Page 1: Introduction to Spring Boot

- **What is Spring Boot?**
  - Overview
  - Key features and benefits
- **Why use Spring Boot?**
  - Simplifies development
  - Reduces boilerplate code
  - Integrated with Spring ecosystem

Page 2: Setting Up Your Development Environment

- **Required Tools:**
  - Java Development Kit (JDK)
  - Integrated Development Environment (IDE) like IntelliJ IDEA or Eclipse
  - Maven or Gradle for project management
- **Installing Spring Boot:**
  - Using Spring Initializr
  - Manual setup

Page 3: Creating Your First Spring Boot Application

- **Using Spring Initializr:**
  - Generating a project
  - Importing the project into your IDE
- **Project Structure Overview:**
  - src/main/java
  - src/main/resources

Page 4: Main Application Class

- **The @SpringBootApplication Annotation:**
  - Explanation
  - Example code
- **Running the Application:**
  - Using the IDE
  - Using the command line

Page 5: Creating a Simple REST Controller

- **Understanding REST:**
  - REST principles
- **Creating a Controller:**
  - Using @RestController annotation
  - Mapping URLs with @GetMapping, @PostMapping, etc.
  - Example code

Page 6: Handling HTTP Requests

- **Path Variables and Request Parameters:**
  - @PathVariable
  - @RequestParam
  - Example code
- **Handling JSON Data:**
  - @RequestBody
  - @ResponseBody

Page 7: Service Layer

- **Creating a Service Class:**
  - Using @Service annotation
  - Example code
- **Injecting Services into Controllers:**
  - Using @Autowired annotation

Page 8: Data Access with Spring Data JPA

- **Introduction to JPA:**
  - Overview of JPA
- **Setting Up a Database:**
  - Configuring database properties
- **Creating Entities and Repositories:**
  - @Entity annotation
  - Extending JpaRepository
  - Example code

Page 9: Connecting to a Database

- **Application Properties:**
  - Configuring datasource in application.properties
  - Example configurations for MySQL, H2, etc.
- **Performing CRUD Operations:**
  - Save, Retrieve, Update, Delete operations
  - Example code

Page 10: Testing Your Application

- **Introduction to Testing in Spring Boot:**
  - Overview of testing frameworks (JUnit, Mockito)
- **Writing Unit Tests:**
  - Example unit test for a controller
- **Running Tests:**
  - Using IDE
  - Using Maven/Gradle

Page 1: Introduction to Spring Boot

What is Spring Boot? Spring Boot is an open-source Java-based framework used to create stand-alone, production-grade Spring-based applications. It is built on top of the Spring Framework, providing a simplified way to create web applications and microservices.

Key Features and Benefits:

- **Auto-configuration:** Automatically configures your Spring application based on the dependencies you have added to the project.
- **Standalone:** Spring Boot applications can be run independently without an external web server.
- **Production-ready:** Comes with embedded servers like Tomcat, Jetty, and Undertow, and provides metrics, health checks, and externalized configuration.
- **Opinionated defaults:** Provides default configurations that can be overridden, simplifying the development process.

Why Use Spring Boot?

- **Simplified Development:** Reduces the need for extensive configuration.
- **Reduced Boilerplate Code:** Minimizes repetitive coding tasks.
- **Integration with Spring Ecosystem:** Seamlessly integrates with other Spring projects like Spring Data, Spring Security, etc.

Page 2: Setting Up Your Development Environment

Required Tools:

- **Java Development Kit (JDK):** Ensure JDK 8 or higher is installed.
- **Integrated Development Environment (IDE):** Use IntelliJ IDEA, Eclipse, or any other preferred IDE.
- **Build Tool:** Maven or Gradle for project management and dependency management.

Installing Spring Boot:

1. **Using Spring Initializr:**
    - Go to Spring Initializr.
    - Select the project settings (e.g., Maven Project, Java, Spring Boot version).
    - Add dependencies (e.g., Spring Web).
    - Generate the project and download the ZIP file.
    - Extract the ZIP file and open it in your IDE.
2. **Manual Setup:**
    - Create a new Maven or Gradle project in your IDE.
    - Add the necessary Spring Boot dependencies to your `pom.xml` (for Maven) or `build.gradle` (for Gradle) file.

