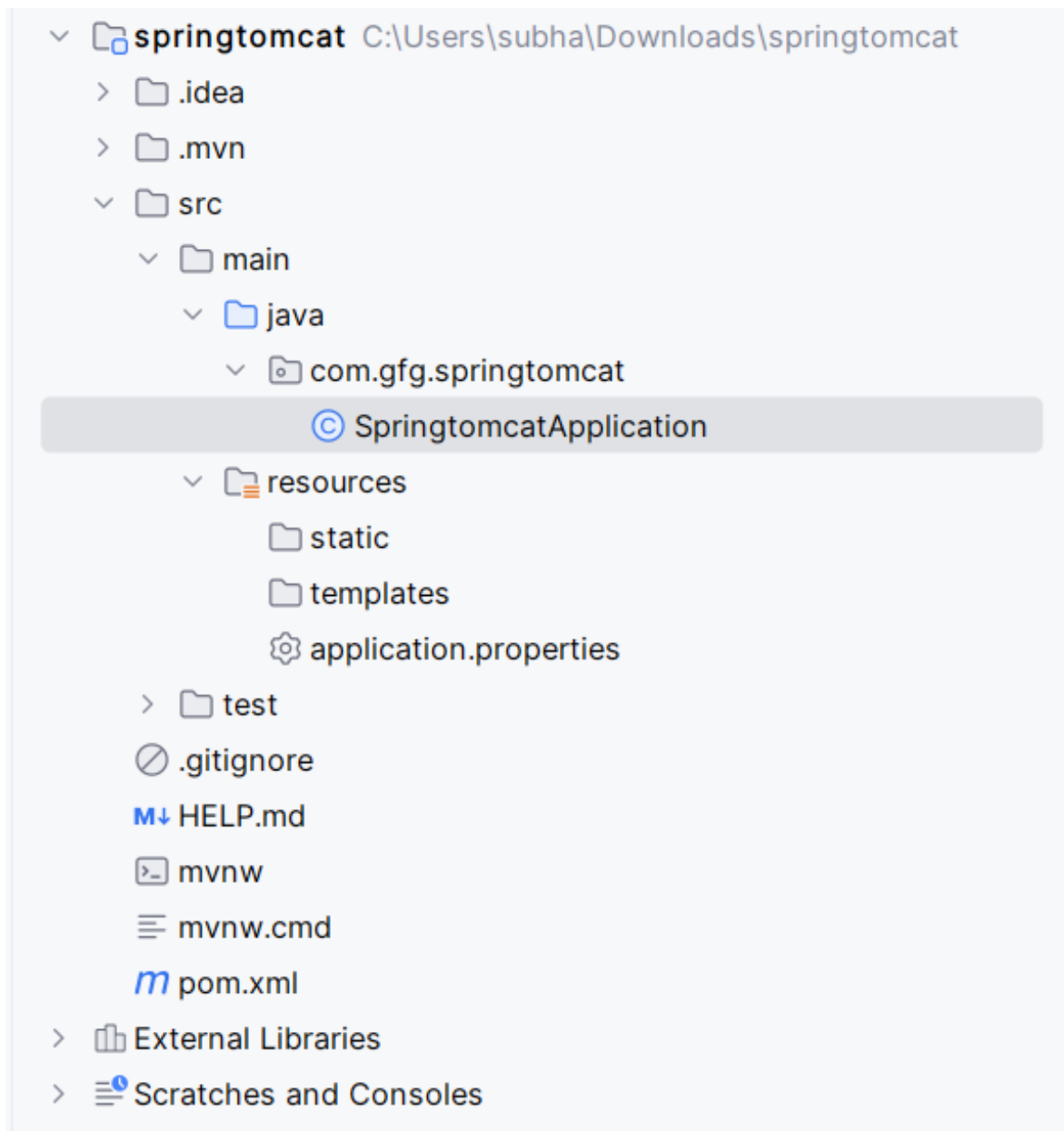
Dependency: Spring Web

Project <input type="radio"/> Gradle - Groovy <input type="radio"/> Gradle - Kotlin <input checked="" type="radio"/> Java <input type="radio"/> Kotlin <input type="radio"/> Groovy <input checked="" type="radio"/> Maven	Language <input checked="" type="radio"/> Java <input type="radio"/> Kotlin <input type="radio"/> Groovy
Spring Boot <input type="radio"/> 3.3.0 (SNAPSHOT) <input type="radio"/> 3.3.0 (M2) <input type="radio"/> 3.2.4 (SNAPSHOT) <input checked="" type="radio"/> 3.2.3 <input type="radio"/> 3.1.10 (SNAPSHOT) <input type="radio"/> 3.1.9	
Project Metadata	
Group	<input type="text" value="com.gfg"/>
Artifact	<input type="text" value="springtomcat"/>
Name	<input type="text" value="springtomcat"/>
Description	<input type="text" value="Demo project for Spring Boot with tomcat server"/>
Package name	<input type="text" value="com.gfg.springtomcat"/>
Packaging	<input checked="" type="radio"/> Jar <input type="radio"/> War
Java	<input type="radio"/> 21 <input checked="" type="radio"/> 17

Dependencies ADD DEPENDENCIES... CTRL + B

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Give proper name to your project and write description if you want. Once done, click on **Generate** and the project will be downloaded in your computer. **Unzip** the project and open it with any of the **IDE** and wait for few seconds for auto configuration, and then go ahead and explore the project.



For dependencies, check the **pom.xml** file below.**pom.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
```

```

    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.2.3</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.gfg</groupId>
  <artifactId>springtomcat</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>springtomcat</name>
  <description>Demo project for Spring Boot with tomcat
server</description>
  <properties>
    <java.version>17</java.version>
  </properties>
  <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>

```

```

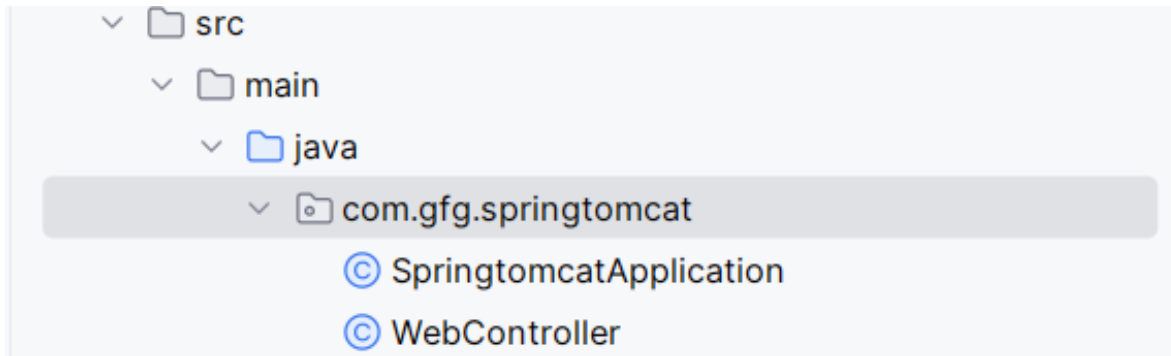
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>

```

```
</project>
```

Step 2: Create a Controller

Now to display anything in webpage we need to create a **controller** which will return some message. So quickly create a **controller** in the **com.gfg.springtomcat** package.



WebController.java

```
package com.gfg.springtomcat;
```

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

```
@RestController  
@RequestMapping  
public class WebController {  
    @GetMapping("/")  
    public String getMessage()  
    {  
        return "Spring Boot Application running on Tomcat  
server!!";  
    }  
}
```

Step 3: Run using embedded Tomcat Server

Spring Boot comes with an **embedded Tomcat server** to run stand alone java application. Let's run our

application using embedded Tomcat and see the result and later on we will run this application on **External Tomcat server** and match the output.

Go to **SpringTomcatApplication.java** and run the application.

```

  ____  _
 / ___|| | | |
| |___| |_| |
|___|_||_|_|_|

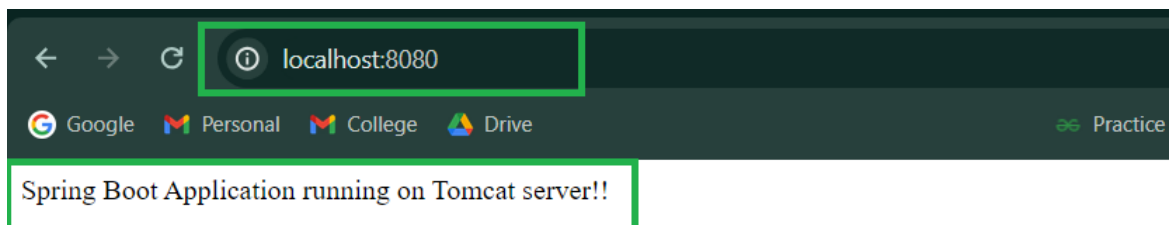
:: Spring Boot ::
      (v3.2.3)

2024-03-19T15:13:53.289+05:30 INFO 9532 --- [springtomcat] [
2024-03-19T15:13:53.294+05:30 INFO 9532 --- [springtomcat] [
2024-03-19T15:13:54.552+05:30 INFO 9532 --- [springtomcat] [
2024-03-19T15:13:54.573+05:30 INFO 9532 --- [springtomcat] [
2024-03-19T15:13:54.573+05:30 INFO 9532 --- [springtomcat] [
2024-03-19T15:13:54.706+05:30 INFO 9532 --- [springtomcat] [
2024-03-19T15:13:54.708+05:30 INFO 9532 --- [springtomcat] [
2024-03-19T15:13:55.358+05:30 INFO 9532 --- [springtomcat] [
2024-03-19T15:13:55.373+05:30 INFO 9532 --- [springtomcat] [

main] c.g.s.SpringtomcatApplication : Starting SpringtomcatApplication using Java 17.0.9 with
main] c.g.s.SpringtomcatApplication : No active profile set, falling back to 1 default profil
main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 8080 (http)
main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.19]
main] o.s.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in
main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path '/'
main] c.g.s.SpringtomcatApplication : Started SpringtomcatApplication in 2.549 seconds (process

```

As you can see from the console output, The embedded Tomcat server has been initialized and application is running on Port no **8080**.Let's go to **localhost:8080** and check the output.



Step 4: Change packaging to WAR

As we have discussed earlier to deploy Spring Application on Tomcat we need a WAR packaging. You can choose the packaging as WAR during project creation. If you have selected WAR then you can skip this step. This step is necessary for those who has a Spring Boot Application with Jar packaging and now want to deploy the application on external server or want

to create a [CI/CD pipeline](#) for the project.

- Define packaging as WAR in **pom.xml**

```
    </parent>
    <packaging>war</packaging>
    <groupId>com.gfg</groupId>
```

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

- As we are going to use the embedded Tomcat server, let's change the **scope** of the embedded Tomcat server. To do that, add the following dependency of tomcat and change the Scope as **provided**.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-tomcat</artifactId>
  <scope>provided</scope>
</dependency>
```

Here is the Complete **pom.xml** file after the changes,

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.2.3</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <packaging>war</packaging>
  <groupId>com.gfg</groupId>
```

```
<artifactId>springtomcat</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>springtomcat</name>
<description>Demo project for Spring Boot with tomcat
server</description>
<properties>
  <java.version>17</java.version>
</properties>
<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>
```

```
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-tomcat</artifactId>
      <scope>provided</scope>
    </dependency>
```

```
</dependencies>
```

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

```
</project>
```

Step 5: Change SpringTomcatApplication.java

Go the **SpringTomcatApplication.java** file which contains the main() method and make the following changes.

