**Spring Boot Basics**

Spring Boot is a powerful framework that simplifies the setup and development of Spring applications by providing a range of features and conveniences. It is designed to facilitate Rapid Application Development (RAD) by reducing the boilerplate code and configuration that is typically required when using Spring Framework. Here are some definitions and examples to help beginners get started with Spring Boot:

Definitions:

* Spring Boot: A module of the Spring Framework that provides RAD capabilities. It simplifies the configuration and setup process, allowing developers to focus more on writing code for their applications rather than setting up and configuring Spring.
* Starters: Convenient dependency descriptors that include a wide range of dependencies required for developing various types of applications. They help to avoid the need to manually search for and add dependencies, making it easier to get started with specific technologies.
* Spring Boot Starter Web: A starter for building web applications or RESTful services. It includes dependencies for Spring MVC, Jackson for JSON binding, Hibernate Validator for validation, and an embedded servlet container like Tomcat.
* Spring Boot Starter Actuator: A starter for monitoring and managing your application. It provides production-ready features such as metrics, health checks, and externalized configuration.
* Spring Boot Starter Security: A starter for adding security features to your application, including authentication and authorization.
* Spring Boot Starter Thymeleaf: A starter for creating web applications with Thymeleaf, a server-side Java template engine for web applications.
* Spring Boot Starter Test: A starter for writing test cases, providing dependencies for testing Spring applications.

Examples:

Creating a Simple Spring Boot Application:

1. Using Spring Initializr: Visit the [Spring Initializr](https://start.spring.io/) website, fill in your project details, and select the necessary options (e.g., Java version, project type, dependencies like Spring Web, Thymeleaf, etc.). Download the generated project as a zip file and extract it.
2. Building the Application:
   * Open the project in an IDE like Eclipse or IntelliJ IDEA.
   * Navigate to the main application class (usually annotated with @SpringBootApplication).
   * Run the application. Spring Boot will automatically configure and start an embedded server (like Tomcat) and deploy your application.
3. Adding a Controller:
   * Create a new Java class in the src/main/java/com/example/demo directory.
   * Annotate the class with @RestController to indicate it's a web controller.
   * Define a simple method annotated with @GetMapping to handle HTTP GET requests.



1. Running the Application:
   * If you're using an IDE, you can run the application by right clicking the main class and selecting "Run".
   * Access http://localhost:8080/hello in your web browser to see the response from your controller.

Spring Boot's "starters" and auto-configuration make it incredibly easy to get started with Spring development, enabling developers to focus on writing application logic rather than setting up the environment.

Annotations

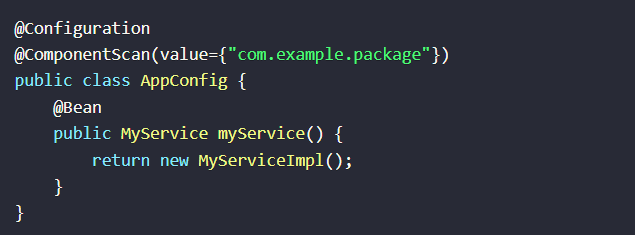
* @Configuration: Marks a class as a source of bean definitions for the application context. It's used in conjunction with @Bean to define beans programmatically.
* @ComponentScan: Tells Spring to scan and automatically detect beans in the specified package. It's typically used with @Configuration.
* @Bean: Marks a method in a configuration class to produce a bean to be managed by the Spring container. It's used to declare beans programmatically.
* @Autowired: Used for automatic dependency injection. Spring looks for matching beans in the application context and injects them where @Autowired is used.
* @Primary: Indicates that a bean should be given preference when multiple beans match a single-valued dependency. It's used in conjunction with @Autowired.
* @Qualifier: Specifies which bean should be autowired when there are multiple beans of the same type. It's used to resolve ambiguity in autowiring.
* @Resource: Similar to @Autowired, but it's part of the JSR 330 standard. It's used for dependency injection.
* @Value: Used to inject values into fields. It can be used to inject values from properties files or directly from the annotation [**5**](https://docs.spring.io/spring-framework/reference/core/beans/java/configuration-annotation.html).
* @PostConstruct and @PreDestroy: Used to mark methods that should be executed after bean initialization (@PostConstruct) or before bean destruction (@PreDestroy)

Dependency Injection

Dependency Injection (DI) is a design pattern that allows for the decoupling of components in an application. In Spring, DI can be achieved through XML configuration or annotation-based configuration. The annotation-based configuration is more modern and is preferred for its simplicity and ease of use.

Annotation-based Configuration

In an annotation-based configuration, you define beans using the @Configuration and @Bean annotations. The @ComponentScan annotation is used to specify which packages Spring should scan for components, configurations, and services.

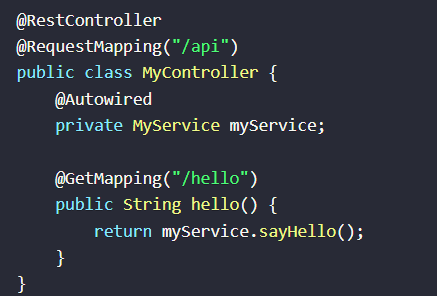


XML-based Configuration

Alternatively, you can configure your Spring application using XML files. This involves defining beans and their dependencies in XML files, which Spring then reads to wire up your application.

Controllers

Controllers in Spring are classes annotated with @Controller (for traditional web applications) or @RestController (for RESTful web services). These classes handle HTTP requests and return responses.



Inversion of Control (IoC)

* Definition: IoC is a design principle where the control of objects is transferred to a container or framework. This means that instead of the application code controlling the instantiation and management of objects, this responsibility is delegated to the Spring IoC container.
* Dependency Injection (DI): IoC is closely related to Dependency Injection (DI), a specific form of IoC where the dependencies of an object are injected into the object by the container, rather than the object having to look up its dependencies. This can be done through constructor arguments, setter methods, or directly into fields.

Spring IoC Container

* BeanFactory: The core IoC container in Spring. It provides advanced configuration mechanisms and is capable of managing any type of object. BeanFactory is a low-level container that provides basic functionality for managing beans.
* ApplicationContext: A more advanced and feature-rich IoC container that extends BeanFactory. It adds easier integration with Spring's AOP features, message resource handling, event publication, and application-layer specific contexts like WebApplicationContext for web applications. ApplicationContext is used exclusively in descriptions of Spring's IoC container.

Dependency Injection in Spring Boot

* @Autowired: Annotates fields, constructors, and setter methods to automatically inject dependencies. Spring looks for matching beans in the application context and injects them where @Autowired is used.
* @Inject: Part of JSR 330, it can be used in place of Spring's @Autowired for injecting dependencies. It's a standard annotation for dependency injection in Java.
* @Resource: Similar to @Autowired, but it's part of the JSR 330 standard. It's used for dependency injection.
* @Value: Used to inject values into fields, either from properties files or directly from the annotation. It's a convenient way to externalize configuration.

Example of IoC in Spring Boot

Consider a simple Spring Boot application that uses a service to perform operations. The service has a dependency on a repository to access data. Here's how IoC can be used to manage these dependencies:



In this example:

* The @Service annotation marks MyService as a bean managed by Spring.
* The @Repository annotation marks MyRepository as a bean that interacts with the database.
* The @Autowired annotation on the constructor of MyService tells Spring to inject an instance of MyRepository into MyService when it creates the MyService bean.
* The @SpringBootApplication annotation on MySpringBootApplication marks it as the entry point of the Spring Boot application, enabling auto-configuration and component scanning.