Class-20: Introduction to Spring Boot

by Pial Kanti Samadder

What is Spring Boot?

Spring Boot is an extension of the Spring Framework, which is widely used in Java application development.

While the Spring Framework provides a comprehensive set of tools for building enterprise-level applications, it can be quite complex to set up and configure. This is where Spring Boot comes into play.

Spring Boot aims to **simplify the development process by providing a set of pre-configured templates, dependencies, and conventions**.

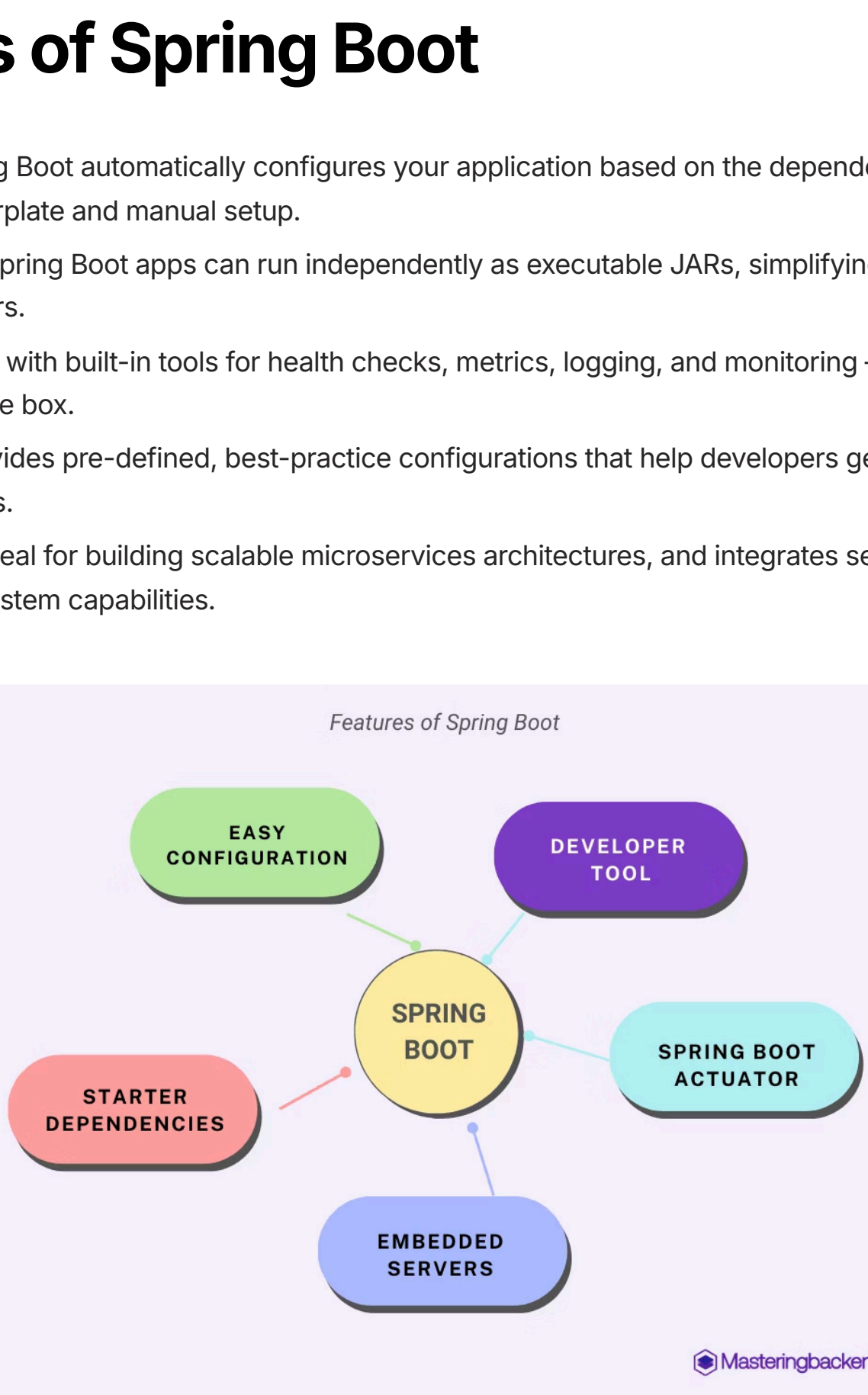
Most Spring Boot applications need very little Spring configuration.