```
package com.gfg.springtomcat;
```

```
import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;
import
org.springframework.boot.builder.SpringApplicationBuilder;
import
org.springframework.boot.web.servlet.support.SpringBootServletInitializer;
```

```
@SpringBootApplication
public class SpringtomcatApplication extends
SpringBootServletInitializer {
```

```
    public static void main(String[] args) {
        SpringApplication.run(SpringtomcatApplication.class,
args);
    }
```

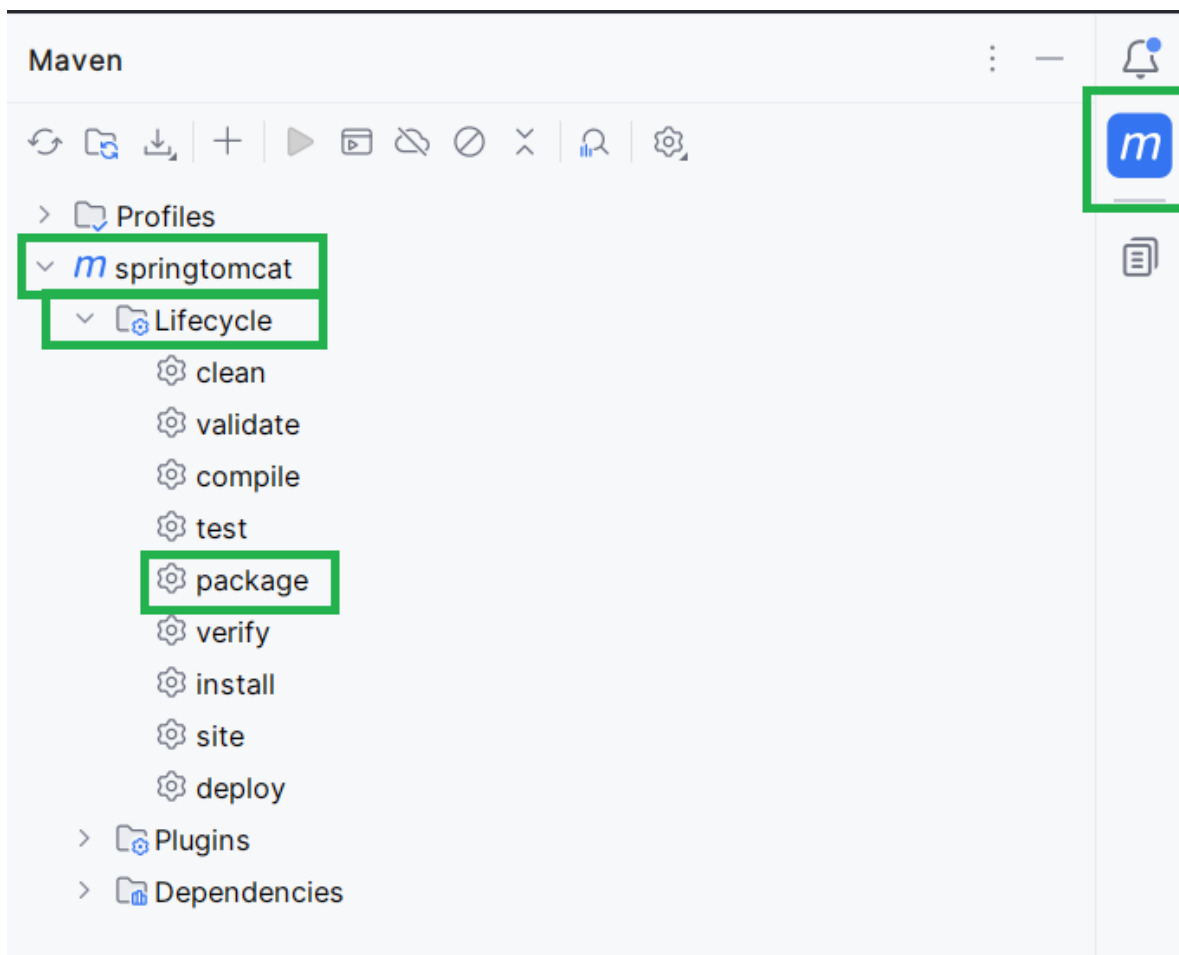
```
    @Override
    protected SpringApplicationBuilder
configure(SpringApplicationBuilder builder) {
        return builder.sources(SpringtomcatApplication.class);
    }
}
```

Here we have extended the **SpringBootServletInitializer** class which is necessary

to run a **WAR** application. This will bind the **application context** to the **server**. We have also override the **SpringApplicationBuilder configure** to define the **class** with **main()** method as the source class.

Step 6: Generate WAR build

Now to generate the WAR build, first delete the **target folder** in the Project. Now click on the **maven logo** on the right corner and enlarge the **Lifecycle section**.



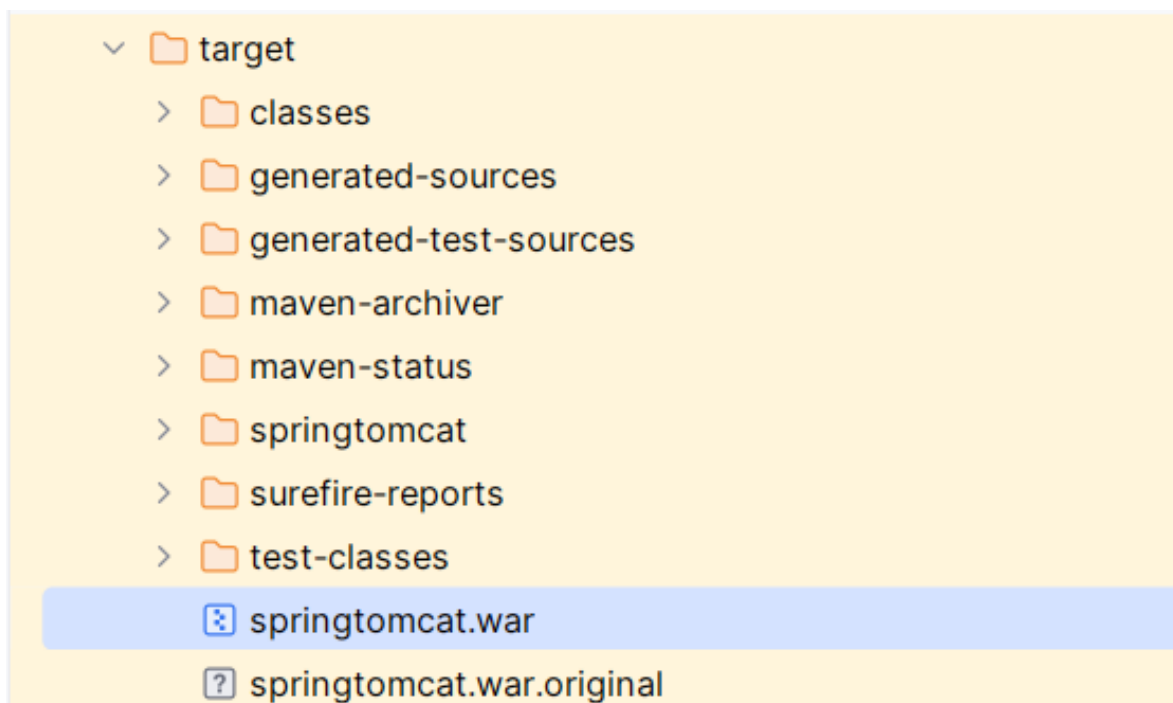
Now click on the **Package** option to create a new WAR packaging of the application.

```

[INFO] Packaging webapp
[INFO] Assembling webapp [springtomcat] in [C:\Users\subha\Downloads\springtomcat\target\springtomcat]
[INFO] Processing war project
[INFO] Building war: C:\Users\subha\Downloads\springtomcat\target\springtomcat.war
[INFO] --- spring-boot:3.2.3:repackage (repackage) @ springtomcat ---
[INFO] Replacing main artifact C:\Users\subha\Downloads\springtomcat\target\springtomcat.war with repackaged archive, adding nested dependencies in BOOT-INF/.
[INFO] The original artifact has been renamed to C:\Users\subha\Downloads\springtomcat\target\springtomcat.war.original
[INFO] BUILD SUCCESS
[INFO] Total time: 7.889 s
[INFO] Finished at: 2024-03-19T15:46:35+05:30
[INFO]

```

As you can see from the console output that the **WAR** build is successful. To check the war file, go to the newly created **target folder**, there you should see a new **.war** file with the name same as your project.



Till now we have successfully package the Spring Application for deployment.

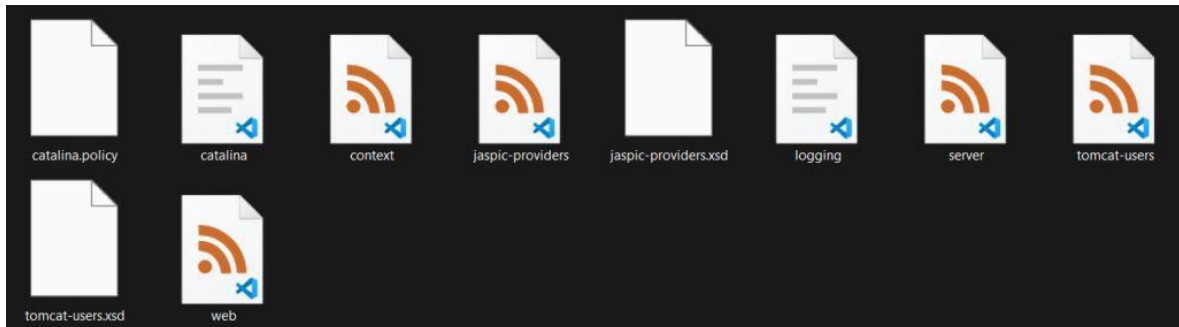
Step 7: Configure Tomcat server

a) Download Tomcat: Download the latest **Apache Tomcat** from the [rel="noopener" target="_blank">link](#) given. It should download a **zip** file in your computer. **Unzip** the file and go to,

apache-tomcat-10.1.19-windows-x64\apache-

tomcat-10.1.19\conf

In the **conf** folder you will find the following files,



b) Configure Port NO. : By default Tomcat is configured to run on port no **8080**, but it may not be available if other applications are using this port, to be safe, let's change the Port to **8085**. Open the **server.xml** file and use the following configuration for **connector**.

