1.1 What is Spring Boot?

Spring Boot is a framework that simplifies the development of Spring-based applications. It removes boilerplate code and provides built-in functionalities like:

- ✅ Auto-configuration (No need for manual XML configuration)
- ✅ Embedded servers (Tomcat, Jetty, Undertow)
- ✅ Production-ready features (Monitoring, Logging, Security)
- ✅ Microservices support

1.2 Features of Spring Boot

- 🔹 Standalone – No need for an external web server (Tomcat is built-in).
- 🔹 Opinionated Defaults – Provides pre-configured settings for faster development.
- 🔹 Spring Boot Starters – Predefined dependencies for common use cases.
- 🔹 Spring Boot CLI – Command-line tool to run Spring applications.
- 🔹 Spring Boot Actuator – Monitoring & management tools.

1.3 Difference Between Spring and Spring Boot

Feature	Spring	Spring Boot
Configuration	Manual XML or Java Config	Auto-configuration
Server	Requires external Tomcat/Jetty	Embedded Tomcat, Jetty
Dependencies	Manually added	Uses Spring Boot Starters
Setup Time	Takes time to configure	Quick setup with Spring Initializr
Microservices	Needs additional setup	Built-in support

```
my-spring-boot-app
├── src
│   ├── main
│   │   ├── java/com/example/demo
│   │   │   ├── DemoApplication.java  <-- Main class
│   │   │   ├── controller
│   │   │   ├── service
│   │   │   ├── repository
│   │   ├── resources
│   │   │   ├── application.properties  <-- Configuration file
│   ├── test  <-- Unit tests
├── pom.xml  <-- Maven dependencies
```

This is the entry point of your Spring Boot app.

```
package com.example.demo;

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

@SpringBootApplication  // Enables auto-configuration and component
scanning
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

REST Controller

```
package com.example.demo.controller;

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

@RestController  // Marks this class as a REST API controller
@RequestMapping("/api")
public class HelloController {

    @GetMapping("/hello")  // Handles HTTP GET requests at /api/hello
    public String sayHello() {
        return "Hello, Spring Boot!";
    }
}
```

Run the Application

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

Open browser and visit: http://localhost:8080/api/hello ✅ You should see: "Hello, Spring Boot!" 🎉

application.properties (Configuration File)

Spring Boot uses application.properties to configure settings.

```
# Application Settings
server.port=9090  # Default port is 8080
```

Now your app will run on http://localhost:9090 instead of 8080.

Chapter 2

Spring Boot automatically configures beans and components based on the dependencies present in the classpath.

Without Auto-Configuration (Traditional Spring)

Before Spring Boot, you had to manually configure the beans in XML or Java:

```
<bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource">
    <property name="driverClassName" value="com.mysql.cj.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/mydb"/>
    <property name="username" value="root"/>
    <property name="password" value="password"/>
</bean>
```

With Auto-Configuration (Spring Boot)

Spring Boot removes this manual work:

```
# Database Configuration
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=password
```

✅ No XML, No Extra Java Code! Spring Boot auto-configures the DataSource when it detects MySQL driver in pom.xml.

How Does Auto-Configuration Work?

- 1️⃣ Spring Boot scans the classpath for dependencies.
- 2️⃣ If a required JAR file is present, Spring Boot automatically configures the related beans.
- 3️⃣ The @EnableAutoConfiguration annotation loads default configurations from spring-boot-autoconfigure package.

Understanding @SpringBootApplication

When you create a Spring Boot project, you see this annotation on the main class:

```
@SpringBootApplication
public class MyApplication {
```

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

This annotation is equivalent to 3 separate annotations:

```
@Configuration  // Marks this class for Spring configuration
@EnableAutoConfiguration  // Enables auto-configuration
@ComponentScan  // Scans components in the package
```

✅ Reduces boilerplate code and simplifies setup!

Spring Boot Starter Dependencies

Spring Boot provides predefined dependencies for common use cases called Starters.

Adding Spring Boot Starters in pom.xml

```
<!-- Spring Boot Web Starter (For REST APIs) -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Starter	Purpose
spring-boot-starter-web	Build REST APIs (Spring MVC, Embedded Tomcat)
spring-boot-starter-data-jpa	Database access with JPA & Hibernate
spring-boot-starter-security	Security & Authentication
spring-boot-starter-thymeleaf	Spring MVC + Thymeleaf Templating
spring-boot-starter-test	Unit Testing with JUnit & Mockito

✅ Less dependency management, faster development!

Customizing Auto-Configuration

Sometimes, you might want to override the default auto-configuration.

Customizing DataSource Bean

Instead of using application.properties, you can manually define a DataSource Bean:

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import javax.sql.DataSource;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

@Configuration
public class DataSourceConfig {

    @Bean
    public DataSource customDataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/mydb");
        dataSource.setUsername("root");
        dataSource.setPassword("password");
        return dataSource;
    }
}
```

✅ Overrides the default DataSource configuration!

Disabling Auto-Configuration

If you don't want Spring Boot to auto-configure certain components, you can exclude them:

Excluding DataSource Auto-Configuration

```java
@SpringBootApplication(exclude = {DataSourceAutoConfiguration.class})
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

✅ Useful when you don't need a database but Spring Boot tries to configure it!

