# Spring Boot & GitHub Actions

## Introduction to Spring Boot and GitHub Actions

- **Why These Tools Matter**:
  - **Spring Boot**: Streamlines Java application development.
  - **GitHub Actions**: Automates CI/CD pipelines, improving delivery speed and reliability.
  - Essential for modern **DevOps practices**.

---

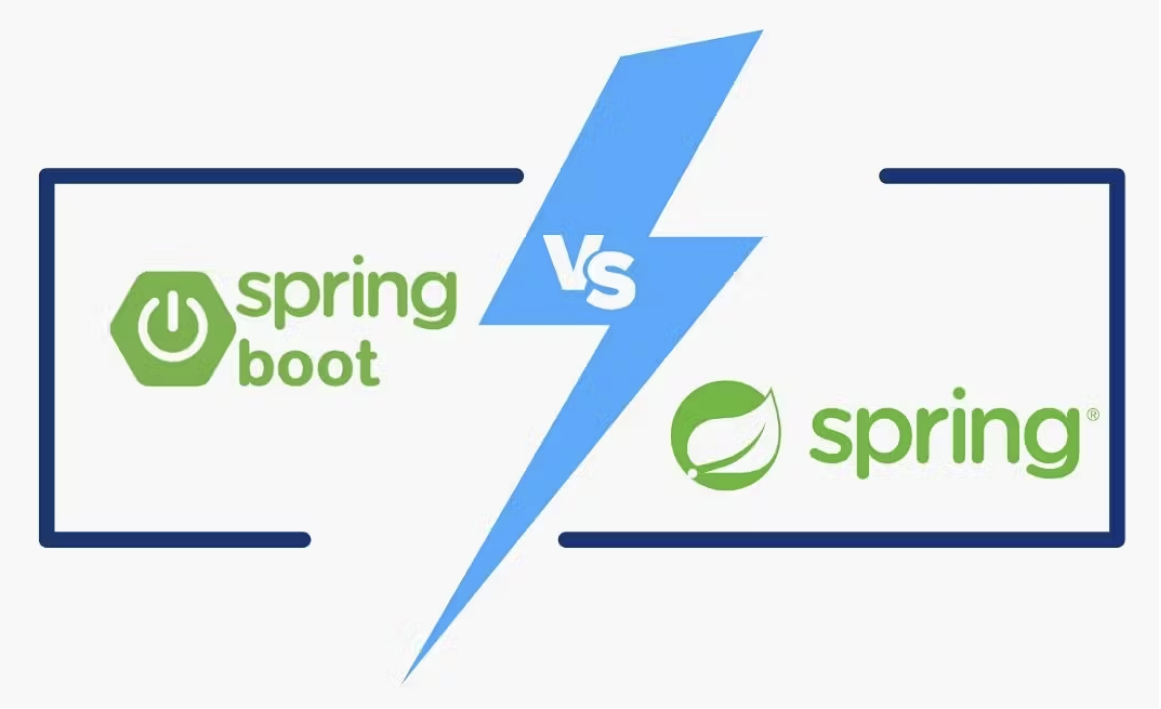
## ▼ Overview of Spring Boot

- **What is Spring Boot?**
  - A framework for building stand-alone Java applications with embedded servers.

    - Provides pre-configured, out-of-the-box functionality to avoid boilerplate code.
- **Why Spring Boot?**
    - Reduces configuration and setup.
    - Focus on **convention over configuration**.
    - Compatible with microservices architecture, REST APIs, and monolithic apps.

---

## ▼ What about Spring ?




- **Spring**: A comprehensive framework for building any Java application, requiring more manual configuration and management of dependencies and application context.
- **Spring Boot**: An extension of the Spring framework aimed at simplifying development, configuration, and deployment, especially for microservices and cloud-based applications.

| Aspect | Spring | Spring Boot |
|---|---|---|
| **Setup** | Manual configuration required. | Automatic configuration with defaults. |

| Aspect | Spring | Spring Boot |
|---|---|---|
| **Embedded Server** | Requires external server (Tomcat, Jetty, etc.). | Comes with an embedded server (Tomcat/Jetty). |
| **Project Complexity** | Highly flexible but requires more setup effort. | Simplifies Spring projects, reducing setup time. |
| **Use Case** | Best for complex, large-scale applications. | Ideal for microservices and fast prototypes. |
| **Deployment** | Requires WAR file and deployment on external server. | Packaged as JAR with an embedded server for easy deployment. |