```
67      -->
68      <Connector port="8085" protocol="HTTP/1.1"
69          connectionTimeout="20000"
70          redirectPort="8443"
71          maxParameterCount="1000"
72      />
```

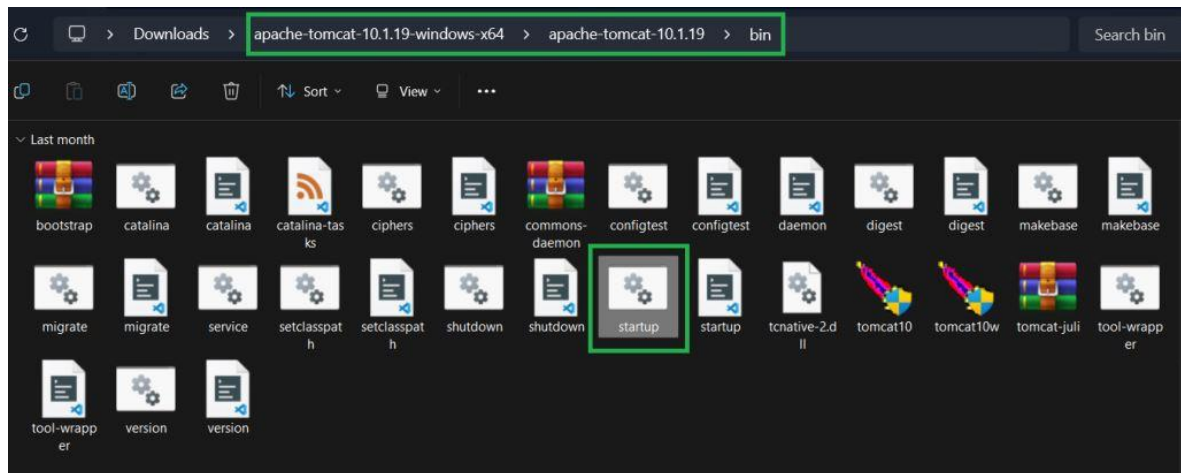
```
<Connector port="8085" protocol="HTTP/1.1"
    connectionTimeout="20000"
    redirectPort="8443"
    maxParameterCount="1000"
/>
```

c) Add user : To add a user, add the following line to the **tomcat-users.xml** file. Here I am defining **username** and **password** both as **admin**. The role is selected as **manager-gui** to get access to Tomcat server's [GUI](#) page.

```
<user username="admin" password="admin" roles="manager-gui"/>
```

d) Start Tomcat server : Go to the bin folder and look for the **startup.bat** file. Click on the file to open it in

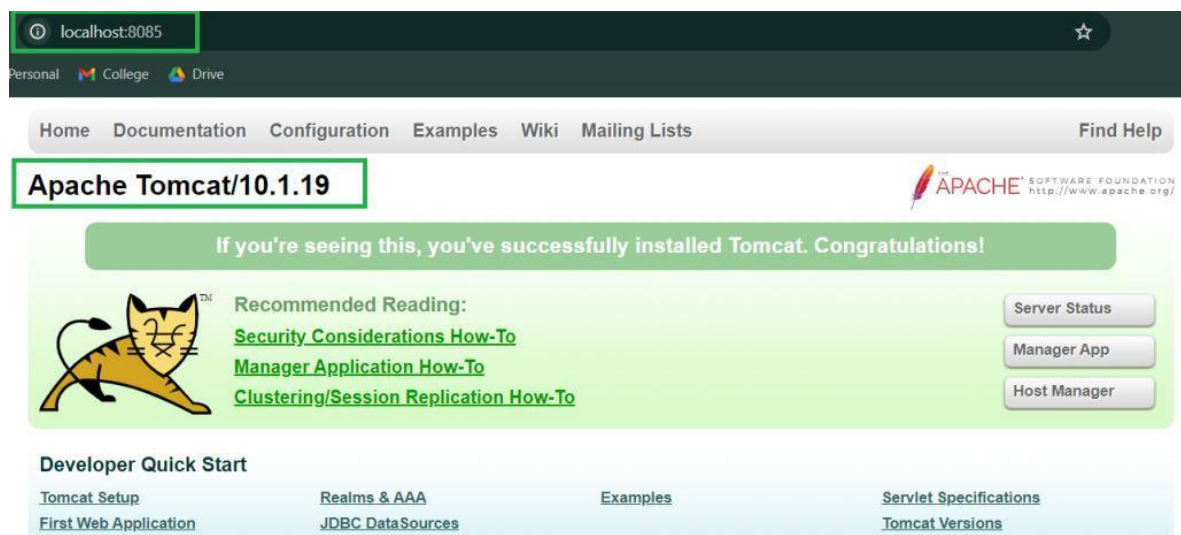
Windows terminal.



Once you click on it, the windows script will start running automatically and start the Tomcat server on the defined port.