✅ Summary ✔ Auto-Configuration automatically configures Spring beans based on dependencies. ✔ @SpringBootApplication is a combination of @Configuration, @EnableAutoConfiguration, and @ComponentScan. ✔ Spring Boot Starters simplify dependency management. ✔ You can customize or disable auto-configuration when needed.

Spring Boot Dependency Injection (DI) & IoC 🔄

- ✅ What is Dependency Injection (DI)?
- ✅ How does IoC (Inversion of Control) work?

- ✅ Types of Dependency Injection (Constructor, Setter, Field Injection)
- ✅ Using @Autowired Annotation
- ✅ Bean Scope & Lifecycle
- ✅ Creating Custom Beans using @Bean

What is Dependency Injection? 🤔

Dependency Injection (DI) is a design pattern where dependencies (objects) are injected into a class instead of being created manually inside the class.

✅ This makes the code loosely coupled, more testable, and maintainable.

- Without Dependency Injection (Tightly Coupled)

```
class Car {
    private Engine engine;

    public Car() {
        this.engine = new Engine();  // Manual object creation
    }
}
```

🚨 Problem:

Hardcoded dependencies → Difficult to change/test Engine. Tightly Coupled → If we want a different engine, we must modify Car class.

- With Dependency Injection (Loosely Coupled)

```
class Car {
  private Engine engine;

  // Dependency Injection via Constructor
  public Car(Engine engine) {
      this.engine = engine;
  }
}
```

✅ Now Car doesn't depend on how Engine is created!

IoC (Inversion of Control) in Spring

In traditional Java, we manually create objects using new(). In Spring, the IoC (Inversion of Control) Container manages objects (Beans) for us.

How IoC Works in Spring?

- 1️⃣ We define a Bean (a Java object managed by Spring).
- 2️⃣ Spring Container creates and injects dependencies automatically.
- 3️⃣ We don't need new() manually.

IoC in Spring Boot

```
@Component
class Engine {
    public void start() {
        System.out.println("Engine started...");
    }
}

@Component
class Car {
    private Engine engine;

    @Autowired  // Injecting Engine automatically
    public Car(Engine engine) {
        this.engine = engine;
    }

    public void drive() {
        engine.start();
        System.out.println("Car is moving...");
    }
}
```

1.Constructor Injection (Best Practice)

Most recommended → Ensures mandatory dependencies at object creation.

```
@Component
class Engine {
   public void start() {
       System.out.println("Engine started...");
   }
}

@Component
class Car {
   private final Engine engine;

   @Autowired
   public Car(Engine engine) {  // Injecting Engine via Constructor
       this.engine = engine;
   }

   public void drive() {
```

```
        engine.start();
        System.out.println("Car is moving...");
    }
}
```

✅ Advantages of Constructor Injection

- ✔ Ensures mandatory dependencies at object creation.
- ✔ Works well with immutable objects (final fields).
- ✔ Better for Unit Testing.

2.Setter Injection

🔹 Used when dependency is optional.

```
@Component
class Car {
    private Engine engine;

    @Autowired
    public void setEngine(Engine engine) {  // Injecting Engine via Setter
        this.engine = engine;
    }
}
```

- ✅ Good when dependencies need to change after object creation.
- 🚨 But: Allows object creation without the dependency (not always ideal).

3.Field Injection (Avoid ⚠)

🔹 Uses @Autowired directly on a field.

```
@Component
class Car {
   @Autowired
   private Engine engine;
}
```

🚨 Why Avoid Field Injection?

- Harder to test (can't mock dependencies easily).
- Breaks immutability.
- Not recommended for large projects.

✅ Use Constructor Injection instead!

Creating Custom Beans using @Bean

Sometimes, we need to manually create beans in Spring Boot. We use @Configuration + @Bean for this.

```
@Configuration
public class AppConfig {

    @Bean
    public Engine customEngine() {  // Custom Bean
        return new Engine();
    }
}
```

Bean Scope in Spring Boot

Spring Boot provides different scopes for beans:

Scope	Description
Singleton(Default)	Only one instance of the bean exist
Prototype	A new instance is created every time
Request	New instance for each HTTP request(Web Apps)
Session	New instance for each HTTP session(Web Apps)

Setting Bean Scope

```
@Component
@Scope("prototype")  // New instance every time
class Engine { }
```

✅ Summary

- ✔ Dependency Injection (DI) makes code loosely coupled & testable.
- ✔ IoC (Inversion of Control) lets Spring manage dependencies instead of new().
- ✔ Constructor Injection is the best practice.
- ✔ Setter Injection is useful for optional dependencies.
- ✔ Field Injection should be avoided.
- ✔ Spring Boot Beans can be customized using @Bean & @Configuration.
- ✔ Bean Scope controls how beans are created & reused.

