

SPRING BOOT

1. Introduction to Spring Boot

1.1 What is Spring Boot?

Spring Boot is an **open-source, Java-based framework** used to create stand-alone, production-ready Spring applications with minimal configuration. Built on top of the Spring Framework, it simplifies bootstrapping and development by offering default settings and reducing boilerplate code.

1.2 Features of Spring Boot

- **Auto-Configuration:** Automatically configures your application based on the dependencies declared in the project.
- **Standalone Applications:** Enables creating applications that can run independently using embedded servers like Tomcat or Jetty.
- **Production-Ready:** Includes built-in features like metrics, health checks, and external configuration via Spring Boot Actuator.

- **Opinionated Defaults:** Reduces the need for manual configurations with pre-defined, best-practice defaults.
 - **No XML Configuration:** Encourages convention over configuration using annotations instead of XML files.
-

1.3 Advantages of Using Spring Boot

-  **Rapid Application Development**
 -  **Auto-Configuration**
 -  **Embedded Web Servers**
 - Supports Tomcat, Jetty, Undertow
 - Run applications using: `java -jar app.jar`
 -  **Microservices-Friendly**
 -  **Production-Ready Features**
 - Health checks, metrics, externalized config via Actuator
 -  **Reduced Boilerplate Code**
 -  **Easy Dependency Management**
 -  **Seamless Integration with Spring Ecosystem**
 - Compatible with Spring Data JPA, Spring Security, Spring WebFlux, Spring Batch
 -  **Flexible Configuration**
 - Supports `application.properties` and `application.yml`
 -  **Community and Ecosystem Support**
-

1.4 Use Cases





Spring Boot is used in a wide variety of domains:

- RESTful API development
- Microservices architecture
- Rapid prototyping
- Enterprise web applications
- Cloud-native applications

Real Life Examples:

- ♦ E-Commerce Platforms
- ♦ Banking & Financial Services
- ♦ Healthcare Systems
- ♦ Travel and Booking Platforms
- ♦ Learning Management Systems (LMS)
- ♦ Logistics & Supply Chain Systems
- ♦ Content Management Systems (CMS)
- ♦ HR & Payroll Systems
- ♦ IoT Backend Services
- ♦ Microservices in Large Enterprises

Language Support:

-  Java (Primary Language)
-  Kotlin
-  Groovy
-  Scala (Limited Support)

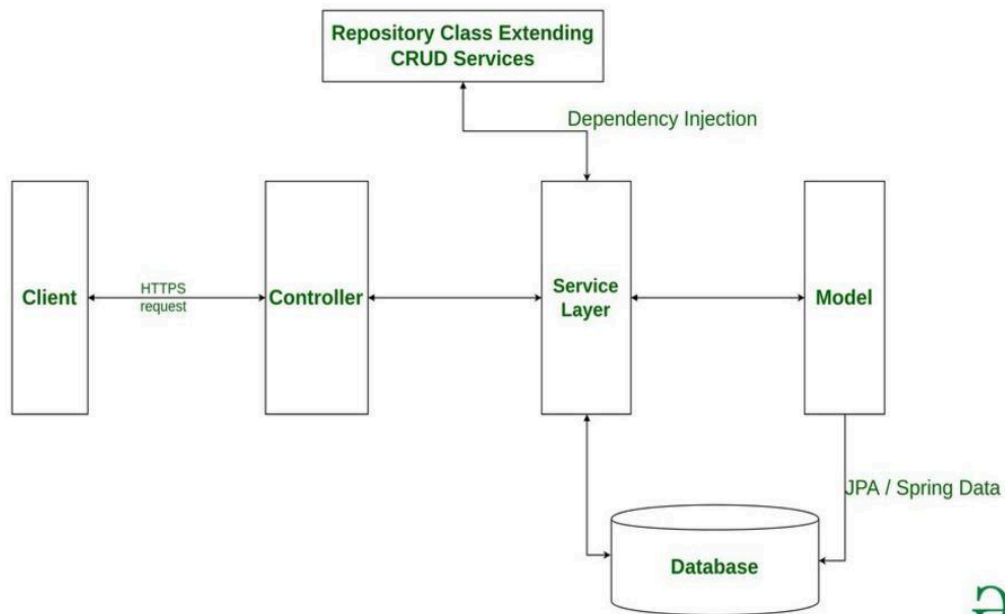
1.5 Spring Boot vs Spring Framework

Key Differences

Feature / Aspect	Spring Framework	Spring Boot
Setup & Configuration	Manual XML/Java-based configuration	Auto-configuration, zero XML
Application Startup	Requires external servlet container	Comes with embedded servers (Tomcat, Jetty, Undertow)
Dependency Management	Developer-managed dependencies	Pre-configured starter dependencies
Boilerplate Code	More boilerplate and configuration	Minimal configuration and boilerplate
Project Structure	Developer decides from scratch	Convention over configuration
Production Readiness	Requires manual setup of monitoring tools	Built-in Actuator endpoints for health, metrics, etc.
Build Output	WAR (typically)	Executable JAR by default
Learning Curve	Steeper for beginners	Easier and quicker to get started
CLI Support	No	Yes, via Spring Boot CLI
Microservices Development	Needs custom setup	Tailored for microservices architecture

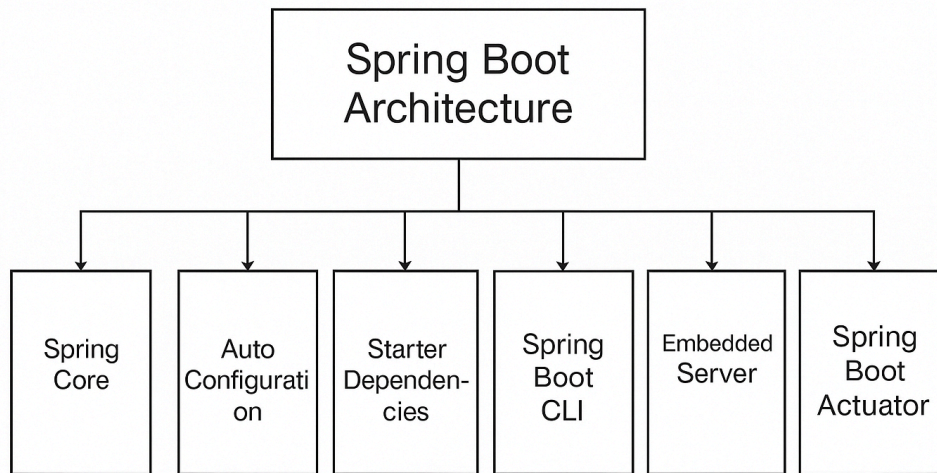
1.5 Spring Boot Architecture

Spring Boot flow architecture



Spring Boot architecture is based on:

- **Spring Core:** Base dependency injection and AOP framework
- **Auto Configuration:** Automatically configures beans based on classpath
- **Starter Dependencies:** Predefined sets of dependencies for various features (web, data, etc.)
- **Spring Boot CLI:** Command-line tool to run and test applications
- **Embedded Server:** Tomcat, Jetty, or Undertow
- **Spring Boot Actuator:** Provides operational endpoints for monitoring and management



Topics Summary:



1. Introduction

- 1.1. What is Spring Boot?**
- 1.2. Features of Spring Boot**
- 1.3. Advantages and Use cases of using Spring Boot**
- 1.4. Spring Boot vs Spring Framework**
- 1.5. Spring Boot Architecture**

➤ **2. Getting Started**

- 2.1. Setting up Spring Boot Project (Spring Initializr, Maven, Gradle)**
- 2.2. Project Structure Overview**
- 2.3. Application Entry Point (@SpringBootApplication)**
- 2.4. Running the Application**

➤ **3. Core Concepts**

3.1. Spring Boot Annotations

- 1. @SpringBootApplication**
- 2. @Component, @Service, @Repository, @Controller, @RestController**
- 3. @Autowired, @Qualifier**

4. @Configuration, @Bean

3.2. Dependency Injection (DI)

3.3. Inversion of Control (IoC)

➤ 4. Configuration

4.1. application.properties vs application.yml

4.2. External Configuration (env vars, CLI args)

4.3. Profiles (@Profile)

4.4. Property Injection: @Value, @ConfigurationProperties

➤ 5. Spring Boot Starters

5.1. What are Starters?

5.2. Common Starters

1. spring-boot-starter-web

2. spring-boot-starter-data-jpa

3. spring-boot-starter-security

4. spring-boot-starter-test, etc.

➤ 6. Web Development (Spring MVC)