```
19-Mar-2024 16:25:50.181 INFO [main] org.apache.catalina.startup.HostConfig.deployDirectory Deploying web application directory [C:\Users\subha\Downloads\apache-tomcat-10.1.19-windows-x64\apache-tomcat-10.1.19\webapps\ROOT]
19-Mar-2024 16:25:50.209 INFO [main] org.apache.catalina.startup.HostConfig.deployDirectory Deployment of web application directory [C:\Users\subha\Downloads\apache-tomcat-10.1.19-windows-x64\apache-tomcat-10.1.19\webapps\ROOT] has finished in [28] ms
19-Mar-2024 16:25:50.214 INFO [main] org.apache.coyote.AbstractProtocol.start Starting ProtocolHandler ["http-nio-8085"]
19-Mar-2024 16:25:50.263 INFO [main] org.apache.catalina.startup.Catalina.start Server startup in [953] milliseconds
```

Let's go to Port no **8085** on the web browser to check if Tomcat is running successfully or not.

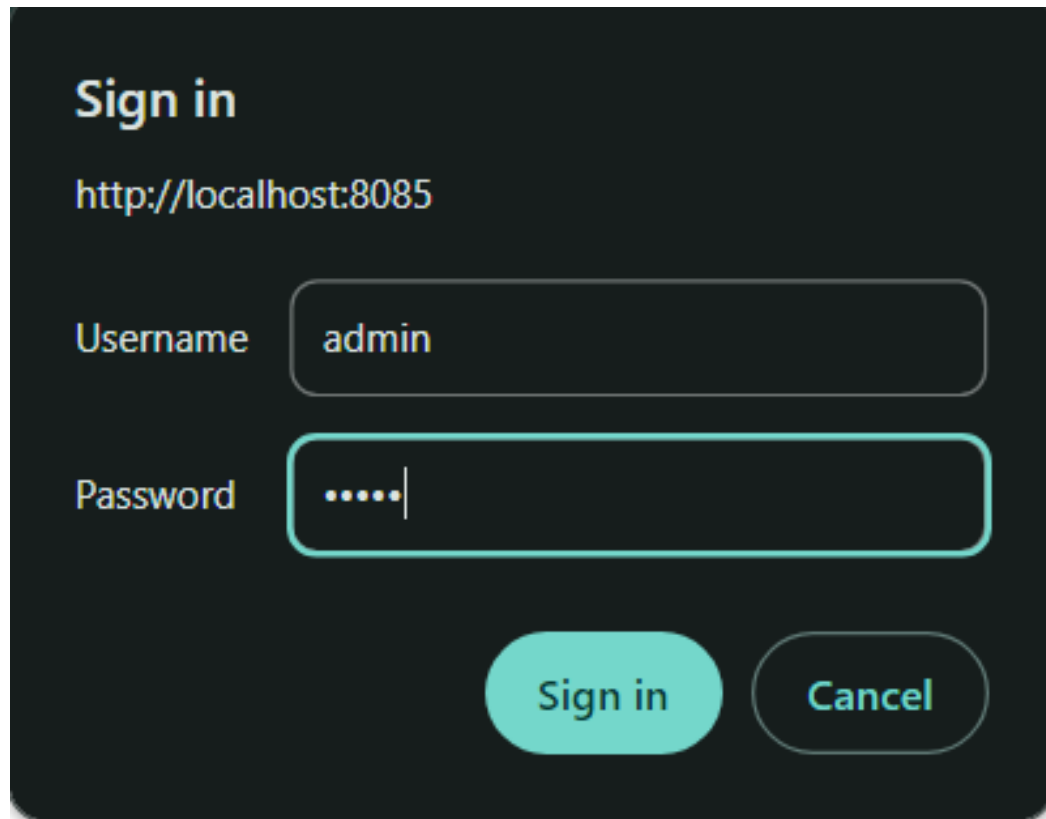


You can see that Tomcat is running successfully on **Port No. 8085**. You can also check out the article on [How to Install Apache Tomcat on Windows](#) for more

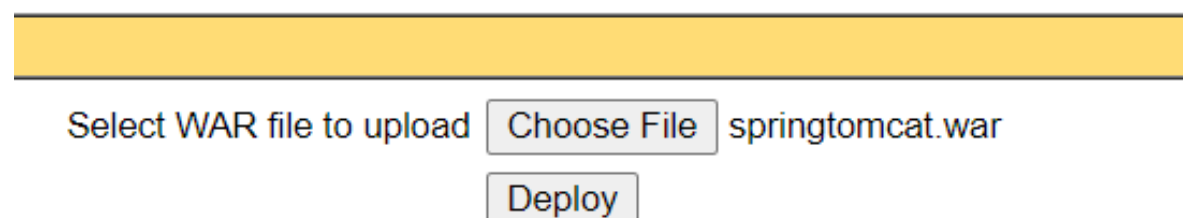
detailed explanation.

Step 8. Deploy Spring Application

Once you get the server running, Click on the **Manager app** button and log in with the **username** and **password** we have created.

A dark-themed sign-in dialog box for the Tomcat Manager App. It has a title 'Sign in' in orange. Below the title is the URL 'http://localhost:8085'. There are two input fields: 'Username' with the text 'admin' and 'Password' with masked characters '.....'. At the bottom are two buttons: 'Sign in' (orange) and 'Cancel' (grey).

This will open the **Manager App** for Tomcat server where you can see all the details of currently running applications in a [GUI](#). To deploy our Spring application, scroll down to the **War** file to deploy section and upload the war file we have created.

A screenshot of the 'War' file deployment section in the Tomcat Manager App. It features a yellow header bar. Below it, the text 'Select WAR file to upload' is followed by a 'Choose File' button and the filename 'springtomcat.war'. At the bottom is a 'Deploy' button.

Now click on **Deploy** and wait for few moments and you

should be able to see the spring project in the applications list.

/springtomcat	None specified		true	0	Start Stop Reload Undeploy
					Expire sessions with idle ≥ 30 minutes

This means we have successfully deployed the application using Tomcat.

Step 9. Check output

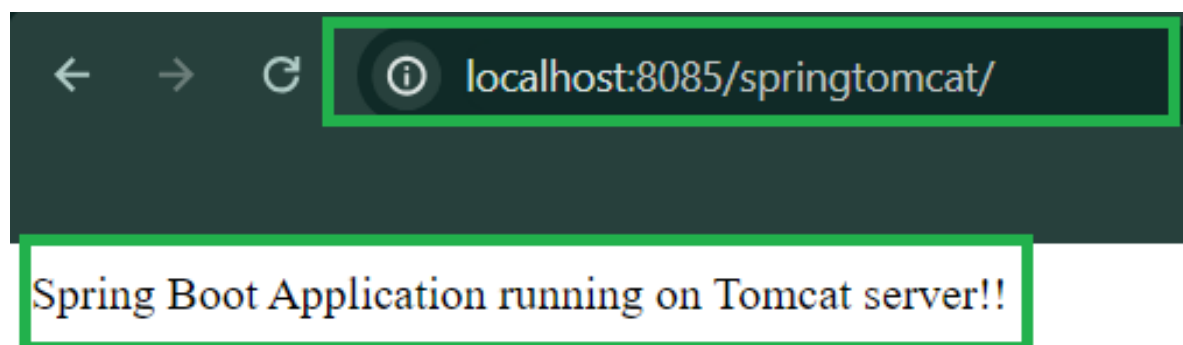
Once the application is deployed, you can access it on your web browser, but now the path will differ from the previous deployment using embedded Tomcat. The Path will look like the following,

localhost:8085/springtomcat

This is because the application is now accessible through Tomcat server which is running on Port **8085**. Now, let's try to access the controller we have created with the mapping as "/". Go to the web browser and use the following path,

localhost:8085/springtomcat/

Output:



As you can see we are the getting desired output which is same as the previous deployment on the specified

path.

Conclusion

Now you can use external **Tomcat** server to deploy your Spring projects. You have also learned how we can change the packaging of a Spring Application from **JAR** to **WAR**. The WAR packaging is necessary to deploy web applications and can be used if we want to streamline the development process with [CI/CD pipeline](#). Now you can move forward and build industry grade Spring Boot applications and deploy them using [DevOps](#) technologies

How To Dockerize A Spring Boot Application With Maven ?

Last Updated : 04 Apr, 2024

Improve

Docker is an open-source **containerization tool** used for building, running, and managing applications in an isolated environment. A container is isolated from another and bundles its software, libraries, and configuration files. In this article, to **dockerize a Spring Boot application** for deployment purposes, we will learn how to create a spring boot app, and how to create a **docker image** for the spring boot app and we will run it on the **docker container**.

Prerequisites: Before continuing any further, please ensure that node and docker are installed on your

machine. If required, visit the [Java Installation Guide](#) or the [Docker Installation Guide](#).

Dockerize a Standalone Spring Boot Application

Standalone Spring Boot Application: A Standalone Spring Boot Application is a Java application utilizing the Spring Boot framework, capable of running independently without external server software. It embeds an application server and can be executed as a standalone JAR file, simplifying deployment and reducing dependencies.

To dockerize a spring boot application, we need to first create a simple spring boot application. Then we need to add the maven plugin to our XML file, and after that, we can create an executable jar file.

Setting up a spring boot application

Follow the steps mentioned below to dockerize the spring boot application.

Step 1: Create a skeleton application using <https://start.spring.io>.

Step 2: Now create a maven project with the following configuration. After entering all the details, click on the 'GENERATE' button to download the project.

Project

☒ Maven Project

☐ Gradle Project

Language

☒ Java

☐ Kotlin

☐ Groovy

Spring Boot

☒ 3.0.0 (SNAPSHOT)

☐ 3.0.0 (M4)

☐ 2.7.4 (SNAPSHOT)

☐ 2.7.3

☐ 2.6.12 (SNAPSHOT)

☐ 2.6.11

Project Metadata

Group

com.docker

Artifact

spring

Name

spring

Description

Dockerizing Spring Boot application

Package name

com.docker.spring

Packaging

☒ Jar

☐ War

Java

☒ 18

☐ 17

☐ 11

☐ 8

Dependencies

ADD ... CTRL + B

Spring Web

WEB

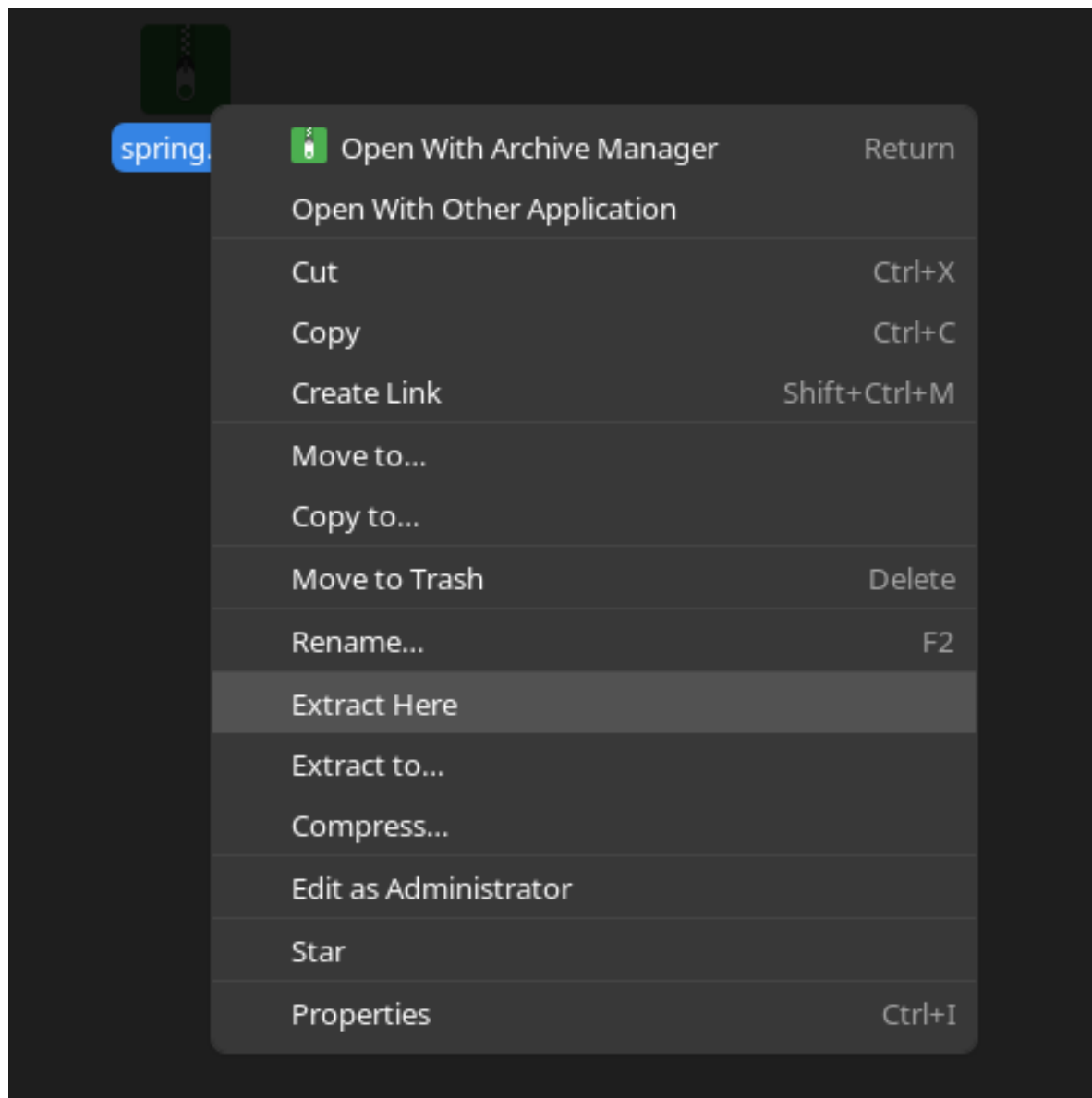
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

GENERATE CTRL + ⌘

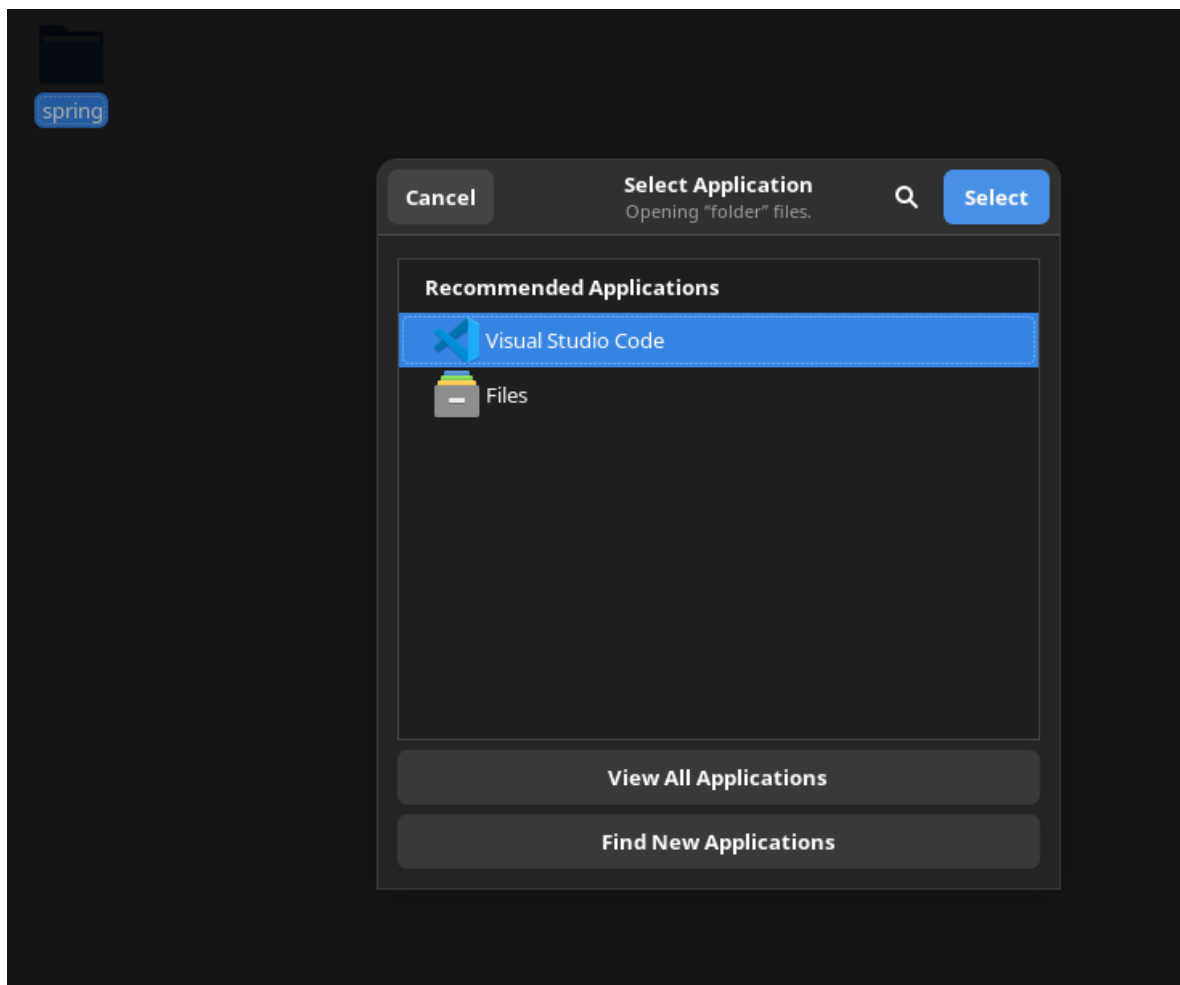
EXPLORE CTRL + SPACE

SHARE...

Step 3: Extract the zipped file.



Now, open it in an IDE of your choice.



Spring Boot buildpacks

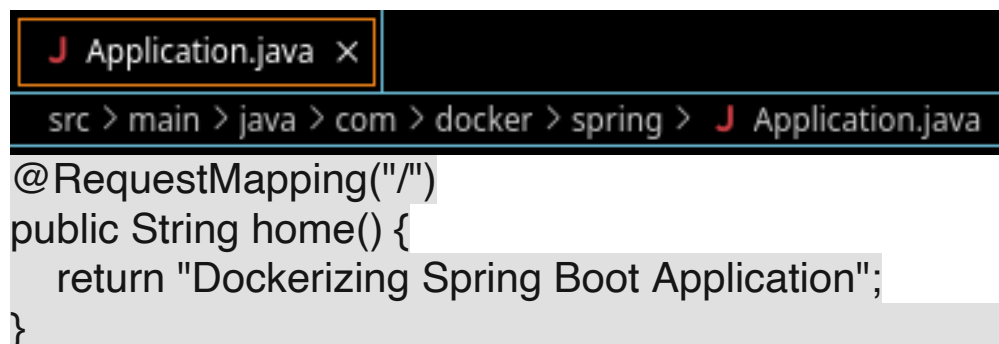
Dockerfiles and manual configuration are not required when pack the Spring Boot apps into the container images thanks to Spring Boot buildpacks. It allow developers to take advantage of automatically image generation specific to their Spring Boot projects, freeing them up to concentrate on writing. These buildpacks automatically identify the prerequisites for the application, including runtime parameters and dependencies, and put everything together to make a container image that is suitable for production. This increases developer productivity, streamlines the containerization process, and improves consistency

between deployments.

Step 4: Below one is the dependency to create docker image with the buildpacks. Now we need to add below Maven plugin in our pom.xml file.

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

Step 5: Open the base java file of the project and add a new controller to the base class of the application.



