# Spring Boot Basics

Hi! My name is Deniz, I am 20 years old, and a Software Engineering student from Stuttgart, Germany currently at the end of my 3th semester. This summary has been created as part of my personal revision, where I have compiled and highlighted the most important core concepts of Spring Boot. The goal of this summary is to provide a concise overview of the essential topics, allowing for a better understanding and quick reference to the key aspects of Spring Boot as I continue to develop my skills. I hope this summary helps you in your learning process, and I wish you much success and enjoyment in exploring the topic!

If you have any questions or would like to connect, feel free to reach out to me via email at deniz.altunkapan@outlook.com or check out my GitHub at DenizAltunkapan.

# Contents



# 1. Introduction

## 1.1. What is Spring Boot?

Spring Boot is an open-source framework built on the Spring Framework, designed to simplify Java application development. It emphasizes convention-over-configuration, reducing boilerplate code and offering auto-configuration for faster setup. With embedded servers and starter dependencies, it enables standalone, production-ready applications with minimal effort. Spring Boot is ideal for building modern, scalable web applications efficiently.

## 1.2. What is Maven?

Maven is a build automation and project management tool used primarily for Java projects. It uses a `pom.xml` file, which acts like a “shopping list” for dependencies. In this file, you specify the libraries and frameworks your project needs, and Maven automatically downloads and manages them. This simplifies dependency management and ensures consistent builds across different environments.

## 1.3. Spring Boot Starters

Spring Boot Starters are pre-configured sets of dependencies that simplify adding common functionality to your Spring Boot application. Each starter includes a group of libraries for a specific task (e.g., web development, data access, security), allowing developers to quickly integrate them without manual configuration.

For example, spring-boot-starter-web bundles dependencies for building web applications, including Spring MVC (for handling HTTP requests and responses) and an embedded Tomcat server. spring-boot-starter-data-jpa provides everything needed for working with databases using JPA and

Hibernate, while spring-boot-starter-security integrates authentication and authorization features. Additionally, spring-boot-starter-actuator adds built-in endpoints for monitoring and managing the application, exposing health status, metrics, and other operational data.

#### 1.3.1. What is Tomcat and a Web Server?

A web server like Tomcat is responsible for hosting and serving web applications. It listens for incoming HTTP requests, processes them (e.g., runs business logic, accesses a database), and sends back responses, typically in the form of HTML, JSON, or other content types.

In the case of Spring Boot, Tomcat is embedded within the application, meaning it runs inside your project instead of being a separate software that needs to be installed and configured. This makes deployment and development much easier. Spring Boot simplifies the use of Tomcat by including it as part of the spring-boot-starter-web starter. When you run your Spring Boot application, an embedded Tomcat web server will automatically start up, without any need for external server configuration.

Running the server: You can run the application easily from your IDE (like IntelliJ) or from the command line. Listening on a port: By default, the embedded Tomcat server listens on port 8080 for incoming HTTP requests. This means that once the application is started, it will be ready to respond to requests on **http://localhost:8080** . For example, if you define a simple REST API, the Tomcat server will listen on port 8080, and any HTTP requests sent to this port will be handled by your application.

## 1.4. Spring Boot Initializr

The Spring Boot Initializr is the ideal starting point for creating a Spring Boot application. It simplifies the process by allowing you to quickly generate a new project with the necessary setup and dependencies. You can access it here. When creating a project, you can select from various **Spring Boot Starters** , which are pre-configured sets of dependencies designed for specific functionality, such as web development, data access, or security. The Initializr automatically adds these Starters to your project's configuration files (like pom.xml for Maven), ensuring all the required libraries are included. This streamlined setup allows developers to focus on building the application rather than managing dependencies, making the Spring Boot Initializr the perfect entry point for any Spring Boot project. **Popular Spring Boot Starters are:**

#### 1.4.1. spring-boot-starter-web

The spring-boot-starter-web starter provides all necessary dependencies for building web applications and RESTful APIs using Spring MVC. It includes Spring Web, an embedded Tomcat server (or Jetty/Undertow), and Jackson for JSON processing. This starter allows developers to create powerful web applications with minimal configuration. By default, it runs on port 8080, and the embedded server eliminates the need for an external web server installation.

#### 1.4.2. spring-boot-starter-data-jpa

The spring-boot-starter-data-jpa starter includes everything needed to work with relational databases using JPA (Java Persistence API) and Hibernate as the default ORM (Object-Relational Mapping) implementation. It simplifies database interactions by allowing developers to use object-oriented programming principles instead of writing raw SQL. With Spring Data JPA, developers can define repositories that automatically generate database queries based on method names, reducing boilerplate code significantly.

#### 1.4.3. spring-boot-devtools

The spring-boot-devtools starter is a development utility that enhances productivity by enabling features like automatic application restart, live reload, and disabling caching during development. Whenever a code change is detected, Spring Boot automatically restarts the application, allowing developers to see updates in real-time without manually restarting the server. Additionally, it improves debugging by offering better exception messages. However, this starter is only recommended for development environments, as it should not be used in production.

#### 1.4.4. spring-boot-starter-security

The spring-boot-starter-security starter provides authentication and authorization features using Spring Security. By default, it secures all endpoints and requires authentication, even without additional configuration. Developers can implement various authentication mechanisms, such as in-memory authentication, database-based authentication, OAuth2, or JWT (JSON Web Token). It also supports encryption, password hashing, and integration with external identity providers (e.g., Google, Keycloak).