6.1. Creating REST APIs (@RestController)

6.2. Handling Request Parameters

@PathVariable, @RequestParam, @RequestBody

6.3. ResponseEntity and Status Codes

6.4. Exception Handling

@ControllerAdvice, @ExceptionHandler

6.5. Content Negotiation

➤ 7. Data Access

7.1. Spring Data JPA

1. Entity Classes (@Entity)

2. Repositories (JpaRepository, CrudRepository)

3. Custom Queries (@Query)

4. Pagination and Sorting

7.2. JDBC with Spring Boot

7.3. MongoDB / NoSQL

7.4. Transactions (@Transactional)

➤ 8. Database Configuration

8.1. H2 In-Memory Database

8.2. MySQL/PostgreSQL setup

8.3. Connection Pooling (HikariCP)

8.4. Schema Initialization (`schema.sql`, `data.sql`)

➤ 9. Validation

9.1. Bean Validation (`javax.validation`)

9.2. Using `@Valid`, `@Validated`

9.3. Custom Validators

➤ 10. Security

10.1. Spring Security Basics

10.2. Authentication and Authorization

10.3. Password Encoding

10.4. JWT Token-based Security

10.5. Role-Based Access Control

➤ 11. Testing

11.1. Unit Testing with JUnit

11.2. Mocking with Mockito

11.3. Integration Testing

11.4. Web Layer Testing (`@WebMvcTest`)

11.5. Data Layer Testing (`@DataJpaTest`)

➤ 12. Logging

12.1. SLF4J and Logback

12.2. Configuring Log Levels

12.3. External Log Configuration

➤ 13. Actuator & Monitoring

13.1. Spring Boot Actuator Overview

13.2. Common Actuator Endpoints

13.3. Custom Metrics

13.4. Prometheus/Grafana Integration

➤ 14. Error Handling

14.1. Default Error Handling

14.2. Custom Error Pages

14.3. Global Exception Handling

➤ 15. Developer Tools

15.1. Spring Boot DevTools

15.2. Auto Restart

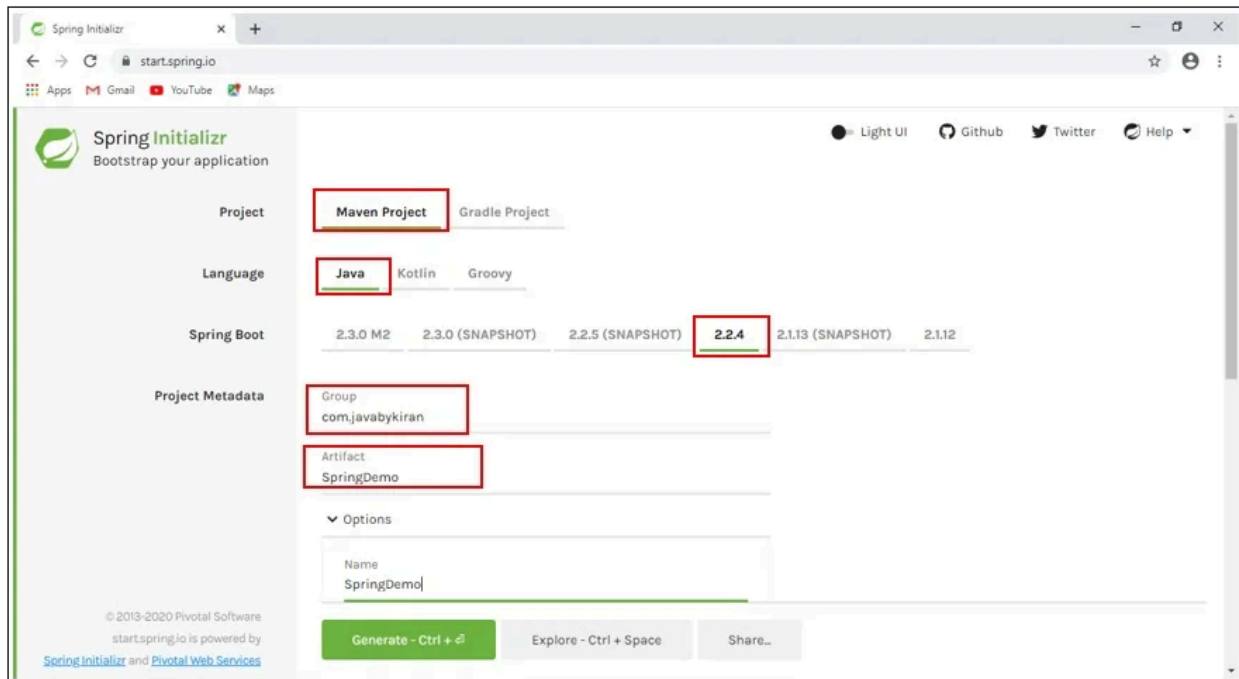
15.3. Live Reload

- **16. Caching**
 - 16.1. Using `@Cacheable`, `@CacheEvict`
 - 16.2. Cache Providers (Ehcache, Caffeine, Redis)
 - **17. Scheduling & Async**
 - 17.1. Scheduling Tasks (`@Scheduled`)
 - 17.2. Asynchronous Execution (`@Async`)
 - **18. Messaging**
 - 18.1. RabbitMQ
 - 18.2. Kafka
 - 18.3. JMS Integration
 - **19. File Upload/Download**
 - 19.1. Multipart File Upload
 - 19.2. File Storage and Download Handling
 - **20. Frontend Integration**
 - 20.1. Serving Static Files
 - 20.2. Using Thymeleaf
 - 20.3. Enabling CORS
 - **21. Build & Deployment**
 - 21.1. Building JAR/WAR
 - 21.2. Dockerizing Spring Boot Application
 - 21.3. Deploying to Cloud (AWS, Azure, GCP)
 - 21.4. CI/CD Integration
 - **22. Advanced Topics**
 - 22.1. Microservices with Spring Boot
 - 1. Spring Cloud
 - 2. Eureka (Service Discovery)
 - 3. Spring Cloud Gateway / Zuul
 - 4. Circuit Breaker (Resilience4j, Hystrix)
 - 5. Config Server
 - 22.2. ApplicationEventPublisher & Event Listeners
 - 22.3. Creating Custom Starters
-

2. Getting Started

2.1. Setting up Spring Boot Project (Spring Initializr, Maven, Gradle)

<https://start.spring.io/>



The screenshot shows the Spring Initializr web application interface. The browser address bar displays 'start.spring.io'. The page features a sidebar on the left with the Spring Initializr logo and the text 'Bootstrap your application'. The main content area is divided into sections for Project, Language, Spring Boot, and Project Metadata. The 'Project' section has 'Maven Project' selected. The 'Language' section has 'Java' selected. The 'Spring Boot' section shows version options with '2.2.4' highlighted. The 'Project Metadata' section includes input fields for 'Group' (com.javabykiran) and 'Artifact' (SpringDemo). Below these is an 'Options' section with a 'Name' field containing 'SpringDemo'. At the bottom, there are three buttons: 'Generate - Ctrl + G', 'Explore - Ctrl + Space', and 'Share...'. The footer contains copyright information for Pivotal Software and mentions that start.spring.io is powered by Spring Initializr and Pivotal Web Services.

Spring Initializr
Bootstrap your application

Project: **Maven Project** | Gradle Project

Language: **Java** | Kotlin | Groovy

Spring Boot: 2.3.0 M2 | 2.3.0 (SNAPSHOT) | 2.2.5 (SNAPSHOT) | **2.2.4** | 2.1.13 (SNAPSHOT) | 2.1.12

Project Metadata

Group: com.javabykiran

Artifact: SpringDemo

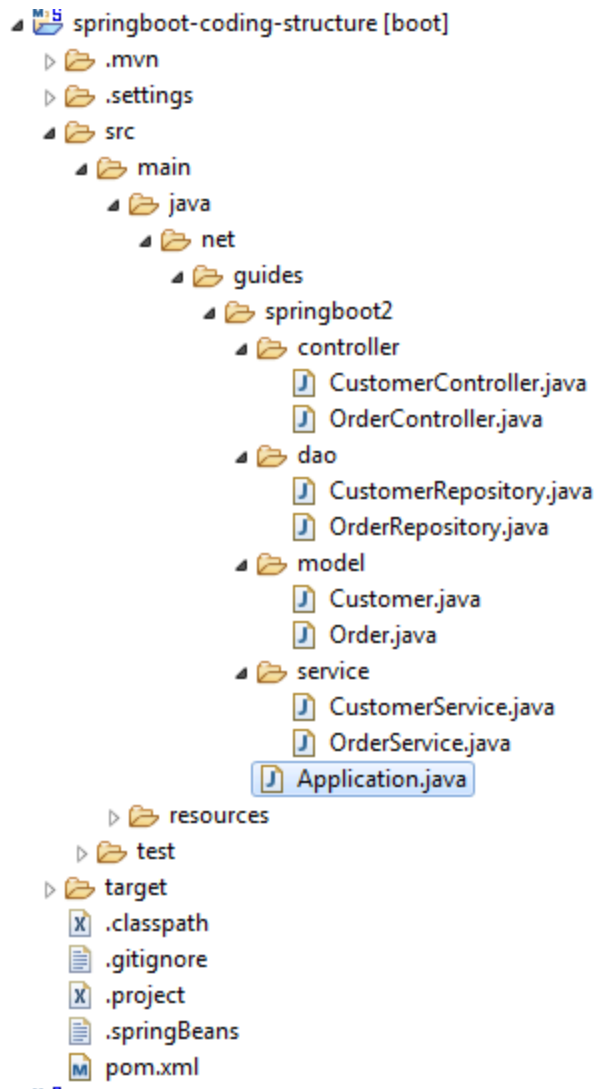
Options

Name: SpringDemo

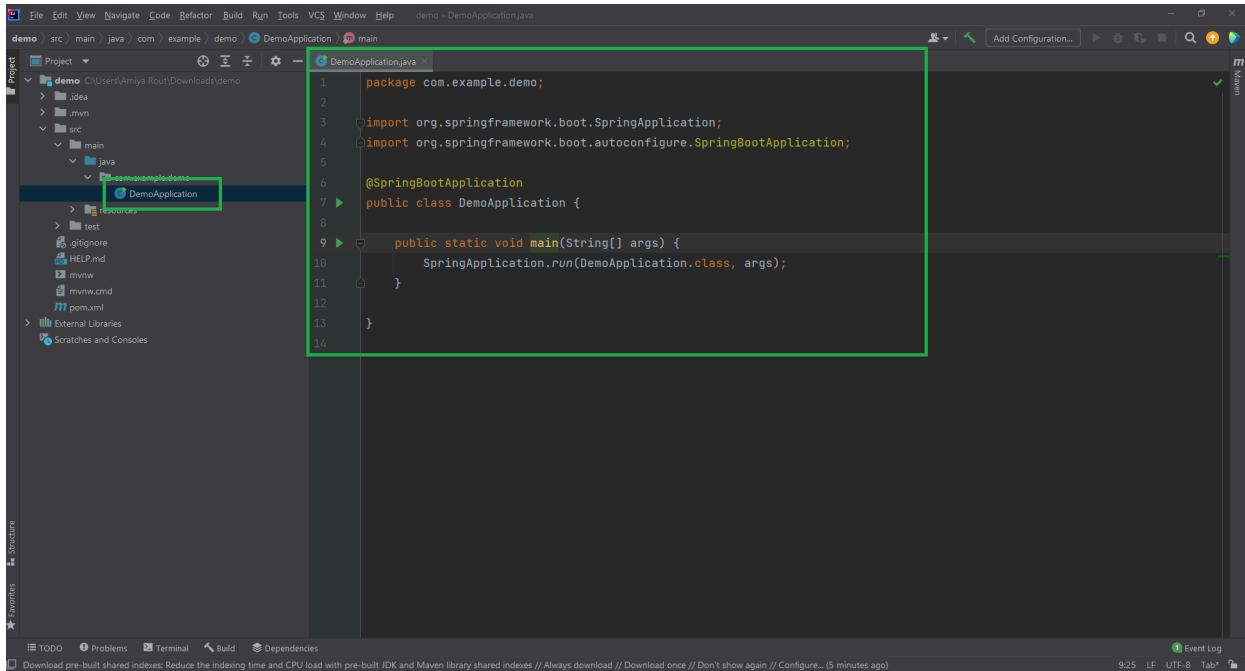
Generate - Ctrl + G | Explore - Ctrl + Space | Share...