```
J Application.java ×
src > main > java > com > docker > spring > J Application.java
@RequestMapping("/")
public String home() {
    return "Dockerizing Spring Boot Application";
}
```

Step 6: Now, add the **@RestController** annotation and import the required packages. In the end, your Application.java file should look like this.

```
package com.docker.spring;

import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
```

```

@SpringBootApplication
@RestController
public class Application {

    @RequestMapping("/")
    public String home() {
        return "Dockerizing Spring Boot Application";
    }

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}

```

Step 7: Now start the application by running the following command,

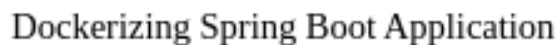
```
$ ./mvnw spring-boot:run
```

```

→ docker-springboot ./mvnw spring-boot:run
[INFO] Scanning for projects...
Downloading from spring-snapshots: https://repo.spring.io/snapshot/org/springframework/boot/spring-boot-starter-parent/3.0.0-SNAPSHOT/maven-metadata.xml
Downloaded from spring-snapshots: https://repo.spring.io/snapshot/org/springframework/boot/spring-boot-starter-parent/3.0.0-SNAPSHOT/maven-metadata.xml (810 B at 190 B/s)
Downloading from spring-snapshots: https://repo.spring.io/snapshot/org/springframework/boot/spring-boot-starter-parent/3.0.0-SNAPSHOT/spring-boot-starter-parent-3.0.0-20220910.145857-773.pom
Downloaded from spring-snapshots: https://repo.spring.io/snapshot/org/springframework/boot/spring-boot-starter-parent/3.0.0-SNAPSHOT/spring-boot-starter-parent-3.0.0-20220910.145857-773.pom (11 kB at 6.3 kB/s)
Downloading from spring-snapshots: https://repo.spring.io/snapshot/org/springframework/boot/spring-boot-dependencies/3.0.0-SNAPSHOT/maven-metadata.xml
Downloaded from spring-snapshots: https://repo.spring.io/snapshot/org/springframework/boot/spring-boot-dependencies/3.0.0-SNAPSHOT/maven-metadata.xml (808 B at 735 B/s)
Downloading from spring-snapshots: https://repo.spring.io/snapshot/org/springframework/boot/spring-boot-dependencies/3.0.0-SNAPSHOT/spring-boot-dependencies-3.0.0-20220910.145857-773.pom
Downloaded from spring-snapshots: https://repo.spring.io/snapshot/org/springframework/boot/spring-boot-dependencies/3.0.0-SNAPSHOT/spring-boot-dependencies-3.0.0-20220910.145857-773.pom (93 kB at 53 kB/s)
Downloading from spring-milestones: https://repo.spring.io/milestone/org/apache/groovy/groovy-bom/4.2.10/groovy-bom-4.2.10.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/groovy/groovy-bom/4.2.10/groovy-bom-4.2.10.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/groovy/groovy-bom/4.2.10/groovy-bom-4.2.10.pom (6.9 kB at 3.6 kB/s)
Downloading from spring-milestones: https://repo.spring.io/milestone/org/apache/groovy/groovy-parent/4.2.10/groovy-parent-4.2.10.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/groovy/groovy-parent/4.2.10/groovy-parent-4.2.10.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/groovy/groovy-parent/4.2.10/groovy-parent-4.2.10.pom (20 kB at 20 kB/s)
Downloading from spring-milestones: https://repo.spring.io/milestone/org/apache/groovy/groovy-bom/4.0.3/groovy-bom-4.0.3.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/groovy/groovy-bom/4.0.3/groovy-bom-4.0.3.pom

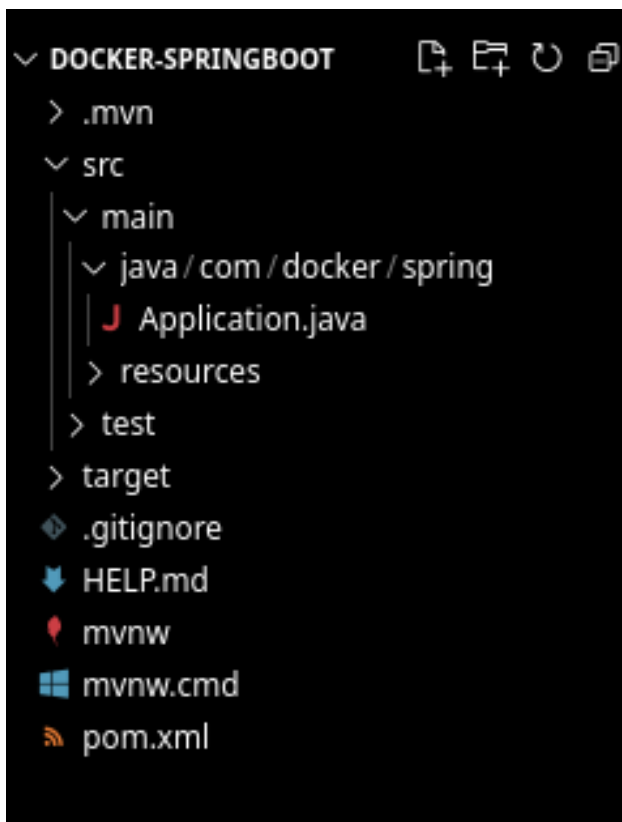
```

Step 8: Navigate to **http://localhost:8080** to test the application.



Project Structure

This is how the project structure should look at this point,



Dockerizing Our Application

Generate a .jar file

- Next, let's generate the .jar file by executing the command `./mvnw clean package` in your IntelliJ terminal. This command compiles the project, runs any necessary tests, and packages the application into a .jar file.
- Once the process is complete, navigate to the target folder within your project directory. You'll find the generated `spring-0.0.1-SNAPSHOT.jar` file there.

Now create a new jar file using maven builder.