Using Maven with Spring Boot

Maven is a powerful build automation tool used primarily for Java projects. It simplifies the build process, dependency management, and project configuration. When working with Spring Boot, Maven helps streamline the development lifecycle by managing dependencies, building the application, and running tests.

Maven Project Structure

A typical Maven project has the following structure:

```
css
Copy code
spring-boot-project/
├── src/
│   ├── main/
│   │   ├── java/
│   │   └── resources/
│   └── test/
│       ├── java/
│       └── resources/
├── pom.xml
└── target/
```

`pom.xml` Configuration

The `pom.xml` file is the core of a Maven project. It defines the project configuration, dependencies, build plugins, and other settings. Below is an example `pom.xml` for a Spring Boot project:

```
xml
Copy code
<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
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>spring-boot-project</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.6.3</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <properties>
        <java.version>11</java.version>
    </properties>
    <dependencies>
        <!-- Spring Boot Starter Web -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <!-- Spring Boot Starter Data JPA -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <!-- Spring Boot Starter Test -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <!-- MySQL Connector -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <scope>runtime</scope>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <!-- Spring Boot Maven Plugin -->
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

Key Sections in `pom.xml`

- **Model Version**: Specifies the POM model version (`4.0.0` for most projects).
- **GroupId**: The unique identifier for the project's group (e.g., `com.example`).
- **ArtifactId**: The name of the project's artifact (e.g., `spring-boot-project`).
- **Version**: The version of the project (e.g., `0.0.1-SNAPSHOT`).
- **Parent**: The parent project for inheritance (`spring-boot-starter-parent`).
- **Properties**: Custom properties like Java version (`java.version`).
- **Dependencies**: Lists all project dependencies.
- **Build**: Configuration for the build process and plugins (e.g., `spring-boot-maven-plugin`).

Managing Dependencies

Dependencies are managed in the `<dependencies>` section of the `pom.xml`. Spring Boot provides a set of "starter" dependencies that simplify the inclusion of common libraries.

Example Dependencies:

- **Spring Boot Starter Web**: Includes libraries for building web applications, including RESTful services.
- **Spring Boot Starter Data JPA**: Includes libraries for working with JPA and Hibernate.
- **Spring Boot Starter Test**: Includes libraries for testing, including JUnit, Mockito, and Spring Test.
- **MySQL Connector**: A runtime dependency for connecting to MySQL databases.

Building and Running a Spring Boot Application

Maven provides several commands to manage the build lifecycle of a Spring Boot application.

1. **Building the Project**: Compiles the source code and packages the application into a JAR or WAR file.

```
sh
Copy code
mvn clean package
```

2. **Running the Application**: Executes the packaged application using the Spring Boot Maven Plugin.

```
sh
Copy code
mvn spring-boot:run
```

3. **Running Tests**: Executes the unit tests.

```
sh
Copy code
mvn test
```

Maven Plugins

Maven plugins extend the capabilities of the build process. The Spring Boot Maven Plugin is particularly useful for Spring Boot applications.

Spring Boot Maven Plugin:

```
xml
Copy code
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
```

This plugin allows you to:

- Run the application using `mvn spring-boot:run`.
- Package the application into an executable JAR or WAR file.
- Include a dependency-insight report to help manage dependencies.

Example Spring Boot Application with Maven

Main Application Class:

```
java
Copy code
@SpringBootApplication
public class SpringBootProjectApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringBootProjectApplication.class, args);
    }
}
```

Controller Class:

```
java
Copy code
@RestController
@RequestMapping("/api")
public class HelloController {
    @GetMapping("/hello")
    public String sayHello() {
        return "Hello, World!";
    }
}
```

Application Properties:

Create a file named `application.properties` in the `src/main/resources` directory to define application-specific properties.