---

## ▼ Why Choose One Over the Other:

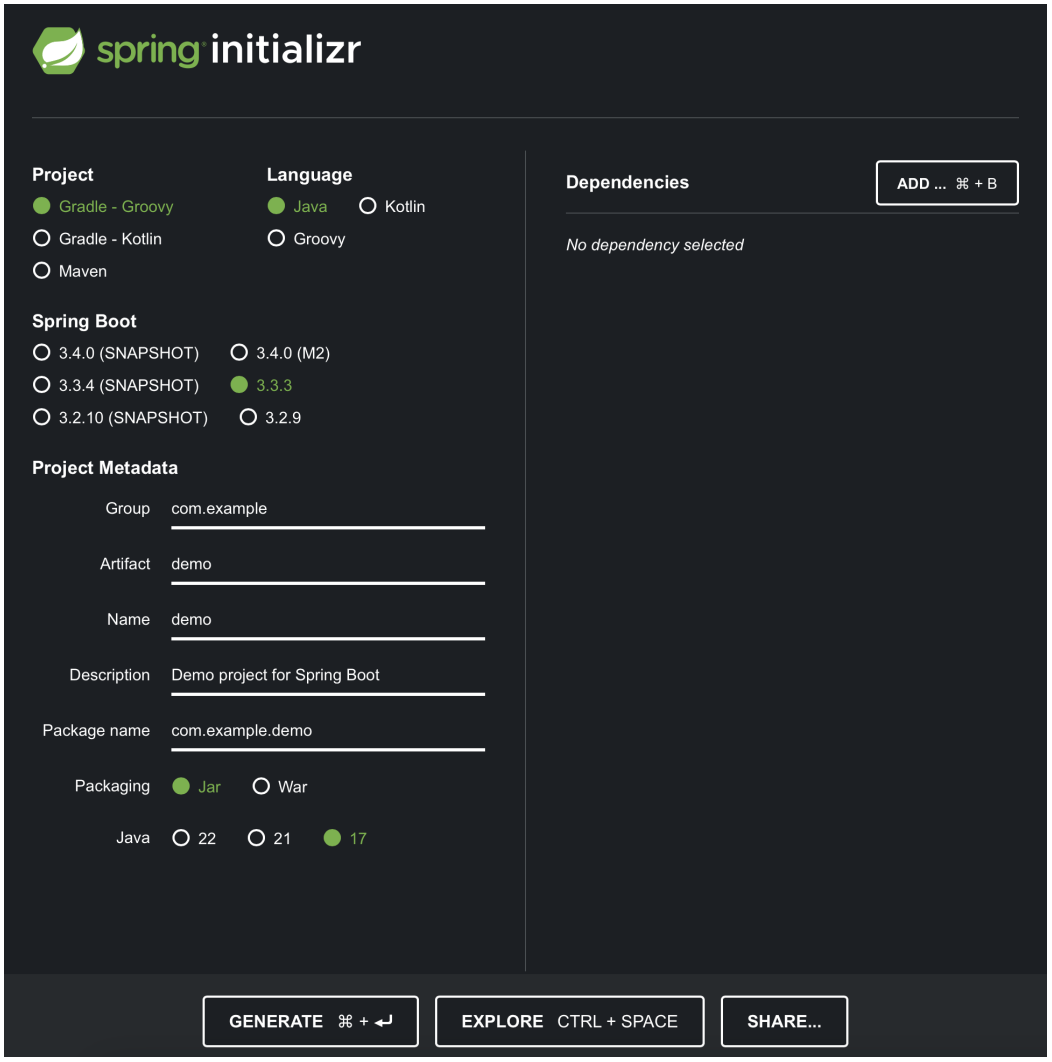
- **Choose Spring** when:
  - Your project needs **extensive customizations**.
  - You're building a **complex enterprise application** where flexibility and modularity are necessary.
  - You have a team experienced in managing detailed configurations.
- **Choose Spring Boot** when:
  - You're building **microservices** or need quick iterations in development.
  - You want an **all-in-one** solution with **auto-configuration**.
  - Your focus is on **simplicity** and **speed** without worrying about configuration details.