```
$ ./mvnw clean package
```

The above command will build our docker image. Below

image refers to generate the jar file by executing the above command. Refer the second image the build was success and jar file created.

```
➔ docker-springboot ./mvnw clean package
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.docker:spring >-----
[INFO] Building spring 0.0.1-SNAPSHOT
[INFO] -----[ jar ]-----
Downloading from spring-milestones: https://repo.spring.io/milestone/org/apache/maven/plugins/maven-clean-plugin/3.2.0/maven-clean-plugin-3.2.0.pom
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-clean-plugin/3.2.0/maven-clean-plugin-3.2.0.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-clean-plugin/3.2.0/maven-clean-plugin-3.2.0.pom (5.3 kB at 1.4 kB/s)
Downloading from spring-milestones: https://repo.spring.io/milestone/org/apache/maven/plugins/maven-plugins/35/maven-plugins-35.pom
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-plugins/35/maven-plugins-35.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-plugins/35/maven-plugins-35.pom (9.9 kB at 7.4 kB/s)
Downloading from spring-milestones: https://repo.spring.io/milestone/org/apache/maven/maven-parent/35/maven-parent-35.pom
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/maven-parent/35/maven-parent-35.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/maven-parent/35/maven-parent-35.pom (45 kB at 44 kB/s)
Downloading from spring-milestones: https://repo.spring.io/milestone/org/apache/apache/25/apache-25.pom
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/apache/25/apache-25.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/apache/25/apache-25.pom (21 kB at 21 kB/s)
Downloading from spring-milestones: https://repo.spring.io/milestone/org/apache/maven/plugins/maven-clean-plugin/3.2.0/maven-clean-plugin-3.2.0.jar
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-clean-plugin/3.2.0/maven-clean-plugin-3.2.0.jar (36 kB at 27 kB/s)
Downloaded from spring-milestones: https://repo.spring.io/milestone/org/apache/maven/plugins/maven-surefire-plugin/2.22.2/maven-surefire-plugin-2.22.2.pom
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-surefire-plugin/2.22.2/maven-surefire-plugin-2.22.2.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-surefire-plugin/2.22.2/maven-surefire-plugin-2.22.2.pom (5.0 kB at 4.8 kB/s)
Downloading from spring-milestones: https://repo.spring.io/milestone/org/apache/maven/surefire/surefire/2.22.2/surefire-2.22.2.pom
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/surefire/surefire/2.22.2/surefire-2.22.2.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/surefire/surefire/2.22.2/surefire-2.22.2.pom (26 kB at 29 kB/s)
Downloading from spring-milestones: https://repo.spring.io/milestone/org/apache/maven/plugins/maven-surefire-plugin/2.22.2/maven-surefire-plugin-2.22.2.jar
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/maven-archiver/3.5.2/maven-archiver-3.5.2.jar
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/wagon/wagon-provider-api/2.10/wagon-provider-api-2.10.jar (54 kB at 69 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/commons/commons-compress/1.20/commons-compress-1.20.jar
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/shared/file-management/3.0.0/file-management-3.0.0.jar (35 kB at 44 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-archiver/4.2.7/plexus-archiver-4.2.7.jar
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-interpolation/1.16/plexus-interpolation-1.16.jar (61 kB at 73 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-io/3.2.0/plexus-io-3.2.0.jar
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/maven-compat/3.0/maven-compat-3.0.jar (285 kB at 191 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/maven-archiver/3.5.2/maven-archiver-3.5.2.jar (26 kB at 18 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/org/iq80/snappy/snappy/0.4/snappy-0.4.jar
Downloaded from central: https://repo.maven.apache.org/maven2/org/tukaani/xz/1.9/xz-1.9.jar
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-io/3.2.0/plexus-io-3.2.0.jar (76 kB at 43 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-utils/3.3.1/plexus-utils-3.3.1.jar
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-archiver/4.2.7/plexus-archiver-4.2.7.jar (195 kB at 110 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/commons/commons-compress/1.20/commons-compress-1.20.jar (632 kB at 329 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/org/iq80/snappy/snappy/0.4/snappy-0.4.jar (58 kB at 26 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/org/tukaani/xz/1.9/xz-1.9.jar (116 kB at 49 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-utils/3.3.1/plexus-utils-3.3.1.jar (262 kB at 99 kB/s)
[INFO] Building jar: /home/ahampriyanshu/docker-springboot/target/spring-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:3.0.0-SNAPSHOT:repackage (repackage) @ spring ---
[INFO] Replacing main artifact with repackaged archive
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 04:03 min
[INFO] Finished at: 2022-09-12T02:20:14+05:30
[INFO]
➔ docker-springboot []
```

A Basic Dockerfile

Run the following command at the root of our project to build a Dockerfile. This command will create an empty Dockerfile in the current directory. After creating the Dockerfile, you can proceed to define the necessary instructions for building your Docker image as mentioned in below steps.

```
$ touch Dockerfile
```

```
→ docker-springboot touch Dockerfile  
→ docker-springboot
```

Add configuration to dockerize the Spring Boot application

Now we have a working Spring Boot Application, To dockerize an application, now paste the following content into the Dockerfile:

```
FROM openjdk:18  
WORKDIR /app  
COPY ./target/spring-0.0.1-SNAPSHOT.jar /app  
EXPOSE 8080  
CMD ["java", "-jar", "spring-0.0.1-SNAPSHOT.jar"]
```

The above file contains the following information:

- FROM openjdk:18: This line specifies the base image for the Docker container. In this case, it uses the official OpenJDK image tagged with version 18. This image provides a runtime environment for Java applications.
- WORKDIR /app: This line sets the working directory inside the container to /app. All subsequent commands will be executed relative to this directory.
- COPY ./target/spring-0.0.1-SNAPSHOT.jar /app: This command copies the Spring Boot application JAR file from the host machine's ./target directory to the /app directory inside the container. It's assuming that you've built the Spring Boot application and the JAR

file is available in the specified location.

- EXPOSE 8080: This line informs Docker that the container will listen on port 8080 at runtime. However, it doesn't actually publish the port. It's merely a documentation of which ports the container is expected to use.
- CMD ["java", "-jar", "spring-0.0.1-SNAPSHOT.jar"]: This command specifies the default command to run when the container starts. It launches the Spring Boot application by executing the java -jar command with the Spring Boot JAR file as its argument.

Now create a docker image by using the docker build command:

```
$ docker build -t [name:tag] .
```

- **-t:** Name and tag for the image
- **.** : The context for the build process

Once the build process has been completed, you will receive the id and tag of your new image.

```
→ docker-springboot docker build -t ahampriyanshu/dockerspringboot .  
Sending build context to Docker daemon 18.03MB  
Step 1/5 : FROM eclipse-temurin:17-jdk-focal  
---> de7fb9000960  
Step 2/5 : WORKDIR /app  
---> Using cache  
---> 7240aaeb6876  
Step 3/5 : COPY ./target/spring-0.0.1-SNAPSHOT.jar /app  
---> 2f5d5fd8f5ce  
Step 4/5 : EXPOSE 8080  
---> Running in 83895f66012e  
Removing intermediate container 83895f66012e  
---> 94f631b87c25  
Step 5/5 : CMD ["java", "-jar", "spring-0.0.1-SNAPSHOT.jar"]  
---> Running in c181bfd41ef6  
Removing intermediate container c181bfd41ef6  
---> ce6c903446db  
Successfully built ce6c903446db  
Successfully tagged ahampriyanshu/dockerspringboot:latest  
→ docker-springboot
```


Run the Spring Boot Docker image in a container

Before running our image in a container, let's ensure that we avoid any potential errors when attempting to map the container port to the localhost port. To do this, we need to specify the correct port binding configuration in our Docker command or Docker Compose file. This ensures that the container's port is exposed and accessible from the host system.

Create a docker container by running following command:

```
$ docker run -d -p [host_port]:[container_port] --name [container_name] [image_id/image_tag]
→ docker-springboot docker run -d -p 8080:8080 --name dockerspringboot ahampriyanshu/dockerspringboot:latest
3b63f5a7ccea55178954c8316abc891b18852b6439cbe6a3acbd73dcd0a3963
→ docker-springboot
```

- **-d**: Run the container while printing the container ID.
- **-p**: Mapping port for our container
- **--name**: Assign a name to the container

Verify whether the container has been created successfully by running below command:

```
$ docker container ps
→ docker-springboot docker container ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS        PORTS
NAMES
3b63f5a7ccea   ahampriyanshu/dockerspringboot:latest  "java -jar spring-0..." 40 seconds ago Up 39 seconds 0.0.0.0:8080->8080/tcp
0/tcp dockerspringboot
→ docker-springboot
```

How to change the base image

Modifying the base image in a Dockerfile is a simple procedure that can have a big effect on the size, security, and functionality of your container. Take these actions to modify the base image:

1. Select a fresh basis image.

It is important to do thorough research to get a suitable base image that meets the requirements of your application before making any changes. The operating system, picture size, security features, and compatibility with your application stack are important factors to take into account. Official images from Docker Hub or other reliable sources should be given priority in order to ensure reliability and compliance with security best practices. You can choose a base image that optimizes both speed and security and serves as a strong basis for your containerized application by carefully weighing these considerations.

2. Make Dockerfile updates.

Open a text editor and choose your Dockerfile. Look for the instruction “FROM” at the start of the file, which indicates the current base image. Replace the existing image with a new one, as demonstrated in the sample below.

