Spring Boot

Why Do We Need Spring Boot?

Spring Boot is an extension of the Spring Framework that simplifies the development of production-ready applications. While Spring Framework is powerful, it involves significant boilerplate configuration, making the development process more complex and time-consuming. Spring Boot addresses these challenges by offering a set of conventions, tools, and auto-configuration features.

Problems with Traditional Spring Development

1. **Extensive Configuration**:

   Developers need to configure beans, data sources, property files, security, etc., manually using XML or Java-based configuration.

2. **Dependency Management**:

   Choosing the correct versions of dependencies and ensuring compatibility between them is complex and error-prone.

3. **Manual Setup**:

   Setting up servers like Tomcat or Jetty, deploying WAR files, and managing the environment requires additional effort.

4. **Difficult Testing**:

   Traditional Spring applications require additional steps for testing, as you often need to mock configurations or deploy the app to test servers.

5. **Slow Development Cycle**:

   Developers need to frequently restart the server for changes to take effect, slowing down the development process.

How Spring Boot Solves These Problems

1. Auto-Configuration

- Spring Boot automatically configures your application based on the dependencies present in your `classpath`.
- For example:
  - If `spring-boot-starter-web` is added, it auto-configures Spring MVC, an embedded Tomcat server, and required beans.

Example Without Configuration:

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

Run the application, and it's live without manual configuration.

---

2. Embedded Servers

- Spring Boot comes with embedded servers like **Tomcat**, **Jetty**, and **Undertow**, eliminating the need to deploy WAR files to an external server. You can run your application as a standalone JAR file.

Example:

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

---

3. Starter Dependencies

- Spring Boot provides **starter dependencies** that include a set of libraries for specific use cases, reducing the need to manually manage dependencies.

Example:

Adding `spring-boot-starter-web` includes:

- Spring MVC
- Jackson for JSON
- Tomcat as an embedded server

Maven:

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

---

4. Spring Boot CLI

- Spring Boot offers a Command Line Interface (CLI) for quickly creating and running Spring applications using Groovy scripts.**Example**:

```
spring run app.groovy
```

---

5. Production-Ready Features

- Spring Boot simplifies monitoring and management with:
  - **Spring Boot Actuator**: Provides metrics, health checks, and monitoring endpoints.
  - **Externalized Configuration**: Easily manage properties for different environments.

---

6. Simplified Testing

- Spring Boot provides testing support with `@SpringBootTest`, allowing you to test your application without extensive configuration.

  Example:

```
@SpringBootTest
public class ApplicationTests {
    @Test
    void contextLoads() {
    }
}
```

7. Rapid Development and Deployment

- Hot reloading with **Spring DevTools**: Automatically restarts the application on code changes.
- Easily deploy Spring Boot applications to the cloud with containerized environments.

Key Features of Spring Boot

Feature	Description
Auto-Configuration	Automatically configures Spring beans based on dependencies.
Starter POMs	Predefined dependency sets for quick project setup.
Embedded Server	Run apps directly without external server setup.
Spring Boot Actuator	Provides operational endpoints for monitoring and managing the application.
Spring Boot CLI	Simplifies application creation using Groovy.
Externalized Configuration	Supports configuration via property files, YAML, environment variables, or command-line args.

When to Use Spring Boot?

1. **Microservices Development**:

   Spring Boot is ideal for creating lightweight, independent services due to its embedded server support and ease of setup.

2. **Quick Prototyping**:

   The auto-configuration and starter dependencies make it easy to create prototypes quickly.

3. **Cloud-Native Applications**:

   Spring Boot integrates well with cloud environments like AWS, Azure, and Google Cloud.

4. **Standalone Applications**:

   Create and run standalone apps with embedded servers like Tomcat or Jetty.

Advantages of Spring Boot

- **Faster Development**: Reduces boilerplate code and manual configuration.
- **Ease of Use**: Comes with sensible defaults and requires minimal setup.
- **Versatile**: Supports monolithic, microservices, REST APIs, and batch processing.
- **Production-Ready**: Includes tools for monitoring, metrics, and diagnostics.
- **Community Support**: Backed by a large community and ecosystem.

Why Choose Spring Boot Over Traditional Spring?

Aspect	Traditional Spring	Spring Boot
Configuration	Requires manual XML/Java-based configuration.	Auto-configured based on classpath dependencies.
Deployment	Deploy WAR files to external servers.	Runs as standalone JAR with an embedded server.
Dependency Setup	Manually manage dependencies and compatibility.	Starter POMs simplify dependency management.