```
properties
Copy code
server.port=8080
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=password
spring.jpa.hibernate.ddl-auto=update
```

Conclusion

Using Maven with Spring Boot simplifies the management of dependencies, builds, and the application lifecycle. By following the Maven conventions and utilizing the Spring Boot starters, you can quickly set up and maintain a robust Spring Boot application. The combination of Maven's powerful build capabilities and Spring Boot's productivity features allows for efficient and streamlined development.

Page 3: Creating Your First Spring Boot Application

Using Spring Initializr:

- Go to Spring Initializr.
- Fill in the project details (Group, Artifact, Name, Description, Package name).
- Add dependencies like "Spring Web".
- Click "Generate", download the project ZIP file, extract it, and open it in your IDE.

Project Structure Overview:

- **src/main/java:** Contains your Java source files.

- **src/main/resources:** Contains static resources, templates, and configuration files like `application.properties`.

Spring Boot Folder Structure

A typical Spring Boot project follows a standard directory layout that adheres to Maven conventions. Below is an explanation of the folder structure for a Spring Boot project.

Root Directory

At the root level of the project, you will find the following files and directories:

```
css
Copy code
spring-boot-project/
├── mvnw
├── mvnw.cmd
├── pom.xml
├── src/
└── .gitignore
```

- **mvnw and mvnw.cmd**: These are Maven Wrapper files, allowing you to run Maven commands without requiring Maven to be installed on the system.
- **pom.xml**: The Project Object Model (POM) file for Maven, containing project configuration, dependencies, plugins, and build information.
- **src/**: The main source directory for your project.
- **.gitignore**: A file specifying which files and directories should be ignored by Git.

src Directory

The `src` directory contains all the source code, configuration files, and resources for your Spring Boot application. It has the following structure:

```
css
Copy code
src/
├── main/
│   ├── java/
│   │   └── com/
│   │       └── example/
│   │           └── springboot/
│   │               ├── SpringBootProjectApplication.java
│   │               ├── controller/
│   │               ├── model/
│   │               ├── repository/
│   │               └── service/
│   │
│   └── resources/
│       ├── application.properties
│       ├── static/
│       └── templates/
└── test/
    └── java/
        └── com/
            └── example/
                └── springboot/
                    └── SpringBootProjectApplicationTests.java
```

src/main/java

This directory contains the Java source code for your application.

- **SpringBootProjectApplication.java**: The main class annotated with `@SpringBootApplication`. This is the entry point of the Spring Boot application.

```
java
Copy code
@SpringBootApplication
public class SpringBootProjectApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringBootProjectApplication.class, args);
    }
}
```

- **controller/**: Contains REST controllers, which handle HTTP requests.

```
java
Copy code
@RestController
@RequestMapping("/api")
public class ExampleController {
    @GetMapping("/hello")
    public String sayHello() {
        return "Hello, World!";
    }
}
```

- **model/**: Contains the application's data model, including entity classes.

```
java
Copy code
@Entity
public class Example {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    // Getters and Setters
}
```

- **repository/**: Contains repository interfaces for data access using Spring Data JPA.

```
java
Copy code
```

```
public interface ExampleRepository extends JpaRepository<Example, Long> {
}
```

- **service/**: Contains service classes, which contain the business logic of the application.

```
java
Copy code
@Service
public class ExampleService {
    private final ExampleRepository exampleRepository;

    @Autowired
    public ExampleService(ExampleRepository exampleRepository) {
        this.exampleRepository = exampleRepository;
    }

    public Example saveExample(Example example) {
        return exampleRepository.save(example);
    }
}
```

src/main/resources

This directory contains configuration files and other resources needed by your application.

- **application.properties**: The main configuration file for Spring Boot. You can define various application settings here.

```
properties
Copy code
server.port=8080
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=password
```

- **static/**: Contains static files such as HTML, CSS, and JavaScript. These files are served directly by the web server.
- **templates/**: Contains template files for rendering views, usually used with Thymeleaf or other templating engines.

src/test/java

This directory contains the test source code for your application.

- **SpringBootProjectApplicationTests.java**: A test class for the main application class. Spring Boot provides integration with testing frameworks like JUnit.