```
FROM image:version #Change the image and version as per  
your requirement
```

3. Adjust dependencies and configuration

With the variations between the old and new basic images, it may be required to update either of the dependencies, configurations, or instructions in your Dockerfile. Ensure that all required tools, libraries, and packages are installed and configured on the new base image.

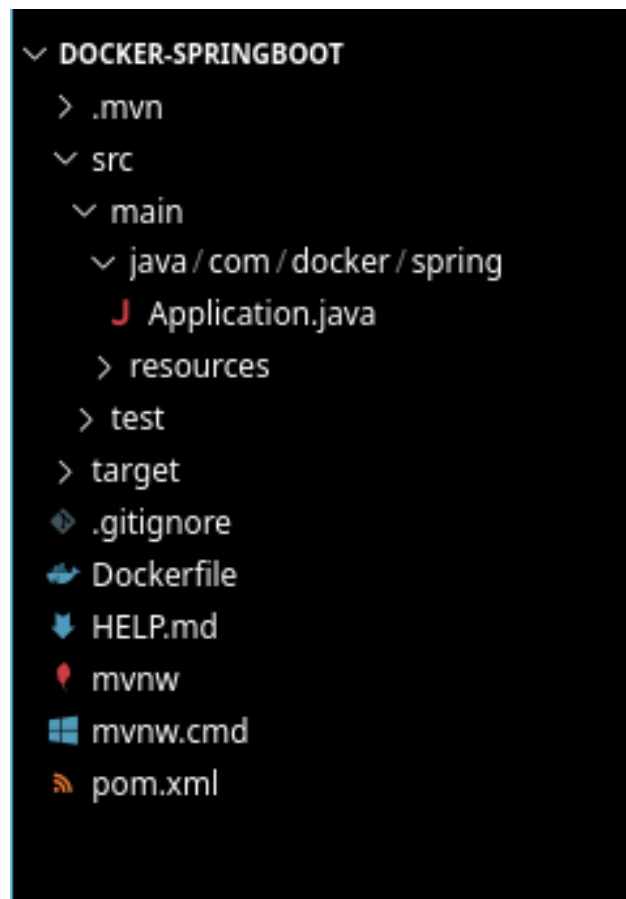
4. Verify your changes.

After making changes to the Dockerfile, re-create your Docker image using the `docker build` command:

```
docker build -t [name:tag] .
```

Check the project

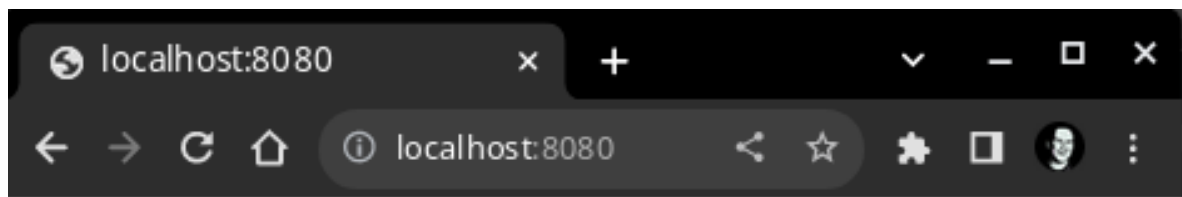
Make sure the project structure complies with the standard procedures and has all the parts required for the Spring Boot application. Additionally, confirm that the application's directory structure is logical and well-defined, and that all dependencies are configured correctly.



Check the application

Open your web browser and go to <http://localhost:8080/> to examine the program. This will show you the local

Spring Boot application running on your computer.



Dockerizing Spring Boot Application

Dockerize the application in a Compose tool

DockerFiles and Docker commands work well for building single containers. Container management slows down if we wish to operate on an application's shared network.

Docker offers a tool called Docker Compose to solve this problem. This utility has a unique build file format in YAML. Multiple services' configurations can be combined into a single file called docker-compose.yml.

The Docker Compose File

Here is the example Docker Compose file is a basic configuration for orchestrating multiple services. It defines two services, “service1” and “service2”, each with specified Docker images, container names, exposed ports, and environment variables. The version we can modify the as per our requirements.

```
version: '3.8'
```

```
services:
```

```
  service1:
```

```
    image: your_image_name
```

```
    container_name: your_container_name
```

```
    ports:
```

```
      - "host_port:container_port"
```

```
    environment:
```

```
      - ENV_VARIABLE=value
```

```
  service2:
```

```
    image: your_image_name
```

```
    container_name: your_container_name
```

```
    ports:
```

```
      - "host_port:container_port"
```

```
    environment:
```

```
      - ENV_VARIABLE=value
```

Below is the command to build docker-compose file:

```
$ docker-compose config
```

For building an image and creating docker container we can use the below command:

```
$ docker-compose up --build
```

To stop the container, we should remove them from Docker. For this we can use the below command:

```
$ docker-compose down
```

Scaling Services

Scaling services in Docker Compose refers to the ability to increase or decrease the number of instances of a service defined in a Docker Compose file. This can be achieved by adjusting the “scale” property of a service to specify the desired number of containers to run for that service.

```
$ docker-compose --file docker-compose.yml up -d --build --scale service=1 --scale product-server=1
```

Conclusion

So, in this way we can publish the images to Docker container using Maven plugin. It is not preferable to upload docker images to any registry using your local environment and it's always the best practice to use CI/CD pipeline or any of the tools.

Containerizing Spring Boot Application

Java applications, known for their robustness sometimes become difficult to deploy and manage. This is where containerization comes into the picture.

Packaging your Java app into a lightweight and self-contained independent unit can provide many benefits to the developers:

- Portability
- Scalability

- Faster Deployments

In this article we will discuss the complete process of [containerizing](#) the Java application using [Spring Boot App](#) and [Dockerfile](#), making it easier than ever to bring your Java apps to deploy. The steps involved, from setting up your Spring Boot app to building and running your very own Create Docker image as follows:

The steps will be as follows:-



















- 1 Setting up a spring-boot app
- 2 Create a docker image
- 3 Building project jar
- 4 Building a docker image by using a docker file
- 5 Running image

Let's examine the above steps in detail

Step By Step Implementation

For understanding first we will use a basic spring-boot greetings project with a single API to greet the user, with the help of a spring-boot-starter-web.

Overall we need to create these files in the directory.

- ✓  spring-boot-docker-demo
 - >  .mvn/wrapper
 - ✓  src
 - ✓  main
 - ✓  java/com/example/springboo...
 -  HelloController.java
 -  SpringBootDockerDemoAp...
 - ✓  resources
 -  application.properties
 - ✓  test/java/com/example/spring...
 -  SpringBootDockerDemoAppl...
 -  .gitignore
 -  Dockerfile
 -  mvnw
 -  mvnw.cmd
 -  pom.xml
 -  LICENSE
 -  README.md

1. Configuration

You can either do project configuration directly through the IDE, or you can select the below method:

- Go to [spring initializr](https://start.spring.io) website
- Select Project – Maven
- Language – Java
- Spring Boot Version – 2.2.1 (You can do this later by making changes in pom.xml)
- Packaging – Jar
- Java – 17
- Dependencies
 - Spring Web

The screenshot shows the Spring Initializr web application at <https://start.spring.io>. The interface is divided into several sections:

- Project:** Maven Project (selected) and Gradle Project.
- Language:** Java (selected), Kotlin, and Groovy.
- Spring Boot:** 2.2.2 (SNAPSHOT), 2.2.1 (selected), 2.1.11 (SNAPSHOT), and 2.1.10.
- Project Metadata:**
 - Group: com.example
 - Artifact: spring-boot-docker-demo
 - Options: > Options
- Dependencies:**
 - Search dependencies to add: Web, Security, JPA, Actuator, Devtools...
 - Selected dependencies: 1 selected
 - Spring Web**: Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container. (checked)

Overall you only need this dependency to for the project:

```
// Controller layer for Testing Project
package com.example.springbootdockerdemo;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
/**
 * Spring Boot controller class
 */
@RestController
public class HelloController {
    /**
     * HTTP GET requests to the root path ("/") and returns greeting
     message.
     */
}
```



```

    * @return message "Greetings from Spring Boot!!".
    */
    @RequestMapping("/")
    public String index() {
        return "Greetings from Spring Boot!!";
    }
}

```

To run this app use command:

```
mvn spring-boot:run
```

3. Creating A Dockerfile for Application

A dockerfile is a text document which contains commands read by docker and is executed in order to build a container image.

```

FROM java:8-jdk-alpine
COPY target/spring-boot-docker-demo-0.0.1-SNAPSHOT.jar /
usr/app/
WORKDIR /usr/app
RUN sh -c 'touch spring-boot-docker-demo-0.0.1-
SNAPSHOT.jar'
ENTRYPOINT ["java","-jar","spring-boot-docker-demo-0.0.1-
SNAPSHOT.jar"]

```

Explanation of docker file:

- **FROM:** The keyword FROM tells Docker to use a given base image as a build base. In this case Java8 is used as base image with jdk-alpine as tag. A tag can be thought as a version.
- **COPY:** copying **.jar file** to the build image inside **/usr/app**
- **WORKDIR:** The **WORKDIR** instruction sets the working directory for any **RUN, CMD, ENTRYPOINT, COPY** and **ADD** instructions that follow in the Dockerfile. Here the workdir is switched to /usr/app

- **RUN:** The **RUN** instruction runs any command mentioned.
- **ENTRYPOINT:** Tells Docker how to run application.
Making array to run spring-boot app as **java -jar .jar**

Building Project Jar

Now run mvn install to build a .jar file in target directory.