© 2013-2020 Pivotal Software
start.spring.io is powered by
Spring Initializr and Pivotal Web Services

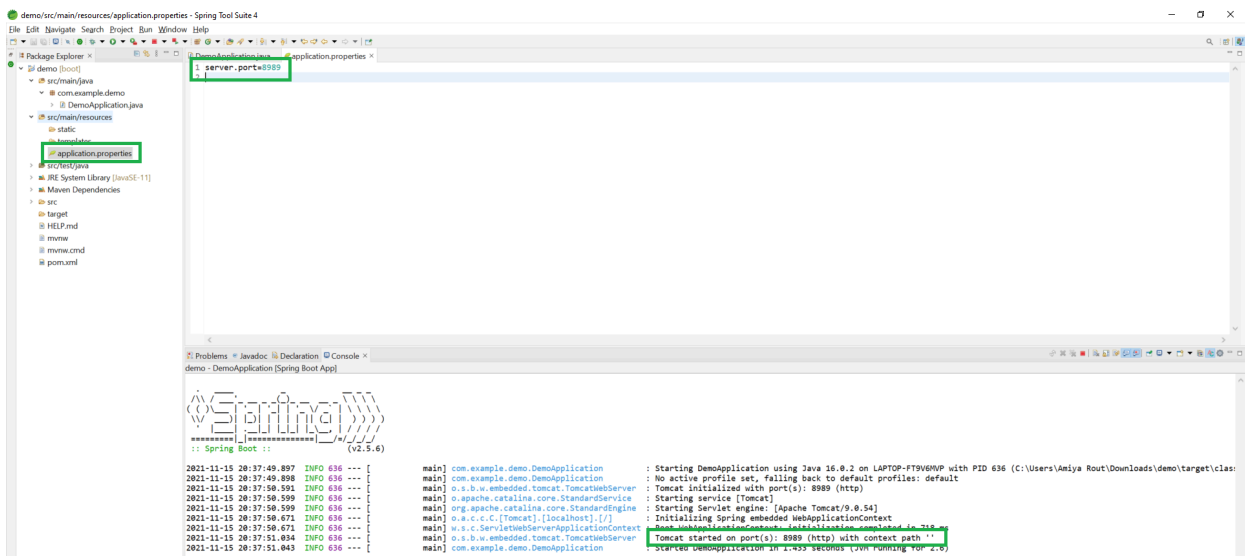
2.2. Project Structure Overview



2.3. Application Entry Point (@SpringBootApplication)



2.4. Running the Application



3. Core Concepts

3.1. Spring Boot Annotations

1. @SpringBootApplication

The **@SpringBootApplication** annotation is the primary entry point of any Spring Boot application. It is a convenience annotation that combines three commonly used annotations in Spring:

@SpringBootApplication = @Configuration + @EnableAutoConfiguration + @ComponentScan

Declaration

@SpringBootApplication

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

2. @Component, @Service, @Repository, @Controller,
@RestController

1. @Component

Definition

@Component is a generic stereotype annotation that marks a class as a Spring-managed component.

```
@Component

public class MyComponent {

    // Business logic

}
```

Use Case

Used when the class doesn't clearly fall into service, repository, or controller layers.

2. @Service

Definition

@Service is a specialization of **@Component** used to annotate **service layer** classes that contain business logic.

```
@Service

public class OrderService {

    public void processOrder() {

        // Business logic

    }

}
```

Use Case

- Encapsulates business logic
- Makes the code more readable and structured
- Can integrate with AOP (for transaction, logging, etc.)

3. @Repository

Definition

@Repository is a specialization of **@Component** used for **DAO (Data Access Object)** classes. It also provides exception translation for database errors.

@Repository

```
public class ProductRepository {  
  
    public void save(Product product) {  
  
        // Save to DB  
  
    }  
  
}
```

Use Case

- Used for data persistence
- Automatically translates exceptions to Spring's `DataAccessException`

4. @Controller

Definition

@Controller is used to define **web layer classes** in Spring MVC. It is used for rendering **views (HTML)** with model data.

@Controller

```
public class HomeController {  
  
    @GetMapping("/")  
  
    public String index(Model model) {  
  
        model.addAttribute("message", "Welcome");  
  
        return "index"; // returns view name  
  
    }  
  
}
```

Use Case

Used when working with traditional web applications (Thymeleaf, JSP, etc.)

5. @RestController

Definition

@RestController = **@Controller** + **@ResponseBody**

Used to define **RESTful web services**. The return values are serialized as JSON or XML and sent directly to the client.

@RestController

```
public class UserController {  
  
    @GetMapping("/users")  
  
    public List<User> getAllUsers() {  
  
        return userService.getAll();  
  
    }  
  
}
```

Use Case

- Used in REST APIs
- Automatically serializes Java objects to JSON

Summary Table

Annotation	Layer	Inherits From	Purpose	Returns Response Body
@Component	Generic	-	General-purpose component	No
@Service	Service Layer	@Component	Business logic and services	No
@Repository	Data Layer	@Component	DAO & exception translation	No

<code>@Controller</code>	Web (MVC)	<code>@Component</code>	Handles web requests & returns views	No (uses View Resolver)
<code>@RestController</code>	Web (REST API)	<code>@Controller</code>	Returns JSON/XML directly to the client	Yes

3. `@Autowired`, `@Qualifier`

`@Autowired` is used to automatically wire dependencies.

`@Qualifier` is used to resolve conflicts when multiple beans of the same type are available.

1. `@Autowired`

Definition

`@Autowired` is used to automatically inject a bean from the Spring container into another bean by **type**.

`@Component`

```
public class CarService {

    @Autowired

    private Engine engine;

    // engine will be injected automatically

}
```

Where It Can Be Used

- On fields

- On constructors
- On setter methods

Example (Constructor Injection)

```
@Service

public class OrderService {

    private final PaymentService paymentService;

    @Autowired

    public OrderService(PaymentService paymentService) {

        this.paymentService = paymentService;

    }

}
```

2. @Qualifier

Definition

@Qualifier is used with **@Autowired** to **specify which bean to inject** when multiple candidates of the same type are available.

```
@Component("dieselEngine")

public class DieselEngine implements Engine { }

@Component("petrolEngine")
```

```
public class PetrolEngine implements Engine { }
```

```
@Service
```

```
public class CarService {
```

```
    @Autowired
```

```
    @Qualifier("dieselEngine")
```

```
    private Engine engine;
```

```
}
```

Use Case

- Disambiguating among multiple beans of the same type.
- Explicitly specifying the bean to be injected.

4. @Configuration, @Bean

1. @Configuration

Definition

@Configuration is a class-level annotation that indicates the class contains **bean definitions** for the Spring IoC container. It serves as a **replacement for XML-based configuration**.

```
@Configuration
```

```
public class AppConfig {
```

```
    // Bean definitions go here
```

```
}

@Configuration

public class AppConfig {

    // Bean definitions go here

}
```

2. @Bean

Definition

`@Bean` is a method-level annotation used inside a `@Configuration`-annotated class to define a **bean explicitly**.

```
@Bean

public DataSource dataSource() {

    return new HikariDataSource();

}


@Configuration

public class AppConfig {

    @Bean

    public UserService userService() {

        return new UserServiceImpl(userRepository());

    }
```

```
@Bean

public UserRepository userRepository() {

    return new InMemoryUserRepository();

}

}
```

3.2. Dependency Injection (DI)

Dependency Injection is a programming technique where an object receives its dependencies from an external source rather than creating them internally.

Why Use Dependency Injection?

- Promotes **loose coupling** between classes
- Improves **testability** and **maintainability**
- Encourages **modular** and **reusable** code
- Follows **SOLID principles** (especially the "D" — Dependency Inversion Principle)

Types of Dependency Injection in Spring

Type	Description	Syntax Example
Constructor Injection	Dependencies are passed through a constructor.	Preferred for mandatory dependencies

Setter Injection	Dependencies are set via setter methods.	Suitable for optional dependencies
Field Injection	Dependencies are injected directly into fields using <code>@Autowired</code> .	Quick but less recommended for testing

Example: Constructor Injection
@Service

```
public class OrderService {  
  
    private final PaymentService paymentService;  
  
    @Autowired  
    public OrderService(PaymentService paymentService) {  
        this.paymentService = paymentService;  
    }  
  
    public void placeOrder() {  
        paymentService.processPayment();  
    }  
}
```

Here:

- `OrderService` depends on `PaymentService`.
- The Spring container injects `PaymentService` at runtime.



Dependency Injection vs. Manual Instantiation



Manual Instantiation	With Spring DI
<pre>PaymentService ps = new PaymentService();</pre>	<pre>@Autowired PaymentService paymentService;</pre>
Tightly coupled code	Loosely coupled, easily testable

Spring Annotations for DI

Annotation	Purpose
<code>@Autowired</code>	Marks a dependency to be injected by Spring
<code>@Qualifier</code>	Resolves ambiguity when multiple beans of same type
<code>@Inject</code> (JSR-330)	Java standard equivalent of <code>@Autowired</code>
<code>@Value</code>	Injects values from properties or expressions

Benefits of Using DI in Spring Boot

-  **Loose Coupling:** Classes are not tightly bound to specific implementations.
-  **Testability:** Easier to write unit tests by injecting mock dependencies.