```
java
Copy code
@RunWith(SpringRunner.class)
@SpringBootTest
public class SpringBootProjectApplicationTests {

    @Test
    public void contextLoads() {
    }
}
```

Additional Notes

- **Maven Structure**: The folder structure adheres to Maven's standard directory layout, which separates source code (`src/main/java`) from resources (`src/main/resources`) and test code (`src/test/java`).
- **Spring Boot Conventions**: The structure follows Spring Boot conventions, allowing Spring Boot's autoconfiguration and component scanning to work seamlessly.

This folder structure ensures that your Spring Boot project is organized, maintainable, and scalable. By following these conventions, you can easily manage your application's components and configurations.

Page 4: Main Application Class

The @SpringBootApplication Annotation:

- Combines `@Configuration`, `@EnableAutoConfiguration`, and `@ComponentScan` annotations.
- Marks the main class of a Spring Boot application.

Example Code:

```
java
Copy code
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

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

Running the Application:

- **Using the IDE:**
  - Right-click the main class and select "Run".
- **Using the Command Line:**
  - Navigate to the project directory and run `./mvnw spring-boot:run` (for Maven) or `./gradlew bootRun` (for Gradle).

Page 5: Creating a Simple REST Controller

Understanding REST:

- REST (Representational State Transfer) is an architectural style for designing networked applications.
- Uses standard HTTP methods (GET, POST, PUT, DELETE) for CRUD operations.

Creating a Controller:

- **@RestController Annotation:** Marks the class as a REST controller.
- **Mapping URLs:** Use `@GetMapping`, `@PostMapping`, etc., to map HTTP requests to handler methods.

Example Code:

```
java
Copy code
package com.example.demo.controller;

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

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

Page 6: Handling HTTP Requests

Path Variables and Request Parameters:

- **@PathVariable:** Used to extract values from the URI.
- **@RequestParam:** Used to extract query parameters from the URL.

Example Code:

```
java
Copy code
package com.example.demo.controller;

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

@RestController
public class GreetingController {
    @GetMapping("/greeting/{name}")
    public String greet(@PathVariable String name,
@RequestParam(defaultValue = "Hello") String salutation) {
        return salutation + ", " + name + "!";
    }
}
```

Handling JSON Data:

- **@RequestBody:** Used to bind the HTTP request body to a method parameter.
- **@ResponseBody:** Used to bind the method return value to the web response body.

Page 7: Service Layer

Creating a Service Class:

- **@Service Annotation:** Marks the class as a service provider.

Example Code:

```
java
Copy code
package com.example.demo.service;

import org.springframework.stereotype.Service;

@Service
public class GreetingService {
    public String greet(String name, String salutation) {
        return salutation + ", " + name + "!";
    }
}
```

Injecting Services into Controllers:

- **@Autowired Annotation:** Used to inject the service into the controller.

Example Code:

```
java
Copy code
package com.example.demo.controller;

import com.example.demo.service.GreetingService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class GreetingController {
    @Autowired
    private GreetingService greetingService;

    @GetMapping("/greeting/{name}")
    public String greet(@PathVariable String name,
@RequestParam(defaultValue = "Hello") String salutation) {
        return greetingService.greet(name, salutation);
    }
}
```

Page 8: Data Access with Spring Data JPA

Introduction to JPA:

- JPA (Java Persistence API) is a specification for managing relational data in Java applications.

Setting Up a Database:

- Configure database properties in `application.properties` file.

  ```
  properties
  Copy code
  spring.datasource.url=jdbc:mysql://localhost:3306/mydatabase
  spring.datasource.username=root
  spring.datasource.password=root
  spring.jpa.hibernate.ddl-auto=update
  ```

Creating Entities and Repositories:

- **@Entity Annotation:** Marks a class as a JPA entity.
- **Extending JpaRepository:** Provides CRUD operations.

Example Code:

```
java
Copy code
package com.example.demo.model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String email;

    // Getters and setters
}
java
Copy code
package com.example.demo.repository;

import com.example.demo.model.User;
```

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface UserRepository extends JpaRepository<User, Long> {
}
```

Page 9: Connecting to a Database

Application Properties:

- Configure your database connection in `application.properties`.