Spring Boot REST API Development🌐

- ✅ What is a REST API?
- ✅ Creating a Spring Boot REST API using @RestController
- ✅ Handling HTTP Methods (GET, POST, PUT, DELETE)
- ✅ Path Variables & Query Parameters
- ✅ Response Status Codes (@ResponseStatus)
- ✅ Exception Handling (@ControllerAdvice)

What is a REST API? 🤔

A REST API (Representational State Transfer API) is a way to communicate between client and server over HTTP.

Follows CRUD operations:

- Create → POST
- Read → GET
- Update → PUT
- Delete → DELETE

📌 Spring Boot makes REST API development easy with @RestController.

Create a REST Controller

- 🔹 Use @RestController to handle HTTP requests.
- 🔹 Use @RequestMapping("/path") to define API routes.

```
@RestController
@RequestMapping("/users")  // Base path for APIs
public class UserController {
    @GetMapping("/hello")  // GET request
    public String sayHello() {
        return "Hello from Spring Boot REST API!";
    }
}
```

✅ Start the application and visit:

```
http://localhost:8080/users/hello
```

📌 You will see "Hello from Spring Boot REST API!"

GET Request - Fetch Data

🔹 @GetMapping is used for retrieving data.

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

    @GetMapping("/{id}")  // GET request with Path Variable
    public String getUserById(@PathVariable int id) {
        return "User ID: " + id;
    }
}
```

✅ Test in Browser/Postman:

```
GET http://localhost:8080/users/5
```

📌 Output: "User ID: 5"

POST Request - Create Data

🔹 @PostMapping is used for creating data. 🔹 Use @RequestBody to get data from the request body.

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

    @PostMapping
    public String createUser(@RequestBody User user) {
        return "User " + user.getName() + " created!";
    }
}
```

✅ Send JSON Data in Postman:

```
POST http://localhost:8080/users
Body: { "name": "Amresh", "email": "amresh@example.com" }
```

📌 Output: "User Amresh created!"

PUT Request - Update Data

🔹 @PutMapping is used for updating data.

```
@PutMapping("/{id}")
public String updateUser(@PathVariable int id, @RequestBody User user) {
    return "User ID " + id + " updated with name " + user.getName();
}
```

✅ Test in Postman:

```
PUT http://localhost:8080/users/5
Body: { "name": "Updated Name" }
```

📌 Output: "User ID 5 updated with name Updated Name"

DELETE Request - Remove Data

🔹 @DeleteMapping is used for deleting data.

```
@DeleteMapping("/{id}")
public String deleteUser(@PathVariable int id) {
    return "User ID " + id + " deleted!";
}
```

✅ Test in Postman:

```
DELETE http://localhost:8080/users/5
```

📌 Output: "User ID 5 deleted!"

Path Variables vs Query Parameters

Path Variable (/{id}) → Used for identification. Query Parameter (?key=value) → Used for filtering/searching. 🔹 Path Variable Example:

```
@GetMapping("/{id}")
public String getUserById(@PathVariable int id) {
    return "User ID: " + id;
}
```

🔹 Query Parameter Example:

```java
@GetMapping("/search")
public String searchUser(@RequestParam String name) {
    return "Searching for user: " + name;
}
```

✅ Test in Browser/Postman:

```
GET http://localhost:8080/users/search?name=Amresh
```

📌 Output: "Searching for user: Amresh"

Returning Response with Status Codes (@ResponseStatus)

🔹 @ResponseStatus helps in customizing HTTP responses.

```java
@GetMapping("/{id}")
@ResponseStatus(HttpStatus.OK)  // Custom Status Code
public User getUser(@PathVariable int id) {
    return new User(id, "Amresh", "amresh@example.com");
}
```

✅ Response Code: 200 OK

Exception Handling in REST APIs (@ControllerAdvice)

Spring provides @ControllerAdvice for global exception handling.

🔹 Handling Resource Not Found Exception

```java
@ResponseStatus(HttpStatus.NOT_FOUND)  // Return 404 Status
class ResourceNotFoundException extends RuntimeException {
    public ResourceNotFoundException(String message) {
        super(message);
    }
}
```

🔹 Handle Exception Globally:

```
@ControllerAdvice
public class GlobalExceptionHandler {

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

✅ Now, if a user is not found, it will return 404 Not Found instead of a generic error.

✅ Summary

- ✔ Spring Boot makes it easy to create REST APIs using @RestController.
- ✔ CRUD operations are implemented using @GetMapping, @PostMapping, @PutMapping, @DeleteMapping.
- ✔ Path Variables (@PathVariable) and Query Parameters (@RequestParam) help in handling requests.
- ✔ Exception Handling (@ControllerAdvice) improves API error responses.