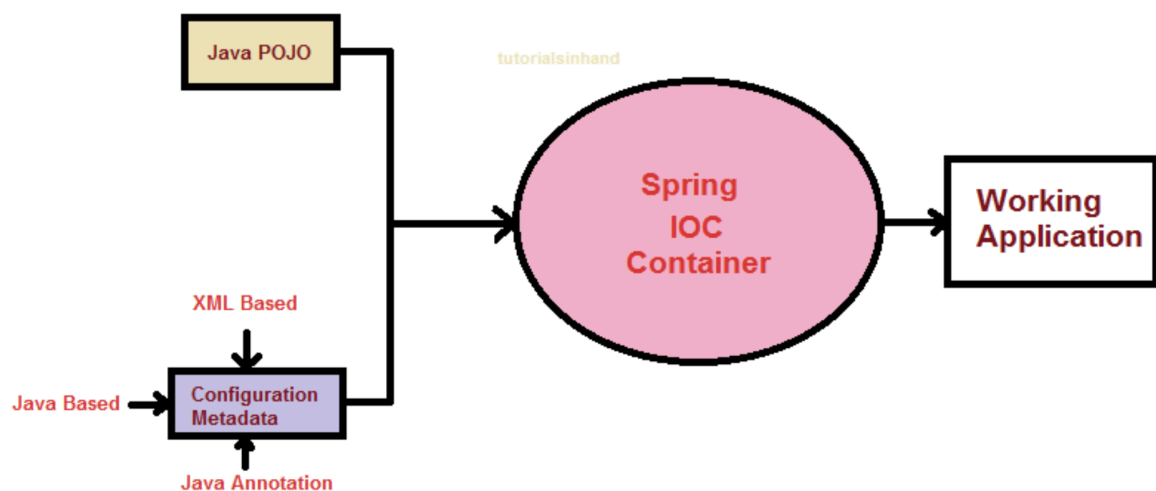
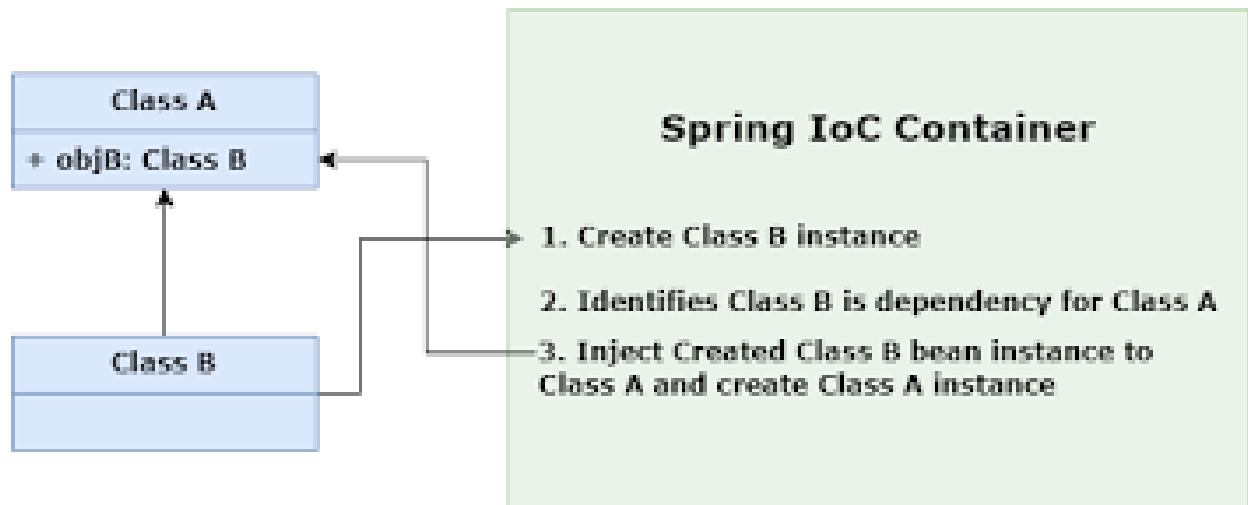
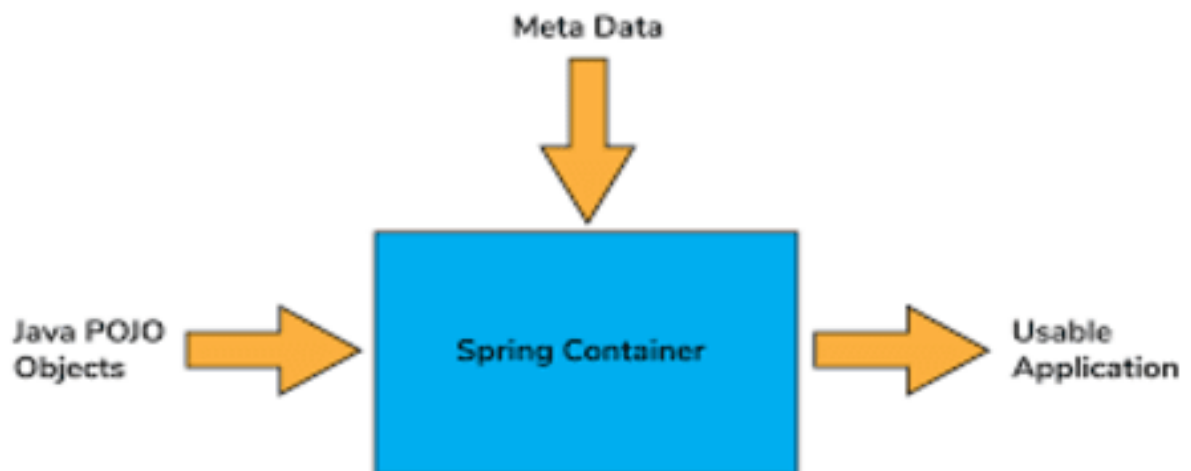
-  **Readability:** Clearly defines class dependencies.
 -  **Maintainability:** Changes to one class do not affect dependent classes directly.
-

3.3. Inversion of Control (IoC)

Inversion of Control (IoC) is a design principle in which the control of object creation and dependency resolution is transferred from the program (developer) to a framework or container.

How IoC Works in Spring

1. The developer defines application components (beans) using annotations or configuration.
2. Spring's IoC container:
 - Scans the classpath.
 - Detects annotated components.
 - Creates and wires dependencies automatically.
3. The container manages the entire lifecycle of those objects.



Working of Spring Container

4. Configuration

Overview

Configuration is a core aspect of Spring Boot that enables you to customize application behavior without changing the source code. Spring Boot provides multiple ways to configure applications, including property files, YAML files, environment variables, command-line arguments, and more.

This section explores key configuration concepts:

- `application.properties` vs `application.yml`
- External configuration sources
- Profiles for environment-specific settings
- Property injection using `@Value` and `@ConfigurationProperties`

4.1 `application.properties` vs `application.yml`

Spring Boot supports two main configuration file formats:

Feature	<code>application.properties</code>	<code>application.yml</code>
---------	-------------------------------------	------------------------------

Format	Key-value pairs	Hierarchical, indentation-based
Readability for complex configs	Less readable for nested values	More readable for nested/grouped configs
Syntax	key=value	YAML syntax using indentation

Example: application.properties

```
server.port=8081

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

spring.datasource.username=root

spring.datasource.password=pass
```

Example: application.yml

```
server:

  port: 8081

spring:

  datasource:

    url: jdbc:mysql://localhost:3306/db

    username: root

    password: pass
```

4.2 External Configuration

Spring Boot allows external configuration via:

Source	Priority (High → Low)
Command-line arguments	✓ Highest priority
Environment variables	✓ OS/environment-specific configs
application.properties/ yaml	✓ Default file-based config
Profile-specific files	✓ Based on environment

Example: Using Environment Variables

```
export SPRING_DATASOURCE_URL=jdbc:mysql://localhost:3306/db
```

Spring automatically maps it to `spring.datasource.url`.

Example: Using Command-line Arguments

```
java -jar myapp.jar --server.port=9090 --logging.level.root=DEBUG
```

These override values from config files.

4.3 Profiles (@Profile)

Spring Profiles allow you to define **environment-specific configurations** (e.g., dev, test, prod).

How to Use Profiles

- Create profile-specific property files:
 - `application-dev.properties`
 - `application-prod.yml`
- Activate profile using:
 - `Application.properties`:

`spring.profiles.active=dev`

- Or via CLI:

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

Example: Conditional Bean with @Profile

`@Configuration`

`@Profile("dev")`

```
public class DevConfig {  
  
    @Bean  
  
    public String dataSource() {  
  
        return "H2 In-Memory DB for Dev";  
  
    }  
  
}
```

```
@Configuration

@Profile("prod")

public class ProdConfig {

    @Bean

    public String dataSource() {

        return "MySQL DB for Production";

    }

}
```

4.4 Property Injection: @Value vs @ConfigurationProperties

Spring provides two ways to inject properties into Java classes.

1. @Value Annotation

- Injects individual values from properties or YAML files.
- Lightweight and direct.

```
@Component
```

```
public class AppSettings {

    @Value("${server.port}")

    private int port;
```

```
@Value("${custom.message:Default Message}")

private String message; // Fallback value if not defined

}
```

2. @ConfigurationProperties Annotation

- Maps a **group of related properties** into a POJO.
- Cleaner for structured configs.

YAML Example

```
app:

  name: MyApp

  version: 1.0
```

```
@Component

@ConfigurationProperties(prefix = "app")

public class AppProperties {

    private String name;

    private String version;


    // Getters and Setters

}
```

Enable Binding (if not using @SpringBootApplication)

`@EnableConfigurationProperties(AppProperties.class)`

Comparison Table

Feature	@Value	@ConfigurationProperties
Purpose	Inject individual values	Bind multiple related properties
Use Case	Simple property access	Structured/grouped configuration
Validation Support	Limited	Strong (with JSR-303 annotations)
Default Value Support	Yes <code>@Value("\${key:default}")</code>	No (use validation instead)

5. Spring Boot Starters

5.1 What are Starters?

A **Starter** is a **pre-defined set of dependencies** bundled under a single starter module, designed to support specific application needs (e.g., web, JPA, security).

Key Benefits

- Simplifies Maven/Gradle configuration
- Reduces boilerplate dependency management

- Ensures compatible versions for Spring Boot

Example pom.xml

```
<!-- Instead of specifying multiple web libraries manually -->  
  
<dependency>  
  
    <groupId>org.springframework.boot</groupId>  
  
    <artifactId>spring-boot-starter-web</artifactId>  
  
</dependency>
```

5.2 Common Starters

Below are the most frequently used Spring Boot starters:

5.2.1 **spring-boot-starter-web**

Used to build **web applications and RESTful services** using Spring MVC.

Includes:

- Spring MVC
- Jackson (for JSON binding)
- Embedded Tomcat (by default)

Example Use Case:

```
@RestController  
  
public class HelloController {  
  
    @GetMapping("/hello")  
  
    public String greet() {
```

```
        return "Hello, World!";
    }
}
```

5.2.2 **spring-boot-starter-data-jpa**

Used to enable **Spring Data JPA** for data access with relational databases.

Includes:

- Spring Data JPA
- Hibernate
- Spring ORM

Example Use Case:

@Entity

```
public class User {

    @Id @GeneratedValue
    private Long id;

    private String name;

}
```

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

5.2.3 **spring-boot-starter-security**

Used to add **authentication and authorization** to your application.

Includes:

- Spring Security

Example Use Case:

Secures all endpoints by default. You can configure custom login:

@Configuration

```
public class SecurityConfig extends WebSecurityConfigurerAdapter {  
  
    @Override  
  
    protected void configure(HttpSecurity http) throws Exception {  
  
        http.authorizeRequests()  
  
            .anyRequest().authenticated()  
  
            .and().formLogin();  
  
    }  
  
}
```

5.2.4 spring-boot-starter-test

Used to write **unit and integration tests**.