| Criteria | Spring | Spring Boot |
|---|---|---|
| **Purpose** | Framework for building complex enterprise-level Java applications. | Simplified framework to quickly build microservices or stand-alone apps. |
| **Development Speed** | Slower to set up due to configuration. | Faster development with minimal setup. |
| **Customization** | Provides maximum flexibility and customization. | Less flexibility, focuses on ease of use. |
| **Project Suitability** | Large-scale, complex, highly customized apps. | Small/medium projects, microservices, rapid |

| Criteria | Spring | Spring Boot |
| --- | --- | --- |
| | | development. |

## ▼ Core Concepts of Spring Boot

- **Spring Boot Starters**:
  - Pre-packaged sets of dependencies that simplify build and configuration.
  - Example: `spring-boot-starter-web` (for building web apps and RESTful APIs).
- **Embedded Servers**:
  - Supports **Tomcat**, **Jetty**, and **Undertow**.
  - No need for separate server setup.
- **Spring Initializr**:
  - Online tool to generate Spring Boot project templates.
  - Customizable dependencies and build tools (Maven/Gradle).
  - Visit: start.spring.io




## ▼ Spring Boot Annotations and Key Components

- **Annotations**:
  - `@SpringBootApplication`: Marks the main class for Spring Boot.

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

  - `@RestController`: Defines REST API controllers.

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

- `@RequestMapping` and `@GetMapping`: Handle HTTP requests.

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

- **Beans and Dependency Injection**:
  - Spring Boot uses **inversion of control (IoC)** to manage beans.
  - `@Autowired`: Injects dependencies automatically.

```
@Autowired
private UserService userService;
```

- **Configuration**:
  - Managed through `application.properties` or `application.yml`.
  - Profiles for different environments (e.g., `dev`, `prod`). Profiles allow you to define different configurations for different environments (e.g., development, testing, production). You can activate profiles with the `spring.profiles.active` property.

  **Example**:

```
# application-dev.yml
spring:
  datasource:
    url: jdbc:mysql://localhost/devDB
```

---

## ▼ Spring Boot Project Structure

- **Typical Structure**:

```
├── src
│   ├── main
│   │   ├── java
│   │   │   └── com.example.demo
│   │   │       └── Application.java
│   │   ├── resources
│   │   │   └── application.properties
│   ├── test
└── pom.xml / build.gradle
```

- **Main Components**:
  - `src/main/java`: Contains Java classes.
  - `src/main/resources`: Configuration files (e.g., `application.properties`).
  - `pom.xml` or `build.gradle`: Defines dependencies and build plugins.

```
project-root/
│
├── src/
│   ├── main/
│   │   ├── java/            # Java source files
│   │   │   └── com/
│   │   │       └── example/
│   │   │           └── MyApp.java
│   │   ├── resources/       # Non-Java resources (properties files, etc.)
│   │   └── webapp/          # Web application resources (for web projects)
│   │
│   └── test/
│       ├── java/            # Test source files
│       │   └── com/
│       │       └── example/
│       │           └── MyAppTest.java
│       └── resources/       # Test resources
│
├── target/                  # Compiled classes and built artifacts
│
├── pom.xml                  # Project Object Model file
│
```

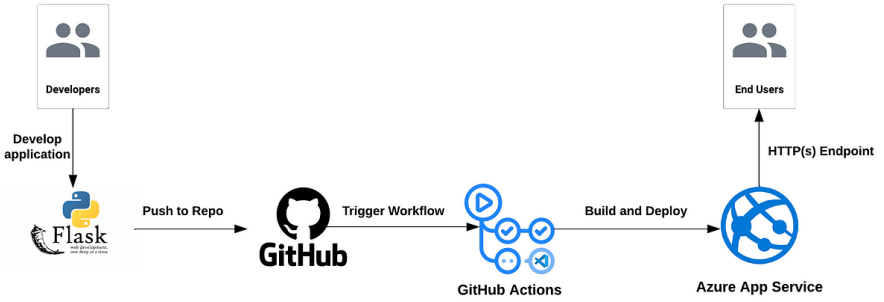
## ▼ Running a Spring Boot Application

- **Steps to Run**:
  1. Clone a Spring Boot project or generate one using Spring Initializr.
  2. Use Maven or Gradle to package and build the application.
  3. Run the application using:
     - `./mvnw spring-boot:run` (Maven wrapper)
     - `./gradlew bootRun` (Gradle wrapper)
  4. Access the app at `localhost:8080` (default port).
- **Useful Tips**:
  - Customize the port with `server.port=8081` in `application.properties`.
  - Use `actuator` for monitoring and health checks (dependency: `spring-boot-starter-actuator`).