You can use Spring Boot to create Java applications that can be started by using `java -jar` or more traditional war deployments.

Key features of Spring Boot

1. **Auto-Configuration:** Spring Boot automatically configures your application based on the dependencies present in the classpath, minimizing boilerplate and manual setup.
2. **Standalone:** Applications Spring Boot apps can run independently as executable JARs, simplifying deployment and eliminating the need for external servers.
3. **Production-Ready:** Comes with built-in tools for health checks, metrics, logging, and monitoring — making applications ready for production use out of the box.
4. **Opinionated Defaults:** Provides pre-defined, best-practice configurations that help developers get started quickly and maintain consistency across projects.
5. **Microservices-Friendly:** Ideal for building scalable microservices architectures, and integrates seamlessly with Spring Cloud for advanced distributed system capabilities.

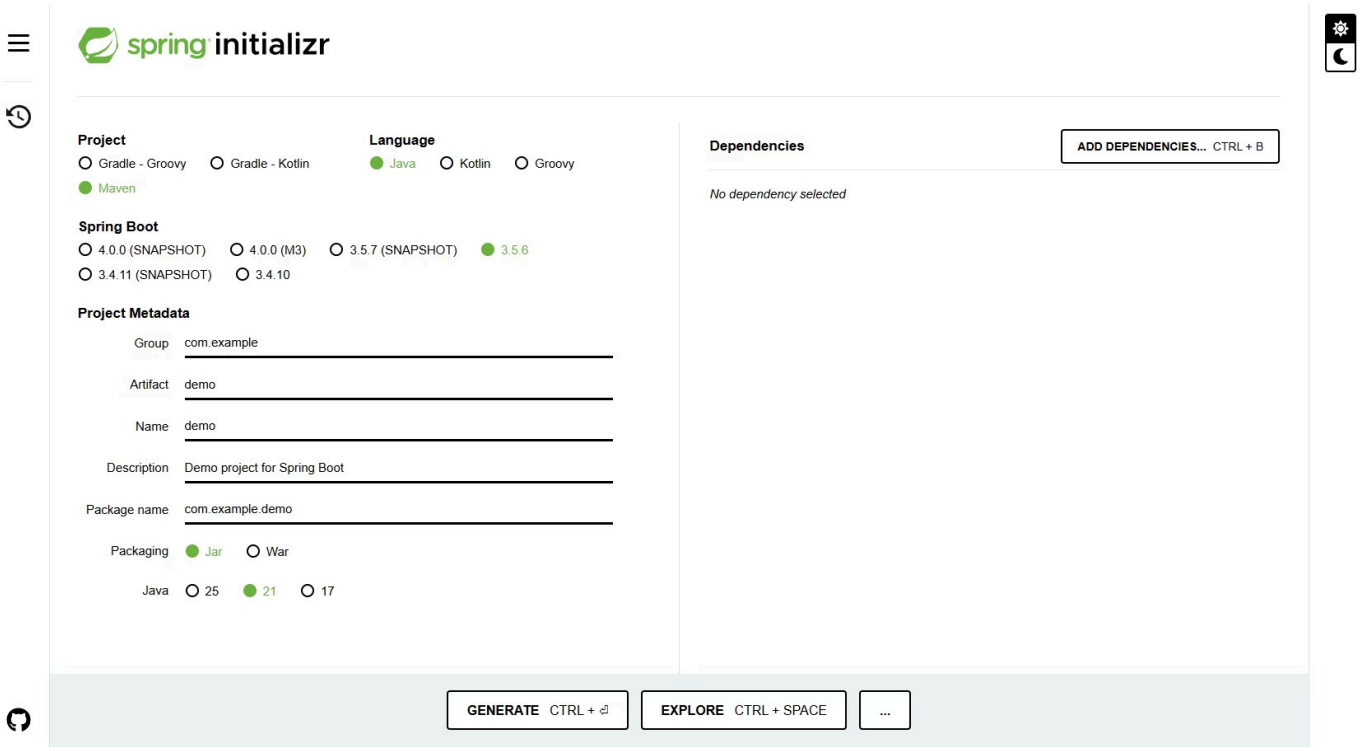
Features of Spring Boot



Creating Your First Spring Boot Project

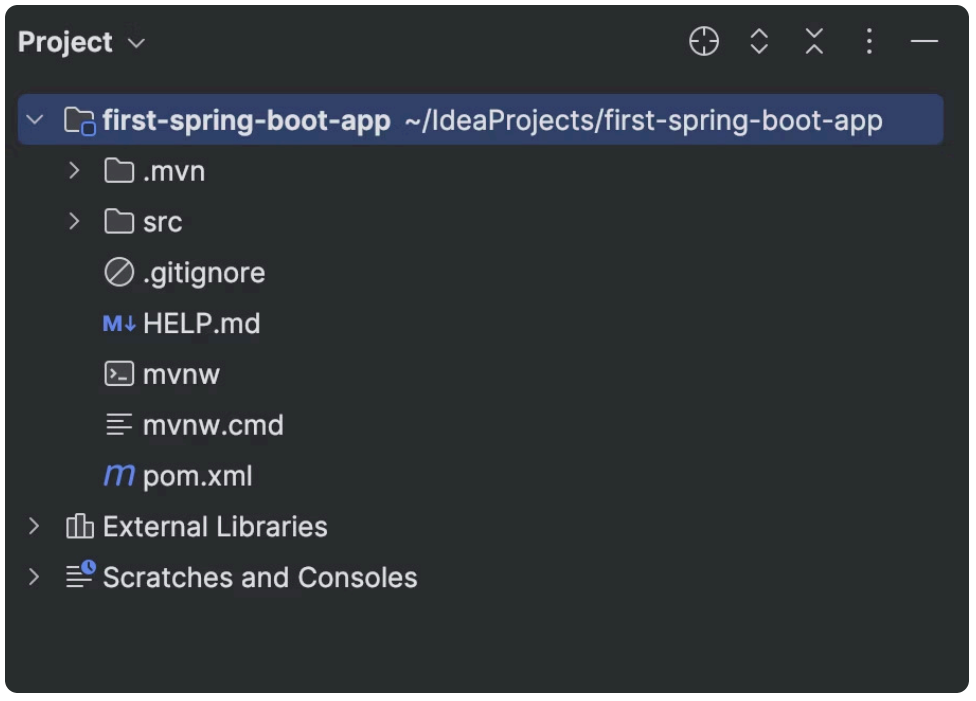
Using Spring Initializer:

Go to [Spring Initializer] in your web browser. The best thing about Spring initializer is user friendly GUI:



1. **Let's Configure Your Project:**
    - Choose the build tool (Maven or Gradle).
    - Select the programming language (Java is recommended).
    - Set the project metadata (group, artifact, name, description, package).
    - Select packaging (JAR is commonly used).
    - Choose your Java version.
2. **Add Dependencies:**
    - Add essential dependencies like **Spring Web**.
    - Rest is up to you. Select the one that suits your need.
3. **Generate and Download:** Click "Generate" to create the project structure and download it as a ZIP file.