Includes:

- JUnit 5
- Mockito
- Spring Test
- AssertJ

Example Use Case:

@SpringBootTest

```
public class MyServiceTest {
```

```
    @Autowired
```

```
private MyService myService;

@Test
void testLogic() {
    assertEquals("expected", myService.doSomething());
}
}
```

Other Useful Starters

Starter	Purpose
<code>spring-boot-starter-thymeleaf</code>	HTML template rendering
<code>spring-boot-starter-mail</code>	Sending emails
<code>spring-boot-starter-actuator</code>	Monitoring and managing applications
<code>spring-boot-starter-validation</code>	Bean validation using Hibernate Validator

6. Web Development with Spring Boot (Spring MVC)

Overview

Spring Boot supports full-stack **web development** using **Spring MVC**, allowing you to build RESTful APIs and web applications easily. This section covers core web development concepts such as REST controllers, request handling, exception management, and content negotiation.

6.1 Creating REST APIs (@RestController)

`@RestController` combines `@Controller` and `@ResponseBody`, enabling RESTful endpoints.

Example

```
@RestController
@RequestMapping("/api/users")
public class UserController {

    @GetMapping("/{id}")
    public User getUser(@PathVariable Long id) {
        return new User(id, "John Doe");
    }
}
```

6.2 Handling Request Parameters

6.2.1 @PathVariable, @RequestParam, @RequestBody

Annotation	Description	Example Usage
<code>@PathVariable</code>	Binds a URL segment to a method parameter	<code>/users/{id}</code> → <code>@PathVariable Long id</code>
<code>@RequestParam</code>	Extracts query parameters from the URL	<code>/users?role=admin</code> → <code>@RequestParam String role</code>
<code>@RequestBody</code>	Binds request JSON/XML body to an object	JSON POST → <code>@RequestBody User user</code>

```
@PostMapping("/create")
public ResponseEntity<String> createUser(@RequestBody User user) {
    return ResponseEntity.ok("User created: " + user.getName());
}

@GetMapping("/find")
public String findUser(@RequestParam String role) {
    return "Finding user with role: " + role;
}
```

6.3 ResponseEntity and Status Codes

`ResponseEntity` allows customizing the response body, headers, and HTTP status.

Example

```
@GetMapping("/{id}")
public ResponseEntity<User> getUser(@PathVariable Long id) {
    User user = userService.findById(id);
    if (user == null) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).build();
    }
    return ResponseEntity.ok(user);
}
```

6.4 Exception Handling

6.4.1 @ControllerAdvice and @ExceptionHandler

Global exception handling using `@ControllerAdvice` and specific error response logic using `@ExceptionHandler`.

Example

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(UserNotFoundException.class)
    public ResponseEntity<String> handleUserNotFound(UserNotFoundException ex) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(ex.getMessage());
    }
}
```

6.5 Content Negotiation

Spring automatically supports **content negotiation**, allowing responses in different formats (e.g., JSON, XML) based on **Accept** headers.

Example

```
@GetMapping(value = "/info", produces = { MediaType.APPLICATION_JSON_VALUE,
    MediaType.APPLICATION_XML_VALUE })
public User getInfo() {
    return new User(1L, "Alice");
}
```

- Request **Accept: application/json** → JSON
 - Request **Accept: application/xml** → XML
-

7. Data Access in Spring Boot

Overview

Spring Boot simplifies data access by providing integration with various data sources like relational databases (via JPA and JDBC) and NoSQL databases (e.g., MongoDB). It also supports robust transaction management.

7.1 Spring Data JPA

Spring Data JPA abstracts data access layers by providing ready-to-use repositories and powerful query capabilities.

1 Entity Classes (@Entity)

@Entity marks a Java class as a table in the database.

Example

@Entity

```
public class User {  
  
    @Id @GeneratedValue  
    private Long id;  
  
    private String name;  
  
}
```

2 Repositories (JpaRepository, CrudRepository)

Spring Data provides interfaces for CRUD operations without boilerplate code.

Example

```
public interface UserRepository extends JpaRepository<User, Long> {  
  
    List<User> findByName(String name);  
  
}
```


3 Custom Queries (@Query)

Define complex queries using JPQL or native SQL.

Example

```
@Query("SELECT u FROM User u WHERE u.name LIKE %:name%")
```

```
List<User> searchByName(@Param("name") String name);
```

4 Pagination and Sorting

Built-in support via `Pageable` and `Sort`.

Example

```
Page<User> page = userRepository.findAll(PageRequest.of(0, 10, Sort.by("name")));
```

7.2 JDBC with Spring Boot

Use `JdbcTemplate` for fine-grained SQL operations.

Example

```
@Repository
```

```
public class UserDao {
```

```
    @Autowired
```

```
    private JdbcTemplate jdbcTemplate;
```

```
    public List<User> getUsers() {
```

```
        return jdbcTemplate.query("SELECT * FROM users",
```

```
            (rs, rowNum) -> new User(rs.getLong("id"), rs.getString("name")));
```

```
    }  
}
```

7.3 MongoDB / NoSQL

Spring Boot provides `spring-boot-starter-data-mongodb` for integrating MongoDB.

Example

@Document

```
public class Product {  
    @Id  
    private String id;  
    private String name;  
}
```

```
public interface ProductRepository extends MongoRepository<Product, String> {  
    List<Product> findByName(String name);  
}
```

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

7.4 Transactions (@Transactional)

Ensures atomicity for multiple database operations.

Example

@Service

```
public class UserService {  
  
    @Autowired  
    private UserRepository userRepository;  
  
    @Transactional  
    public void registerUser(User user) {  
        userRepository.save(user);  
  
        // Additional DB operations  
    }  
}
```

You can also use `@Transactional(readOnly = true)` for read operations.

8. Database Configuration in Spring Boot

Overview

Spring Boot offers seamless integration with relational databases such as **H2**, **MySQL**, **PostgreSQL**, and supports configuration for **connection pooling** and **schema initialization** through simple properties or SQL files.

8.1 H2 In-Memory Database

The **H2 database** is an in-memory relational database ideal for development and testing. It runs in memory and is destroyed when the application stops.

Add Dependency

```
<dependency>  
  
  <groupId>com.h2database</groupId>  
  
  <artifactId>h2</artifactId>  
  
  <scope>runtime</scope>  
  
</dependency>
```

Configure (application.properties)

```
spring.datasource.url=jdbc:h2:mem:testdb  
  
spring.datasource.driver-class-name=org.h2.Driver  
  
spring.datasource.username=sa  
  
spring.datasource.password=  
  
spring.h2.console.enabled=true
```

H2 Console

Access the console at: <http://localhost:8080/h2-console>
Driver Class: [org.h2.Driver](#)
JDBC URL: [jdbc:h2:mem:testdb](#)

8.2 MySQL / PostgreSQL Setup

MySQL Example Configuration

```
<!-- Add MySQL dependency -->
```

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-j</artifactId>

<scope>runtime</scope>

</dependency>

application.properties

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

spring.datasource.username=root

spring.datasource.password=your_password

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

PostgreSQL Configuration

<dependency>

<groupId>org.postgresql</groupId>

<artifactId>postgresql</artifactId>

<scope>runtime</scope>

</dependency>

spring.datasource.url=jdbc:postgresql://localhost:5432/mydb

spring.datasource.username=postgres

spring.datasource.password=your_password

8.3 Connection Pooling (HikariCP)

Spring Boot uses HikariCP as the default connection pool for better performance.

Default Settings (optional to override)

`spring.datasource.hikari.maximum-pool-size=10`

`spring.datasource.hikari.minimum-idle=5`

`spring.datasource.hikari.idle-timeout=30000`

`spring.datasource.hikari.pool-name=SpringBootHikariCP`

8.4 Schema Initialization (schema.sql, data.sql)

Spring Boot executes these SQL files automatically on startup to initialize the database.

Usage

Place these files in `src/main/resources/`:

- `schema.sql` → For creating tables
- `data.sql` → For inserting initial data

Example: schema.sql

```
CREATE TABLE users (  
    id BIGINT PRIMARY KEY AUTO_INCREMENT,  
    name VARCHAR(100)  
);
```

Example: data.sql

```
INSERT INTO users (name) VALUES ('John Doe'), ('Alice');
```

9. Validation in Spring Boot

Overview

Spring Boot supports **bean validation** using **JSR-380 (Jakarta Bean Validation / `javax.validation`)**. It allows automatic input validation for REST APIs and form data using annotations like `@NotNull`, `@Email`, and more. Validation can be enhanced using `@Valid`, `@Validated`, and custom validators.

9.1 Bean Validation (`javax.validation`)

Spring Boot auto-configures bean validation when `spring-boot-starter-validation` is on the classpath.

Add Dependency (if needed)

```
<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-validation</artifactId>

</dependency>
```

Common Validation Annotations

Annotation	Description
<code>@NotNull</code>	Field must not be null

`@NotBlank` Field must not be null or empty

`@Email` Must be a valid email address

`@Size(min, max)` String/collection size constraints

`@Min, @Max` Numeric value limits

Example

```
public class User {
```

```
    @NotBlank
```

```
    private String name;
```

```
    @Email
```

```
    private String email;
```

```
    @Min(18)
```

```
    private int age;
```

```
    // Getters & Setters
```

```
}
```


9.2 Using **@Valid** and **@Validated**

@Valid

Used to trigger validation on a method parameter (commonly in controllers).

@RestController

@RequestMapping("/users")

public class UserController {

@PostMapping

public ResponseEntity<String> addUser(@Valid @RequestBody User user) {

return ResponseEntity.ok("User is valid");

}

}

Spring will automatically return a 400 Bad Request if validation fails, along with error messages.

@Validated

Used on a class to validate method-level constraints or with groups for conditional validation.

@Validated

@Service

public class PaymentService {

public void process(@Min(100) int amount) {

// Only processes if amount >= 100

```
    }  
}
```

9.3 Custom Validators

You can create custom constraints using `@Constraint`.

Steps to Create a Custom Validator

1. Define the annotation:

```
@Constraint(validatedBy = UsernameValidator.class)  
  
@Target({ FIELD })  
  
@Retention(RUNTIME)  
  
public @interface ValidUsername {  
  
    String message() default "Invalid username";  
  
    Class<?>[] groups() default {};  
  
    Class<? extends Payload>[] payload() default {};  
  
}
```

2. Create the validator class:

```
public class UsernameValidator implements ConstraintValidator<ValidUsername, String>  
{  
  
    @Override  
  
    public boolean isValid(String value, ConstraintValidatorContext context) {  
  
        return value != null && value.matches("^[a-zA-Z0-9_]{5,20}$");  
  
    }  
  
}
```

```
    }  
}
```

3. Use the annotation:

```
public class User {  
    @ValidUsername  
    private String username;  
}
```

10. Security in Spring Boot

Overview

Spring Boot integrates **Spring Security**, a powerful and customizable authentication and access-control framework. It provides robust security for web applications, REST APIs, and microservices.

10.1 Spring Security Basics

To enable security in Spring Boot, include the **starter dependency**:

Dependency

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

By default, Spring Security:

- Secures all endpoints
- Provides a default login form
- Uses an auto-generated password (logged at startup)

10.2 Authentication and Authorization

Authentication identifies **who** the user is.

Authorization defines **what** the user is allowed to do.

Basic Configuration Example

@Configuration

@EnableWebSecurity

public class SecurityConfig {

 @Bean

 public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

 http

 .authorizeHttpRequests(auth -> auth

 .requestMatchers("/admin/**").hasRole("ADMIN")

 .requestMatchers("/user/**").hasAnyRole("USER", "ADMIN")

 .anyRequest().authenticated()

)

 .httpBasic(); // For simple authentication

 return http.build();

```

    }

    @Bean

    public InMemoryUserDetailsManager userDetailsService() {

        UserDetails user =
        User.withUsername("user").password("{noop}1234").roles("USER").build();

        UserDetails admin =
        User.withUsername("admin").password("{noop}admin123").roles("ADMIN").build();

        return new InMemoryUserDetailsManager(user, admin);

    }
}

```

10.3 Password Encoding

Use **PasswordEncoder** to store hashed passwords securely.

Example using BCrypt

```

@Bean

public PasswordEncoder passwordEncoder() {

    return new BCryptPasswordEncoder();

}

```

```
String encoded = passwordEncoder.encode("mypassword");
```

Always use encoded passwords (e.g., in DB) instead of plain text.

10.4 JWT Token-based Security

JWT (JSON Web Tokens) are widely used for securing REST APIs in stateless systems.

JWT Flow

1. User sends credentials → gets a **JWT**
2. Client sends JWT with requests
3. Server validates the JWT

Example Controller to Issue JWT

@RestController

```
public class AuthController {  
  
    @PostMapping("/login")  
    public String login(@RequestBody AuthRequest request) {  
        // Authenticate and generate JWT  
        return jwtService.generateToken(request.getUsername());  
    }  
}
```

JWT Filter (Simplified)

```
public class JwtFilter extends OncePerRequestFilter {  
  
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse  
response, FilterChain chain) {  
  
        String token = request.getHeader("Authorization");  
  
        // validate token and set authentication
```

```
        chain.doFilter(request, response);
    }
}
```

You must register the filter in the security chain.

10.5 Role-Based Access Control (RBAC)

Spring Security allows controlling access to resources by roles or authorities.

Example

```
@GetMapping("/admin/dashboard")
@PreAuthorize("hasRole('ADMIN')")
public String adminDashboard() {
    return "Admin area";
}
```

Enable Method-Level Security

```
@EnableMethodSecurity
public class MethodSecurityConfig {
}
```

11. Testing in Spring Boot

Overview

Spring Boot provides first-class support for testing through **JUnit**, **Mockito**, and **Spring TestContext Framework**. It enables testing of individual components (unit tests), interactions (mocking), and full application context (integration testing).

11.1 Unit Testing with JUnit

JUnit is the standard framework for writing unit tests in Java.

Dependency

```
<dependency>

  <groupId>org.springframework.boot</groupId>

  <artifactId>spring-boot-starter-test</artifactId>

  <scope>test</scope>

</dependency>
```

Example

```
class CalculatorService {

    int add(int a, int b) {

        return a + b;

    }

}
```

```
class CalculatorServiceTest {

    @Test

    void testAdd() {

        CalculatorService calc = new CalculatorService();
```



```
        assertEquals(5, calc.add(2, 3));
    }
}
```

11.2 Mocking with Mockito

Mockito is used to **mock dependencies** for isolated unit tests.

Example

```
@Service
class OrderService {

    @Autowired private PaymentService paymentService;

    public boolean placeOrder() {
        return paymentService.pay();
    }
}
```

```
class OrderServiceTest {

    @Mock PaymentService paymentService;

    @InjectMocks OrderService orderService;

    @BeforeEach
```

```
void init() {  
  
    MockitoAnnotations.openMocks(this);  
  
}  
  
@Test  
  
void testPlaceOrder() {  
  
    when(paymentService.pay()).thenReturn(true);  
  
    assertTrue(orderService.placeOrder());  
  
}  
}
```

11.3 Integration Testing

Integration tests load the full application context to test real behavior of beans.

Example

```
@SpringBootTest  
  
class UserServiceIntegrationTest {  
  
    @Autowired UserService userService;  
  
    @Test  
  
    void testFindUser() {  
  
        User user = userService.findById(1L);  
  
        assertNotNull(user);  
    }  
}
```

```
}  
}
```

11.4 Web Layer Testing (@WebMvcTest)

Used to test only the **web layer** (controllers) without loading the full context.

Example

```
@WebMvcTest(UserController.class)
```

```
class UserControllerTest {
```

```
    @Autowired private MockMvc mockMvc;
```

```
    @Test
```

```
    void testGetUser() throws Exception {
```

```
        mockMvc.perform(get("/users/1"))
```

```
            .andExpect(status().isOk());
```

```
    }
```

```
}
```

It auto-configures Spring MVC infrastructure and can mock service beans using `@MockBean`.

11.5 Data Layer Testing (@DataJpaTest)

Used to test **repository layer** with in-memory database and only JPA components.

Example

```
@DataJpaTest
```

```
class UserRepositoryTest {
```

```
    @Autowired private UserRepository userRepository;
```

```
    @Test
```

```
    void testSaveUser() {
```

```
        User user = new User("John");
```

```
        User saved = userRepository.save(user);
```

```
        assertNotNull(saved.getId());
```

```
    }
```

```
}
```

Uses H2 in-memory database by default and rolls back transactions after each test.

12. Logging in Spring Boot

Overview

Spring Boot uses **SLF4J** (Simple Logging Facade for Java) as a logging API and **Logback** as the default logging implementation. Logging helps monitor application behavior, track errors, and audit key events.

12.1 SLF4J and Logback

SLF4J

- A facade that allows switching between logging frameworks like Logback, Log4j, etc.
- Spring Boot includes SLF4J and Logback by default.

Basic Usage

```
import org.slf4j.Logger;
```

```
import org.slf4j.LoggerFactory;
```

```
@RestController
```

```
public class DemoController {
```

```
    private static final Logger logger = LoggerFactory.getLogger(DemoController.class);
```

```
    @GetMapping("/test")
```

```
    public String test() {
```

```
        logger.info("INFO log - test endpoint called");
```

```
        logger.warn("WARN log");
```

```
        logger.error("ERROR log");
```

```
        return "Check logs";
```

```
    }
```

```
}
```

12.2 Configuring Log Levels

Spring Boot allows configuring log levels in `application.properties` or `application.yml`.

Example: `application.properties`

```
logging.level.root=INFO
```

```
logging.level.com.example.demo=DEBUG
```

```
logging.file.name=app.log
```

Log Levels

- `TRACE` – Finest level (e.g., for debugging)
 - `DEBUG` – Development details
 - `INFO` – General application events
 - `WARN` – Potential issues
 - `ERROR` – Serious issue
-

12.3 External Log Configuration

You can use an external `logback-spring.xml` for advanced logging configurations.

Steps:

1. Place `logback-spring.xml` in `src/main/resources`
2. Use Spring profile-specific logging, log rotation, custom patterns, etc.

Example: `logback-spring.xml`

```
<configuration>

<property name="LOG_FILE" value="myapp.log"/>

<appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>${LOG_FILE}</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
        <fileNamePattern>${LOG_FILE}.%d{yyyy-MM-dd}.gz</fileNamePattern>
    </rollingPolicy>
    <encoder>
        <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>
    </encoder>
</appender>

<root level="INFO">
    <appender-ref ref="FILE"/>
</root>

</configuration>
```

You can also use environment variables and Spring profiles for flexible logging configurations.

13. Actuator & Monitoring in Spring Boot

Overview

Spring Boot Actuator provides built-in **monitoring and management endpoints** to inspect application health, metrics, and environment details. It is essential for observing applications in production environments.

13.1 Spring Boot Actuator Overview

Dependency

```
<dependency>

  <groupId>org.springframework.boot</groupId>

  <artifactId>spring-boot-starter-actuator</artifactId>

</dependency>
```

Enable Actuator Endpoints

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

By default, only limited endpoints like **/health** and **/info** are exposed.

13.2 Common Actuator Endpoints

Endpoint	Description
/actuator/health	Shows application health status

<code>/actuator/info</code>	Displays custom application info
<code>/actuator/metrics</code>	Shows system and custom metrics
<code>/actuator/env</code>	Lists environment properties
<code>/actuator/beans</code>	Lists all Spring beans
<code>/actuator/mappings</code>	Shows all request mappings

Example: Custom Info

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

`info.app.name=MyApp`

`info.app.version=1.0`

13.3 Custom Metrics

You can create and register custom metrics using `MeterRegistry`.

Example: Custom Counter

`@Component`

`public class VisitCounter {`

```
private final Counter counter;

public VisitCounter(MeterRegistry registry) {
    this.counter = registry.counter("custom.visit.counter");
}

public void increment() {
    counter.increment();
}
}
```

Access at: </actuator/metrics/custom.visit.counter>

13.4 Prometheus/Grafana Integration

Actuator integrates with Micrometer, which supports exporting metrics to Prometheus, Datadog, New Relic, etc.

Prometheus Setup

Add dependency:

```
<dependency>

    <groupId>io.micrometer</groupId>

    <artifactId>micrometer-registry-prometheus</artifactId>

</dependency>
```

`application.properties:`

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

`management.metrics.export.prometheus.enabled=true`

Access Prometheus endpoint:

`/actuator/prometheus`

Grafana Integration

1. Add Prometheus as a data source in Grafana.
 2. Use `actuator` metrics in custom dashboards.
 3. Monitor CPU, memory, GC, and custom metrics visually.
-

14. Error Handling in Spring Boot

Overview

Spring Boot provides a default error handling mechanism and allows full customization via exception handlers, error pages, and controller advice. Proper error handling ensures a better user experience and easier debugging.

14.1 Default Error Handling

By default, Spring Boot handles exceptions and returns a **JSON error response** for REST APIs or a **whitelabel error page** for web applications.

Example: Default Error Response