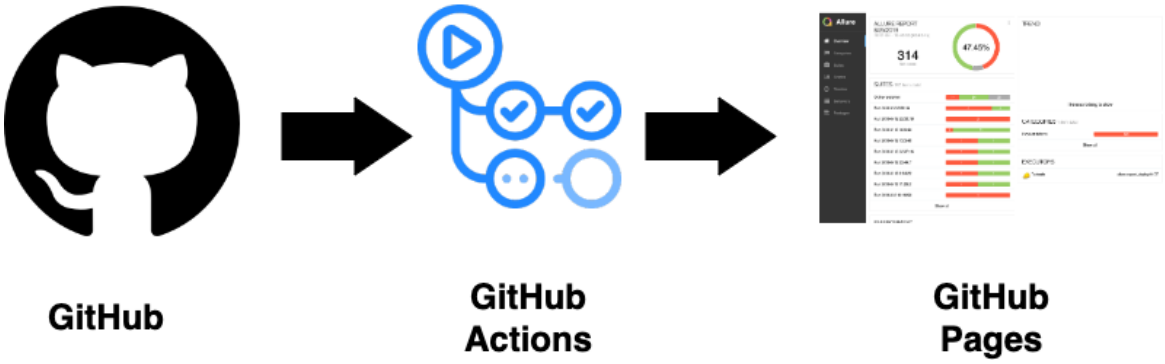
---

## ▼ GitHub Actions

- **What is CI/CD?**
  - **CI** (Continuous Integration): Automatically integrates and tests code on each commit.
  - **CD** (Continuous Deployment/Delivery): Automatically deploys tested code to production or staging.
- **Why GitHub Actions?**
  - Automates your workflow by triggering events like `push`, `pull_request`, and `release`.
  - Easily integrates with other tools like **Docker**, **AWS**, **Heroku**.




## ▼ Key Components of GitHub Actions




- **Workflows**:
  - Defined in **YAML** format (`.github/workflows/` folder).
  - Triggered by events such as `push`, `pull_request`.
- **Jobs**:
  - Define **units of work** that run on a runner (e.g., `ubuntu-latest`, `macos-latest`).
  - Jobs can run sequentially or in parallel.
- **Steps**:
  - Each job consists of a series of **steps** (e.g., checking out the code, building, testing).
- **Runners**:
  - GitHub-hosted runners (e.g., Ubuntu, macOS) execute workflows.
  - Self-hosted runners allow workflows to run on your own infrastructure.

## ▼ Setting Up a Simple CI/CD Pipeline with GitHub Actions

- **Basic Example**:

```
name: Java CI with Maven

on: [push]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
      uses: actions/checkout@v2

    - name: Set up JDK 11
      uses: actions/setup-java@v2
      with:
        java-version: '11'

    - name: Build with Maven
      run: mvn clean install
```

- **Breakdown**:
  - **on: [push]**: Trigger the workflow when a `push` event occurs.
  - **jobs**: Defines the `build` job that runs on `ubuntu-latest`.
  - **steps**:
    - Checkout the code.
    - Set up Java 11.
    - Build the project using Maven.

---

## ▼ Tips and Tricks for GitHub Actions

- **Caching**:
  - Speed up builds by caching dependencies with the `actions/cache` action.

- **Secrets**:
  - Use **GitHub Secrets** to securely store sensitive information (e.g., API keys).
  - Accessible in workflows as `secrets.MY_SECRET_KEY`.
- **Reusability**:
  - Use **composite actions** to define reusable workflows.
- **Debugging**:
  - Add `set -x` to enable debugging in your bash scripts.
  - Use `matrix` for testing across multiple environments (e.g., different Java versions).

---

## ▼ Useful Resources:

Spring Boot

Level up your Java code and explore what Spring can do for you.

https://spring.io/projects/spring-boot

GitHub Actions documentation - GitHub Docs

Automate, customize, and execute your software development workflows right in your repository with GitHub Actions. You can discover, create, and share

https://docs.github.com/en/actions

Guides for GitHub Actions - GitHub Docs

These guides for GitHub Actions include specific use cases and examples to help you configure workflows.

https://docs.github.com/en/actions/guides

---