4. **Import Project:** Extract the ZIP file, open your IDE, and import the project as a Maven or Gradle project.

Project Structure Overview



At first, you'll have a **structure** like this, and here you'll **find** the `pom.xml` **file**, which represents **Maven** or `build.gradle` if **Gradle** build system is used.

There's also the `src` **directory**, which, when **expanded**, reveals **two** more **directories** called `main` and `test`. The `test` **directory** is used for **writing tests.**

When we **expand** the `main` **directory**, we see **two more directories**: `java` and `resources`.

The `resources` **directory** is used to store **all the resources for our website** along with `application.properties` which manages application configurations such as database configuration, server port, logging level and much more.

Starting Point:

Spring Boot applications typically have a main application class annotated with `@SpringBootApplication`.

`@SpringBootApplication` Annotation

The `@SpringBootApplication` annotation in Spring Boot is a fundamental building block that simplifies the setup of a Spring Boot application. This powerful annotation serves as a combination of three essential Spring annotations:

1. `@Configuration`
2. `@EnableAutoConfiguration`
3. `@ComponentScan`.

`@SpringBootApplication` acts as the main entry point for your application.

@Configuration

- In the Spring Framework, `@Configuration` **marks a class as a source of bean definitions**.
- Beans define objects managed by the Spring container.

```
@Configuration
public class AppConfig {
    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }
}
```

Purpose: Helps define and organize beans manually when customization is needed.