```
{  
  "timestamp": "2025-06-14T12:30:00.000+00:00",  
  "status": 404,  
  "error": "Not Found",  
  "path": "/api/users/100"  
}
```

14.2 Custom Error Pages

You can provide custom HTML error pages by adding files in `/resources/public` or `/resources/static` directories with the following names:

Examples:

- `/resources/public/error/404.html` → Custom page for 404 errors
- `/resources/public/error/500.html` → Custom page for 500 errors

These are served automatically for browser-based requests.

14.3 Global Exception Handling

Use `@ControllerAdvice` and `@ExceptionHandler` to define centralized exception handling for all controllers.

Example:

`@ControllerAdvice`

```
public class GlobalExceptionHandler {
```

```

    @ExceptionHandler(ResourceNotFoundException.class)

    public ResponseEntity<String> handleNotFound(ResourceNotFoundException ex) {

        return new ResponseEntity<>("Resource not found: " + ex.getMessage(),
        HttpStatus.NOT_FOUND);

    }

    @ExceptionHandler(Exception.class)

    public ResponseEntity<String> handleGeneral(Exception ex) {

        return new ResponseEntity<>("Internal error: " + ex.getMessage(),
        HttpStatus.INTERNAL_SERVER_ERROR);

    }

}

```

This ensures consistent and customizable error responses across the application.

Best Practices

- Use **@ResponseStatus** to bind HTTP status codes to exceptions.
- Log exceptions for debugging.
- Return **meaningful messages** to users, hide sensitive stack traces in production.
- Customize **error attributes** using **ErrorAttributes** bean if needed.

15. Developer Tools in Spring Boot

Overview

Spring Boot Developer Tools (**DevTools**) enhance the development experience by enabling **automatic restarts**, **live reload**, and additional features like **runtime property override** and **template caching disablement**.

15.1 Spring Boot DevTools

`spring-boot-devtools` is a module that provides development-time features to increase productivity.

Dependency

Add this dependency to your `pom.xml` (it's automatically disabled in production):

```
<dependency>

  <groupId>org.springframework.boot</groupId>

  <artifactId>spring-boot-devtools</artifactId>

  <optional>true</optional>

</dependency>
```

In Gradle:

```
developmentOnly("org.springframework.boot:spring-boot-devtools")
```

15.2 Auto Restart

Spring Boot DevTools monitors classpath files. When a change is detected (like a Java class file update), the application is automatically restarted.

Benefits:

- Eliminates the need to manually stop and start the app.

- Improves feedback loop during development.

Example

- Modify a controller method in your IDE.
- Save → Spring Boot automatically restarts the context.
- Changes are reflected immediately.

You can exclude directories from being monitored using:

```
spring.devtools.restart.exclude=static/**,public/**
```

15.3 Live Reload

Spring Boot DevTools integrates with **LiveReload** to refresh the browser automatically when static resources (HTML, CSS, JS) change.

Steps to Enable:

1. Add DevTools dependency.
2. Install LiveReload browser extension.
3. Make a change in an HTML or template file.
4. The browser reloads automatically.

Configuration (optional)

```
spring.devtools.livereload.enabled=true
```

Other Features

- **Disable Caching** for `Thymeleaf`, `Freemarker`, etc.
 - **Remote Debugging Support** for deploying DevTools to remote environments (optional and secured).
 - **Automatic Property Overrides** for dev-time config.
-

16. Caching in Spring Boot

Overview

Caching improves performance by storing frequently accessed data in memory or external caches. Spring Boot supports transparent caching using annotations and multiple providers like Ehcache, Caffeine, and Redis.

16.1 Using `@Cacheable` and `@CacheEvict`

Spring provides simple annotations to manage caching behavior.

`@EnableCaching`

Enable caching in your main class:

```
@SpringBootApplication
```

```
@EnableCaching
```

```
public class MyApp { ... }
```

`@Cacheable`

Marks a method's return value to be cached. The result is stored the first time and reused for subsequent calls with the same parameters.

```
@Cacheable("products")
```

```
public Product getProductById(Long id) {  
    simulateSlowService(); // e.g., DB call  
    return productRepository.findById(id).orElse(null);  
}
```

products is the cache name.

On first call, the method executes and result is cached.

On subsequent calls with same **id**, value is fetched from the cache.

✓ @CacheEvict

Removes an entry from the cache.

```
@CacheEvict(value = "products", key = "#id")
```

```
public void deleteProduct(Long id) {  
    productRepository.deleteById(id);  
}
```

- Ensures cache is cleared when the data changes.

16.2 Cache Providers

Spring Boot supports pluggable caching providers. Some popular ones:

Ehcache (In-memory, Java-based)

➤ Add dependency:

```
<dependency>  
  <groupId>org.ehcache</groupId>  
  <artifactId>ehcache</artifactId>  
</dependency>
```

➤ Configuration example (**ehcache.xml** or Java config)

```
<config>  
  <cache alias="products">  
    <heap>1000</heap>  
  </cache>  
</config>
```

Caffeine (Fast Java-based cache)

➤ Add dependency:

```
<dependency>  
  <groupId>com.github.ben-manes.caffeine</groupId>  
  <artifactId>caffeine</artifactId>  
</dependency>
```

➤ Configuration (**application.properties**):

spring.cache.type=caffeine

spring.cache.caffeine.spec=maximumSize=1000,expireAfterAccess=5m

Redis (Distributed cache)

➤ Add dependencies:

```
<dependency>
```

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

```
    <artifactId>spring-boot-starter-data-redis</artifactId>
```

```
</dependency>
```

➤ Configuration:

```
spring.cache.type=redis
```

```
spring.redis.host=localhost
```

```
spring.redis.port=6379
```

Ideal for microservices and distributed environments.

17. Scheduling & Asynchronous Execution in Spring Boot

Overview

Spring Boot simplifies the execution of background and periodic tasks through annotations like `@Scheduled` and `@Async`. These features enable automation, performance optimization, and non-blocking operations.

17.1 Scheduling Tasks (@Scheduled)

`@Scheduled` allows you to run methods at fixed intervals or specific times.

✓ Enable Scheduling

```
@SpringBootApplication
```

```
@EnableScheduling
```

```
public class MyApp { }
```

✓ Using @Scheduled

```
@Component
```

```
public class ScheduledTasks {
```

```
    @Scheduled(fixedRate = 5000) // every 5 seconds
```

```
    public void reportStatus() {
```

```
        System.out.println("Status Check: " + LocalDateTime.now());
```

```
    }
```

```
    @Scheduled(cron = "0 0 9 * * ?") // every day at 9 AM
```

```
    public void dailyTask() {
```

```
        System.out.println("Running daily task");
```

```
    }
```

```
}
```

fixedRate: Runs task at regular intervals.

fixedDelay: Runs with a delay after the previous execution.

cron: Offers flexible scheduling using cron expressions.

17.2 Asynchronous Execution (@Async)

@Async enables methods to run in a separate thread, allowing non-blocking execution.

✓ Enable Async Execution

```
@SpringBootApplication
```

```
@EnableAsync
```

```
public class MyApp { }
```

✓ Using @Async

```
@Service
```

```
public class EmailService {
```

```
    @Async
```

```
    public void sendEmail(String to) {
```

```
        // Simulate delay
```

```
        try {
```

```
            Thread.sleep(3000);
```

```
        } catch (InterruptedException ignored) {}

        System.out.println("Email sent to: " + to);

    }

}
```

✓ Calling the Async Method

@RestController

```
public class NotificationController {

    @Autowired

    private EmailService emailService;

    @GetMapping("/notify")

    public String notifyUser() {

        emailService.sendEmail("user@example.com");

        return "Notification started";

    }

}
```

- The method returns immediately.
- Actual email sending happens in a background thread.

⚠ Async methods must be called from another bean, not the same class.

18. Messaging in Spring Boot

Overview

Spring Boot offers powerful support for **message-driven architecture** using brokers like **RabbitMQ**, **Apache Kafka**, and **JMS (Java Message Service)**. Messaging enables **asynchronous communication**, **event-driven processing**, and **microservice decoupling**.

18.1 RabbitMQ

RabbitMQ is a message broker that supports **AMQP** for reliable, asynchronous message delivery.

✓ Add Dependency

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

✓ Configuration (`application.properties`)

```
spring.rabbitmq.host=localhost  
  
spring.rabbitmq.port=5672  
  
spring.rabbitmq.username=guest  
  
spring.rabbitmq.password=guest
```

✓ Producer Example

@Service

```
public class MessageProducer {  
  
    @Autowired  
    private RabbitTemplate rabbitTemplate;  
  
    public void send(String message) {  
        rabbitTemplate.convertAndSend("myQueue", message);  
    }  
}
```

✓ Consumer Example

@Component

```
public class MessageConsumer {  
  
    @RabbitListener(queues = "myQueue")  
    public void receive(String message) {  
        System.out.println("Received: " + message);  
    }  
}
```

18.2 Apache Kafka

Kafka is a distributed event streaming platform used for high-throughput and fault-tolerant messaging.

✓ Add Dependency

<dependency>


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

✓ Configuration (`application.properties`)

```
spring.kafka.bootstrap-servers=localhost:9092  
spring.kafka.consumer.group-id=my-group  
spring.kafka.consumer.auto-offset-reset=earliest
```

✓ Producer Example

`@Service`

```
public class KafkaProducer {  
    @Autowired  
    private KafkaTemplate<String, String> kafkaTemplate;  
  
    public void send(String message) {  
        kafkaTemplate.send("myTopic", message);  
    }  
}
```

✓ Consumer Example

`@Component`

```
public class KafkaConsumer {  
    @KafkaListener(topics = "myTopic", groupId = "my-group")
```

```
    public void listen(String message) {  
        System.out.println("Received from Kafka: " + message);  
    }  
}
```

18.3 JMS Integration

JMS (Java Message Service) is a Java API for messaging, often used with **ActiveMQ**.

✓ Add Dependency

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

✓ Configuration

```
spring.artemis.broker-url=tcp://localhost:61616  
  
spring.artemis.user=admin  
  
spring.artemis.password=admin
```

✓ Producer Example

```
@Service  
  
public class JmsProducer {  
    @Autowired  
    private JmsTemplate jmsTemplate;
```