Example Configuration for MySQL:

```
properties
Copy code
spring.datasource.url=jdbc:mysql://localhost:3306/mydatabase
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Performing CRUD Operations:

- **Save:**

  ```
  java
  Copy code
  User user = new User();
  user.setName("John Doe");
  user.setEmail("john.doe@example.com");
  userRepository.save(user);
  ```

- **Retrieve:**

  ```
  java
  Copy code
  List<User> users = userRepository.findAll();
  ```

- **Update:**

  ```
  java
  Copy code
  user.setName("Jane Doe");
  userRepository.save(user);
  ```

- **Delete:**

  ```
  java
  Copy code
  userRepository.deleteById(userId);
  ```

Page 10: Testing Your Application

Introduction to Testing in Spring Boot

Testing is a crucial part of application development, ensuring that your code behaves as expected and is free from bugs. Spring Boot supports various testing frameworks and provides tools to facilitate different types of testing, including unit testing, integration testing, and end-to-end testing.

Overview of Testing Frameworks

1. **JUnit**: JUnit is a popular testing framework for Java applications. It provides annotations to identify test methods and assertions to check the expected outcomes.
   - Annotations: `@Test`, `@Before`, `@After`, `@BeforeClass`, `@AfterClass`
   - Assertions: `assertEquals()`, `assertTrue()`, `assertFalse()`, `assertNotNull()`
2. **Mockito**: Mockito is a mocking framework that allows you to create mock objects for testing purposes. It helps in isolating the unit of work being tested by simulating the behavior of dependencies.
   - Creating Mocks: `@Mock`
   - Defining Behavior: `when()`
   - Verifying Interactions: `verify()`

Writing Unit Tests

Unit tests focus on testing individual components of the application, such as methods or classes, in isolation from other parts of the system.

Example Unit Test for a Controller

Let's write a unit test for a simple controller using JUnit and Mockito.

Controller Code:

```
java
Copy code
@RestController
@RequestMapping("/api")
public class GreetingController {

    @GetMapping("/greeting")
    public String getGreeting() {
        return "Hello, World!";
    }
}
```

Unit Test Code:

```
java
Copy code
@RunWith(SpringRunner.class)
@WebMvcTest(GreetingController.class)
public class GreetingControllerTest {
    @Autowired
    private MockMvc mockMvc;

    @Test
    public void testGetGreeting() throws Exception {
        mockMvc.perform(get("/api/greeting"))
                .andExpect(status().isOk())
                .andExpect(content().string("Hello, World!"));
    }
}
```

In this test:

- `@RunWith(SpringRunner.class)`: This annotation tells JUnit to run using Spring's testing support.
- `@WebMvcTest(GreetingController.class)`: This annotation sets up a web context with just the specified controller.
- `MockMvc`: This class allows you to perform HTTP requests and verify the results.

Running Tests

You can run your tests using an Integrated Development Environment (IDE) or a build tool like Maven or Gradle.

Using IDE

Most IDEs, like IntelliJ IDEA, Eclipse, or VS Code, have built-in support for running tests. Here's how you can run tests using an IDE:

1. **IntelliJ IDEA**:
   - Open the test class in the editor.
   - Right-click on the test class or test method.
   - Select "Run 'TestClassName'" or "Run 'testMethodName'".
2. **Eclipse**:
   - Open the test class in the editor.
   - Right-click on the test class or test method.
   - Select "Run As" -> "JUnit Test".
3. **VS Code**:
   - Ensure you have the Java Test Runner extension installed.
   - Open the test class.
   - Click on the "Run Test" or "Debug Test" icons in the editor.

Using Maven/Gradle

You can also run tests from the command line using Maven or Gradle.

1. **Using Maven**:
   - Open a terminal and navigate to the root directory of your project.
   - Run the following command to execute the tests:

     ```
     sh
     ```

```
Copy code
mvn test
```

2. **Using Gradle**:
   - Open a terminal and navigate to the root directory of your project.
   - Run the following command to execute the tests:

     ```
     sh
     Copy code
     ./gradlew test
     ```

Example Maven `pom.xml` Configuration for Testing

Ensure your `pom.xml` includes the necessary dependencies for testing:

```
xml
Copy code
<dependencies>
    <!-- Spring Boot Starter Test -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <!-- Mockito -->
    <dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-core</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Example Gradle `build.gradle` Configuration for Testing

Ensure your `build.gradle` includes the necessary dependencies for testing:

```
groovy
Copy code
dependencies {
    // Spring Boot Starter Test
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
    // Mockito
    testImplementation 'org.mockito:mockito-core'
}
```

By following these guidelines, you can effectively test your Spring Boot application, ensuring that it meets the required quality standards and behaves as expected in various scenarios.