@EnableAutoConfiguration

`@EnableAutoConfiguration` is a Spring Boot-specific annotation that makes the process of configuring beans easier by **automatically setting up components based on the dependencies present in the classpath**. **When this annotation is included, Spring Boot inspects the dependencies in the project (such as JPA, Web, or Security) and configures necessary components accordingly**.

For example, **if you include** `spring-boot-starter-data-jpa` **as a dependency,** `@EnableAutoConfiguration` **will automatically configure a JPA entity manager, a data source, and transaction management, saving you from the manual setup**.

However, there are times when you may want to exclude specific configurations that might not be required for your application.

For example, if your application doesn't interact with a database, you may want to exclude the `DataSourceAutoConfiguration` class:

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

In this example, Spring Boot will skip the auto-configuration for data sources, preventing unnecessary setup and keeping your application's startup time and memory usage optimized.

How `@EnableAutoConfiguration` Works Internally

Behind the scenes, `@EnableAutoConfiguration` leverages the Spring Factories Loader mechanism to locate and load configurations.

Spring Boot checks the classpath for `META-INF/spring.factories` files. Inside this file, it looks for the key `org.springframework.boot.autoconfigure.EnableAutoConfiguration` and gets a list of all classes under this key.

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
...
```

These classes are configurations that Spring Boot applies to its ApplicationContext. It's important to note that these configurations are not applied blindly. Each auto-configuration class is equipped with conditions that decide whether it should be applied.

These conditions are typically annotated with `@ConditionalOnClass`, `@ConditionalOnBean`, `@ConditionalOnMissingBean`, `@ConditionalOnProperty`, etc. These conditions collectively define whether a certain configuration will be applied based on the presence or absence of a specific class, bean, or property.

Customizing Auto-configuration

Though auto-configuration aims to be as intelligent as possible, you might need to tweak its behavior to suit your application's needs.

- **Exclude Auto-configuration Classes**

One of the ways to customize the auto-configuration process is by excluding specific auto-configuration classes that you don't want to be applied. You can do this using the `exclude` attribute of `@EnableAutoConfiguration`:

```
@EnableAutoConfiguration(exclude = DataSourceAutoConfiguration.class)
@ComponentScan(basePackages = "com.example.myapp")
public class MyApp {

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

In this example, `DataSourceAutoConfiguration` will not be applied, even if it meets all other conditions.

- **Overriding Auto-configuration**

Another way to customize is by declaring your beans, which overrides the beans defined in auto-configuration.

For instance, if you define your `DataSource` bean, it will override the auto-configured one.

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MyConfiguration {
    @Bean
    public DataSource dataSource() {
        // return your custom datasource
    }
}
```

@ComponentScan

- The `@ComponentScan` annotation in Spring Framework defines the package or packages to be scanned for components.
- Components in Spring are classes annotated with `@Component`, `@Service`, `@Repository`, or `@Controller`, and they are automatically registered as beans within the Spring IoC (Inversion of Control) container.

By default, when you annotate a class with `@SpringBootApplication`, `@ComponentScan` **is automatically applied to the package where the main application class resides. Spring Boot will scan this package and its sub-packages to find any components that need to be managed by the container**.

Example:

```
@ComponentScan(basePackages = "com.example.service")
public class AppConfig {
    // Configuration details
}
```

Customizing the Component Scan

In some cases, you may want to limit the scanning scope to improve the startup time and memory usage of your application. **You can customize `@ComponentScan` by specifying multiple base packages or using filters to include or exclude specific types of components**:

```
@SpringBootApplication
@ComponentScan(basePackages = {"com.example.app", "com.example.util"})
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

Here, `@ComponentScan` is configured to scan both `com.example.app` and `com.example.util`, making it easier to structure applications with multiple packages.

This flexibility allows for better control over which components are included in the application context, especially in larger applications where packages may be divided based on functionality or modularity.

Benefits of Using `@SpringBootApplication`

- **Unified Configuration:** Combines `@Configuration`, `@EnableAutoConfiguration`, and `@ComponentScan` into one — cleaner and easier to understand.
- **Predictable Behavior:** Automatically configures beans based on dependencies (e.g., adds JPA or Web setup when corresponding starters are included).
- **Flexible Customization:** Supports exclusions and custom scan paths for fine-tuned setups.
- **Faster Development:** Reduces setup time with intelligent defaults and embedded servers — ready to run out of the box.

In short, `@SpringBootApplication` *simplifies configuration, accelerates development, and keeps your codebase clean.*