```
        public void send(String msg) {  
            jmsTemplate.convertAndSend("queue.sample", msg);  
        }  
    }  
}
```

✓ Consumer Example

@Component

```
public class JmsConsumer {  
    @JmsListener(destination = "queue.sample")  
    public void receive(String msg) {  
        System.out.println("Received from JMS: " + msg);  
    }  
}
```

19. File Upload & Download in Spring Boot

Overview

Spring Boot provides built-in support for handling **file uploads** using `MultipartFile`, and allows **secure file storage and download** through REST APIs. These features are essential for applications like user portals, document systems, and image uploads.

19.1 Multipart File Upload

✓ Controller for File Upload

```

@RestController

@RequestMapping("/files")

public class FileUploadController {

    @PostMapping("/upload")

    public ResponseEntity<String> handleFileUpload(@RequestParam("file") MultipartFile
file) throws IOException {

        String uploadDir = "uploads/";

        Path path = Paths.get(uploadDir + file.getOriginalFilename());

        Files.createDirectories(path.getParent());

        Files.write(path, file.getBytes());

        return ResponseEntity.ok("File uploaded successfully: " + file.getOriginalFilename());

    }

}

```

✓ Frontend / Postman Test

- Set **POST** method to **/files/upload**
- Use **form-data** with key **file** and select a file

19.2 File Storage and Download Handling

✓ Download File Endpoint

```

@GetMapping("/download/{filename}")

```

```

public ResponseEntity<Resource> downloadFile(@PathVariable String filename) throws
IOException {

    Path path = Paths.get("uploads/" + filename);

    Resource resource = new UrlResource(path.toUri());

    if (!resource.exists()) {

        return ResponseEntity.notFound().build();

    }

    return ResponseEntity.ok()

        .contentType(MediaType.APPLICATION_OCTET_STREAM)

        .header(HttpHeaders.CONTENT_DISPOSITION, "attachment; filename=\"" + filename +

"\")

        .body(resource);

}

```

- Files are stored in a local **uploads/** directory.
- Files are downloaded with correct headers for browser compatibility.

Security Tip

Always validate:

- File type (e.g., **.jpg**, **.pdf**)
- File size limits
- Filename/path sanitization to avoid directory traversal

`spring.servlet.multipart.max-file-size=5MB`

`spring.servlet.multipart.max-request-size=10MB`

20. Frontend Integration in Spring Boot

Overview

Spring Boot simplifies frontend integration by allowing you to serve static content, use server-side templating engines like **Thymeleaf**, and enable **CORS** for cross-origin requests from frontend frameworks like Angular, React, or Vue.

20.1 Serving Static Files

Spring Boot automatically serves static content (HTML, CSS, JS, images) placed in specific directories:

✓ Default Static Locations

Place your files in:

`src/main/resources/static/`

`src/main/resources/public/`

`src/main/resources/resources/`

✓ Example

`src/main/resources/static/index.html`

Accessible via: `http://localhost:8080/index.html`

Add JS/CSS/images in the same folder:

`/static/js/app.js`

/static/css/style.css

20.2 Using Thymeleaf

Thymeleaf is a server-side Java template engine for rendering dynamic HTML.

✓ Add Dependency

```
<dependency>

  <groupId>org.springframework.boot</groupId>

  <artifactId>spring-boot-starter-thymeleaf</artifactId>

</dependency>
```

✓ Controller

@Controller

```
public class HomeController {

    @GetMapping("/home")

    public String home(Model model) {

        model.addAttribute("message", "Welcome to Spring Boot + Thymeleaf!");

        return "home";

    }

}
```

✓ Template: `src/main/resources/templates/home.html`

```
<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">
```

```
<head><title>Home</title></head>

<body>

    <h1 th:text="${message}"></h1>

</body>

</html>
```

Access via: <http://localhost:8080/home>

20.3 Enabling CORS

CORS (Cross-Origin Resource Sharing) allows frontend apps (like React or Angular) to call Spring Boot APIs hosted on another domain or port.

Enable CORS Globally

@Configuration

```
public class WebConfig implements WebMvcConfigurer {

    @Override

    public void addCorsMappings(CorsRegistry registry) {

        registry.addMapping("/api/**")

            .allowedOrigins("http://localhost:4200")

            .allowedMethods("GET", "POST", "PUT", "DELETE");

    }

}
```


✅ Enable CORS at Controller Level

```
@CrossOrigin(origins = "http://localhost:4200")

@RestController

@RequestMapping("/api")

public class ProductController {

    @GetMapping("/products")

    public List<String> getProducts() {

        return List.of("Pen", "Notebook", "Pencil");

    }

}
```

21. Build & Deployment in Spring Boot

Overview

Spring Boot simplifies application deployment with flexible options such as **executable JARs/WARs**, **Docker containers**, **cloud deployment**, and **CI/CD integration** for automated delivery.

21.1 Building JAR/WAR

✅ JAR Packaging (Default)

Add to `pom.xml`:

```
<packaging>jar</packaging>
```

Build the JAR:

mvn clean package

Run the JAR:

```
java -jar target/myapp-0.0.1-SNAPSHOT.jar
```

✓ WAR Packaging (Optional)

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

Extend `SpringBootServletInitializer`

```
@SpringBootApplication
```

```
public class MyApp extends SpringBootServletInitializer {  
  
    @Override  
  
    protected SpringApplicationBuilder  
configure(SpringApplicationBuilder builder) {  
  
        return builder.sources(MyApp.class);  
  
    }  
  
}
```

Deploy to external servers (Tomcat, JBoss, etc.)

21.2 Dockerizing Spring Boot Application

✓ Create `Dockerfile`

```
FROM openjdk:21
```

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

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

✓ Build Docker Image

```
docker build -t myapp .
```

✓ Run Container

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

21.3 Deploying to Cloud

Spring Boot can be deployed on AWS, Azure, GCP, and others.

✓ AWS (Elastic Beanstalk)

- Package as JAR/WAR
- Use AWS CLI or Management Console
- Example:

```
eb init
```

```
eb create my-spring-env
```

```
eb deploy
```

✓ Azure App Service

- Create app using Azure CLI or portal
- Deploy using:

```
az webapp up --name myapp --resource-group mygroup
```

✓ GCP App Engine

- Create `app.yaml`:

```
runtime: java
```

```
env: standard
```

Deploy:

```
gcloud app deploy
```

21.4 CI/CD Integration

Automate builds, tests, and deployments using CI/CD tools.

✓ GitHub Actions (Sample Workflow)

```
name: Build Spring Boot App
```

```
on: [push]
```

```
jobs:
```

```
  build:
```

```
    runs-on: ubuntu-latest
```

```
    steps:
```

- uses: actions/checkout@v3

- name: Set up JDK

uses: actions/setup-java@v3

with:

java-version: '21'

- name: Build

run: mvn clean package

✓ Other CI/CD Options

- Jenkins – Create pipelines to build and deploy automatically.
- GitLab CI – `.gitlab-ci.yml` for stages.
- CircleCI, Bitbucket Pipelines, Azure DevOps – Full cloud-based CI/CD support.

22. Advanced Topics in Spring Boot

Overview

Advanced Spring Boot features enable the development of **resilient**, **scalable**, and **modular** applications using **microservices**, **event-driven architecture**, and **custom starters**.

22.1 Microservices with Spring Boot

Spring Boot combined with **Spring Cloud** offers comprehensive tools for building distributed systems.

22.1.1 Spring Cloud

A set of tools to support microservice patterns: configuration, discovery, routing, load balancing, and resilience.

```
<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>2023.0.0</version>

<type>pom</type>

<scope>import</scope>

</dependency>
```

22.1.2 Eureka (Service Discovery)

- **Eureka Server:**

```
@EnableEurekaServer

@SpringBootApplication

public class EurekaServerApp { ... }
```

Eureka Client:

```
@EnableEurekaClient

@SpringBootApplication

public class ProductServiceApp { ... }
```

Add dependency:

```
<dependency>
```

```
<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>

</dependency>
```

22.1.3 Spring Cloud Gateway / Zuul

- **Spring Cloud Gateway** is the modern replacement for Zuul.

```
@SpringBootApplication
```

```
public class GatewayApp { ... }
```

Config in **application.yml**:

```
spring:
  cloud:
    gateway:
      routes:
        - id: product_route
          uri: http://localhost:8081
          predicates:
            - Path=/products/**
```

22.1.4 Circuit Breaker (Resilience4j, Hystrix)

- Use **Resilience4j** for fault tolerance.

```
<dependency>
  <groupId>io.github.resilience4j</groupId>
  <artifactId>resilience4j-spring-boot3</artifactId>
</dependency>
```

```
@CircuitBreaker(name = "productService", fallbackMethod = "fallback")
public String getProduct() {
    return restTemplate.getForObject("http://product-service/api", String.class);
}
```

```
public String fallback(Throwable t) {
    return "Fallback response";
}
```

```
}
```

22.1.5 Config Server

- Centralized external configuration management for microservices.

Config Server:

```
@EnableConfigServer
@SpringBootApplication
public class ConfigServerApp { ... }
```

application.yml:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/my-org/config-repo
```

Client:

```
spring:
  config:
    import: optional:configserver:http://localhost:8888
```

22.2 ApplicationEventPublisher & Event Listeners

✓ Define Custom Event

```
public class UserCreatedEvent extends ApplicationEvent {
    private String email;
    public UserCreatedEvent(Object source, String email) {
        super(source);
        this.email = email;
    }
}
```

✓ Publish Event


```

@Autowired
private ApplicationEventPublisher publisher;

public void createUser(String email) {
    // Save user logic
    publisher.publishEvent(new UserCreatedEvent(this, email));
}

```

✓ Listen to Event

```

@Component
public class UserEventListener {
    @EventListener
    public void handleUserCreated(UserCreatedEvent event) {
        System.out.println("User created: " + event.getEmail());
    }
}

```

22.3 Creating Custom Starters

Creating your own Spring Boot starter allows packaging reusable logic.

✓ Steps:

1. Create a separate module (e.g., **my-spring-boot-starter**)
2. Include **spring-boot-autoconfigure**
3. Add **META-INF/spring.factories** (or **spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports** for Spring Boot 3)

```

org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.example.autoconfig.MyAutoConfiguration

```

4. Example Auto-Configuration:

```

@Configuration
public class MyAutoConfiguration {
    @Bean
    public MyService myService() {
        return new MyService();
    }
}

```

```
}  
    }  
}
```

Then include your custom starter as a dependency in another project.