```
mvn install
```

Building Docker Image

Execute the below command to complete the build of Docker Image

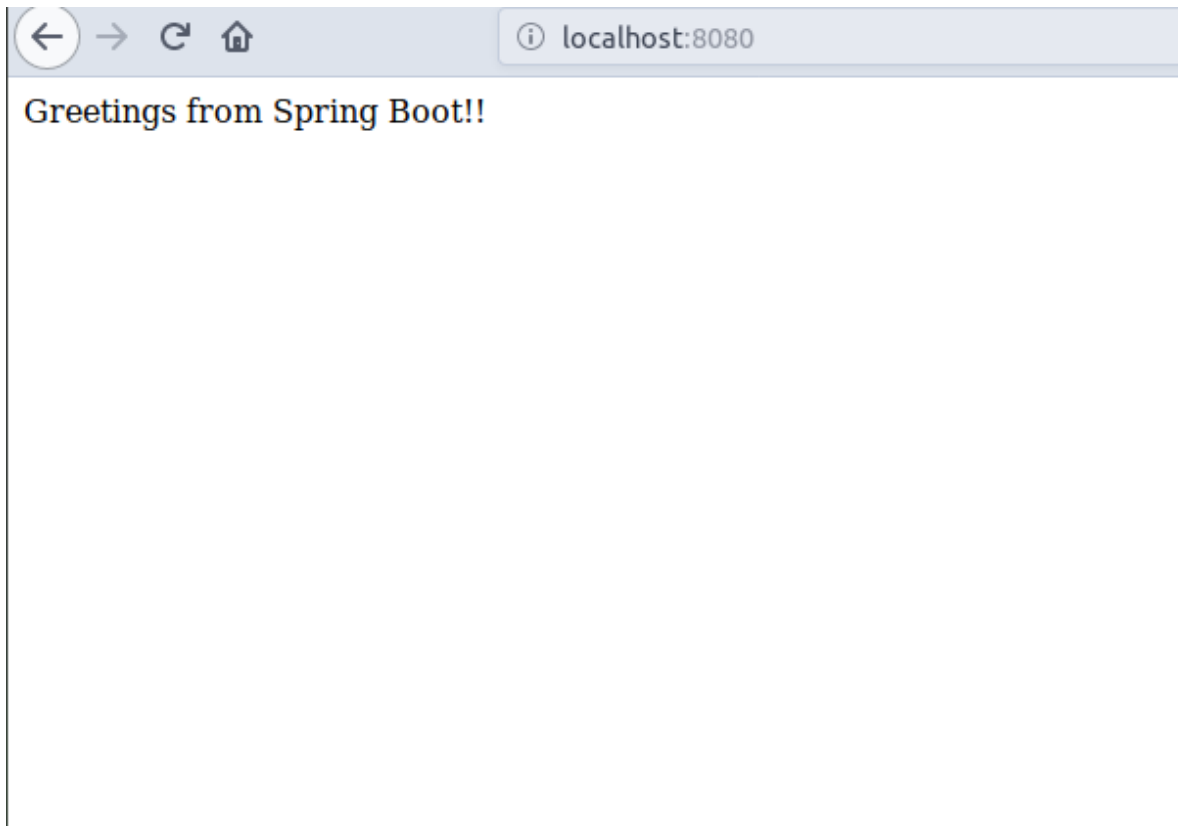
```
docker build -t spring-boot-docker-demo
```

```
-t spring-boot-docker-demo .
Sending build context to Docker daemon 17.73MB
Step 1/5 : FROM java:8-jdk-alpine
----> 3fd9dd82815c
Step 2/5 : COPY target/spring-boot-docker-demo-0.0.1-SNAPSHOT.jar /usr/app/
----> e28acf2b00a5
Step 3/5 : WORKDIR /usr/app
----> Running in a25d3d6c939c
Removing intermediate container a25d3d6c939c
----> 19a5287aala1
Step 4/5 : RUN sh -c 'touch spring-boot-docker-demo-0.0.1-SNAPSHOT.jar'
----> Running in bbfbf3eef0f8
Removing intermediate container bbfbf3eef0f8
----> df480211426f
Step 5/5 : ENTRYPOINT ["java","-jar","spring-boot-docker-demo-0.0.1-SNAPSHOT.jar"]
----> Running in a93972e0a744
Removing intermediate container a93972e0a744
----> 2b8b261d900c
Successfully built 2b8b261d900c
Successfully tagged spring-boot-docker-demo:latest
```

Run the image build

Execute the below command to complete the image build

```
docker run spring-boot-docker-demo
```



Conclusion

Now we have a portable, self-sufficient unit making Spring Boot application containerized using Docker now these application can be easily used across different environments-

- **Simplified deployments:** complex configurations and manual installations not necessary.
- **Scalability on demand:** No need acquire new servers or worry about resource constraints.
- **Improved collaboration:** Developers and operations teams can work more better by sharing the same docker image for development, testing, and production. Consistency and efficiency is improved.

Building and Packaging Spring Boot Applications

Creating Executable JARs and WARs

Executable JAR

Spring Boot applications are typically packaged as executable JARs, which include an embedded server.

Maven:

Add the Spring Boot Maven plugin to your pom.xml.

```
<build>

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

Build the project with:

```
mvn clean package
```

The resulting JAR file will be located in the target directory.

The resulting JAR file will be located in the build/libs directory.

Executable WAR

For deploying to traditional application servers, Spring Boot can also be packaged as a WAR.

Maven:

Modify your pom.xml to generate a WAR file.

```
<packaging>war</packaging>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
```

```

        <artifactId>spring-boot-starter-tomcat</artifactId>
        <scope>provided</scope>
    </dependency>
    <!-- Other dependencies -->
</dependencies>

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

```

Build the project with:

```
mvn clean package
```

Using Maven and Gradle for Builds

Maven

Maven is a build automation tool used primarily for Java projects. It uses a pom.xml file to manage project dependencies, build configurations, and plugins.

Managing Dependencies:

```

<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>
    <!-- Other dependencies -->
</dependencies>

```

Building the Project:

```
mvn clean package
```

Containerization with Docker

Introduction to Docker

Docker is a platform that enables developers to package applications into containers —standardized units that contain everything the software needs to run.

- **Docker Image:** A lightweight, standalone, executable package that includes everything needed to run a piece of software.
- **Docker Container:** A runtime instance of a Docker image.

Dockerizing Spring Boot Applications

Create a Dockerfile:

Dockerfile

```
FROM openjdk:11-jre-slim
```

```
VOLUME /tmp
```

```
COPY target/myapp.jar myapp.jar
```

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

Build the Docker Image:

```
sh
```

```
docker build -t myapp:latest .
```

Run the Docker Container:

```
sh
```

```
docker run -p 8080:8080 myapp:latest
```

Using Docker Compose for Multi-Container Applications

Docker Compose is a tool for defining and running multi-container Docker applications.

Create a `docker-compose.yml` File:

```
yaml
version: '3.8'

services:
  app:
    image: myapp:latest
    ports:
      - "8080:8080"
  database:
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: mydb
      MYSQL_USER: user
      MYSQL_PASSWORD: password
    ports:
      - "3306:3306"
```

Start the Multi-Container Application:

```
docker-compose up
```

Day 3: Deployment Strategies

Deploying to Cloud Platforms

AWS (Amazon Web Services)

1 Elastic Beanstalk:

- Create an Elastic Beanstalk application and environment.
- Upload your JAR/WAR file to deploy.

2 EC2 (Elastic Compute Cloud):

- Launch an EC2 instance.
- SSH into the instance and deploy your JAR/WAR file.

Google Cloud

1 App Engine:

- Create an App Engine application.
- Deploy using the gcloud command.

2

gcloud app deploy

Compute Engine:

- Create a VM instance.
- SSH into the instance and deploy your JAR/WAR file.

Azure

1 App Service:

- Create an App Service.
- Deploy using the Azure CLI.

```
az webapp deploy --resource-group myResourceGroup --name  
myAppName --src-path myapp.jar
```

VM (Virtual Machine):

- Create a VM.
- SSH into the VM and deploy your JAR/WAR file.

Using CI/CD Pipelines

Jenkins

Install Jenkins.

Create a Pipeline Job.

Configure the Pipeline:

GitHub Actions

Create a .github/workflows/ci.yml File:

yaml

name: CI

on: [push]

jobs:

build:

runs-on: ubuntu-latest

steps:

- name: Checkout code
uses: actions/checkout@v2
- name: Set up JDK 11
uses: actions/setup-java@v1
with:
java-version: '11'
- name: Build with Maven
run: mvn clean package

Monitoring and Logging

Monitoring Tools

Prometheus:

Open-source monitoring system.
Collects metrics from monitored targets.

prometheus:

config:

global:

scrape_interval: 15s

scrape_configs:

- job_name: 'spring-boot'

static_configs:

- targets: ['localhost:8080']

Grafana:

Visualization tool for time-series data.

Integrates with Prometheus.

Logging

Logback:

Default logging framework for Spring Boot.

xml [Copy code](#)

```
<configuration>
  <appender name="STDOUT"
class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%d{yyyy-MM-dd HH:mm:ss} - %msg%n</pattern>
    </encoder>
  </appender>
  <root level="INFO">
    <appender-ref ref="STDOUT" />
  </root>
</configuration>
```

ELK Stack (Elasticsearch, Logstash, Kibana):

Centralized logging solution.

Collects, parses, and visualizes logs.

yaml [Copy code](#)

```
filebeat.inputs:
- type: log
  paths:
    - /var/log/*.log
```

```
output.logstash:
  hosts: ["localhost:5044"]
```

```
docker run -d --name elasticsearch -p 9200:9200 -p 9300:9300
elasticsearch:7.10.1
docker run -d --name logstash -p 5044:5044 -v "$PWD/logstash.conf":/usr/
share/logstash/pipeline/logstash.conf logstash:7.10.1
docker run -d --name kibana -p 5601:5601 kibana:7.10.1
```

Summary

- **Building and Packaging:** Create executable JARs and WARs using Maven and Gradle, and manage dependencies effectively.
- **Containerization with Docker:** Dockerize Spring Boot applications and use Docker Compose for multi-container applications.
- **Deployment Strategies:** Deploy applications to cloud platforms (AWS, Google Cloud, Azure) and implement CI/CD pipelines with Jenkins and GitHub Actions.
- **Monitoring and Logging:** Monitor applications with Prometheus and Grafana, and implement centralized logging with the ELK stack.

These notes provide a comprehensive guide for deploying and managing Spring Boot applications in various environments, ensuring a smooth and efficient DevOps workflow.

This Spring Boot course provides a comprehensive overview of developing and deploying Spring Boot applications. Starting with basic concepts, it covers the entire development lifecycle, including data access, building RESTful web services, securing applications, testing, advanced topics like microservices, and finally, deployment and DevOps strategies. The course equips learners with practical skills and best practices for building robust and scalable Spring Boot applications, making it ideal for both beginners and experienced developers aiming to enhance their Spring Boot expertise.