Monitoring	Requires custom setup for metrics and health.	Comes with Actuator for monitoring.

Traditional Spring MVC Web Application vs. Spring Boot

Let's compare the development of a **simple web application** (e.g., a "Hello, World!" app) using **traditional Spring MVC** and **Spring Boot**, highlighting the problems Spring Boot solves.

Traditional Spring MVC Web Application

1. Configuration Complexity

In traditional Spring MVC, you need to configure everything manually.

1. **Web.xml Configuration** (Dispatcher Servlet):

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
             http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
         version="3.0">

    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/spring-config.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
```

```
    </servlet>

    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

2. **Spring Configuration File** (`spring-config.xml`):

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instanc
e"
       xsi:schemaLocation="http://www.springframework.org/
schema/beans
           http://www.springframework.org/schema/beans/spr
ing-beans.xsd">

    <context:component-scan base-package="com.example" />
    <bean class="org.springframework.web.servlet.view.Inte
rnalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/views/" />
        <property name="suffix" value=".jsp" />
    </bean>
</beans>
```

3. **Controller**:

```
@Controller
public class HelloController {
    @RequestMapping("/hello")
```

```
    public String sayHello(Model model) {
        model.addAttribute("message", "Hello, World!");
        return "hello";
    }
}
```

4. **View** (`/WEB-INF/views/hello.jsp`):

```
<html>
<body>
    <h1>${message}</h1>
</body>
</html>
```

Problems in Traditional Spring MVC

Problem	Why It's a Problem
XML Configuration Overhead	Requires verbose `web.xml` and `spring-config.xml`.
Manual Dependency Management	Developers must manage dependencies and versions manually in `pom.xml` or `build.gradle`.
External Server Setup	Requires deployment on external servers like Tomcat or Jetty.
Slow Development Cycle	Every code change requires server restart and re-deployment.
Testing Challenges	Requires additional setup for testing because of tightly coupled configurations.
Fragmented Structure	Configuration files are scattered across multiple locations (`web.xml`, XML configs, etc.).

Spring Boot Web Application

Spring Boot simplifies development by eliminating most of the manual setup.

1. Minimal Configuration

1. **Dependencies** (No manual dependency conflicts):

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

2. **Controller**:

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

3. **Main Application**:

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

4. **Embedded Server**:
    - No need for `web.xml` or an external Tomcat server. The application runs on an embedded Tomcat server.
    - Run the application using:

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

Problems Solved by Spring Boot

Problem	How Spring Boot Solves It
XML Configuration Overhead	Auto-configuration replaces `web.xml` and XML configuration.
Manual Dependency Management	Starter dependencies (e.g., `spring-boot-starter-web`) automatically manage transitive dependencies.
External Server Setup	Comes with an embedded Tomcat server, so no external server setup is needed.
Slow Development Cycle	**Spring DevTools** enables hot reloading for instant feedback during development.
Testing Challenges	Provides out-of-the-box testing support with annotations like `@SpringBootTest`.
Fragmented Structure	A single configuration class (`@SpringBootApplication`) centralizes configuration.

Side-by-Side Comparison

Aspect	Traditional Spring MVC	Spring Boot
Configuration	Requires `web.xml` and multiple XML configuration files.	No XML configuration; uses annotations and auto-configuration.
Dependency Management	Manual management of dependencies and versions.	Uses starter dependencies to reduce configuration.

Server Deployment	Requires external servers (e.g., Tomcat, Jetty).	Embedded servers; runs as a standalone application.
Development Cycle	Slow due to frequent redeployment.	Hot reloading with Spring DevTools.
Testing	Requires additional setup for integration testing.	Simplified testing with `@SpringBootTest`.
Structure	Scattered configuration files.	Centralized configuration in `@SpringBootApplication`.
Ease of Use	Steeper learning curve for new developers.	Developer-friendly and beginner-friendly.

Spring Boot in Action

Run Spring Boot:

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

Access the API:

- Navigate to `http://localhost:8080/hello`.Output:

```
Hello, World!
```

Conclusion

Spring Boot simplifies application development by:

- Eliminating boilerplate configuration.
- Providing embedded servers.
- Simplifying dependency management.

- Enabling rapid development cycles with hot reloading.

By automating tedious tasks, Spring Boot allows developers to focus on writing business logic instead of managing infrastructure, making it an excellent choice for modern web applications.