#### 1.4.5. spring-boot-starter-validation

The spring-boot-starter-validation starter enables Bean Validation using Jakarta Validation (formerly Javax Validation). It allows developers to enforce constraints on model attributes using annotations like @NotNull, @Size, @Email, or @Pattern, ensuring data integrity before processing requests. This starter is commonly used in combination with Spring MVC for validating request bodies in REST APIs and with Spring Data JPA for validating database entities before persisting them.

# 2. Key Concepts

## 2.1. Bean

In Spring Boot, a Bean is a simple Java object that is managed by the Spring Framework. Spring takes care of the creation, configuration, and lifecycle of these objects. When you annotate a class with a special annotation like @Component, @Service, @Repository, or @Controller, Spring treats this class as a Bean and includes it in the Spring IoC Container (ApplicationContext).

The ApplicationContext is the central concept in Spring and acts as a container that manages all Beans. You can think of the ApplicationContext as a kind of “registry” where all Beans managed by Spring are stored. It ensures that Beans are instantiated, configured, and, if necessary, connected to each other – all automatically, without you having to worry about the details of object creation.

For example, if you mark a class with @Component, Spring will create a Bean for this class in the ApplicationContext. In doing so, Spring handles both the creation of the object and the setting of dependencies that the Bean might need. These dependencies are automatically injected through Dependency Injection (DI). This means that Spring automatically detects the required dependencies (other Beans) and injects them, without you having to manually create or link objects.

By default, in Spring Boot, the ApplicationContext is scanned within the package where the main class (@SpringBootApplication) is located. This means that all subpackages within this base package are also automatically scanned.

## 2.2. Inversion of Control

Inversion of Control (IoC) is a fundamental principle in software development that states that the control over object creation and management is transferred from the application itself to a framework. Instead of a developer manually creating objects and managing their dependencies, the Spring Framework takes over this task automatically.

### 2.3. Dependency Injection

Dependency Injection (DI) is one of the core concepts of the Spring Framework and a key mechanism for implementing Inversion of Control (IoC). It enables loose coupling of components and improves the maintainability, testability, and flexibility of applications. Dependency Injection is a design pattern in which the dependencies of an object (e.g., other classes or services) are provided from the outside, rather than being created or managed within the object itself. Spring Boot uses annotations to identify and inject dependencies. To inject dependencies into a Bean, Spring uses the @Autowired annotation. When you annotate a field, constructor, or setter method with @Autowired, Spring automatically provides the required dependency from the ApplicationContext.

Constructor injection is preferred in Spring because it makes a class's dependencies explicit and immutable, improving maintainability and testability. All required dependencies are provided when the object is created, reducing the risk of `null` values. Field injection, where dependencies are set via fields, is less transparent and complicates testing. Therefore, constructor injection is generally recommended.

### 2.4. Lazy Initialization

Lazy initialization in Spring Boot (@Lazy) refers to a technique where Spring delays the instantiation of beans (components) until they are actually needed, instead of creating them immediately when the application starts. This can reduce the application's startup time and save resources, as not all beans are loaded at the beginning of the application.

## 3. Database Management with JPA and Hibernate

### 3.1. JPA

The Jakarta Persistence API (JPA) is a specification that describes how Java objects can be stored and managed in relational databases. JPA enables object-oriented database interaction by providing an abstraction over SQL queries. Instead of manually writing SQL, developers can work with Java classes that are automatically mapped to database tables.

JPA provides a set of standard annotations that define how a Java class is transformed into a database table. For example, @Entity marks a class as a persistent entity, while @Table(name = "users") specifies the name of the associated table. The primary key of an entity is defined with @Id, and relationships between entities can be modeled using @OneToMany or @ManyToOne.

Since JPA is only a specification, it does not provide its own implementation. This means that JPA alone does not execute SQL commands or interact with the database. Instead, it requires a concrete implementation to translate JPA rules into executable code. One of the most well-known and widely used implementations is Hibernate.

### 3.2. Hibernate

Hibernate is a powerful ORM (Object-Relational Mapping) library that serves as an implementation of JPA. It handles the actual work of database communication and ensures that Java objects can be stored, updated, and retrieved without developers having to write SQL queries.

When a JPA-annotated class is defined, Hibernate automatically generates the corresponding SQL table and creates the appropriate SQL statements for CRUD operations (Create, Read, Update, Delete). For example, a developer can save an object of a User class using the method userRepository.save(user); – Hibernate internally converts this into an INSERT SQL statement.

Additionally, Hibernate offers advanced features that go beyond the JPA standard, such as Lazy Loading, which delays the loading of data, or Caching, which improves performance by storing the

results of database queries. In summary: Spring Boot uses Hibernate as the default implementation for JPA. This means developers can work with JPA annotations while Hibernate automatically handles the database logic. This significantly simplifies development, as Spring Boot automatically configures many settings.

Since JPA is a general specification, Hibernate could be replaced by another implementation such as EclipseLink or OpenJPA. However, Hibernate is the preferred choice because it is widely used, well-optimized, and seamlessly integrated into Spring Boot.

# 4. Creating REST Endpoints with CRUD

## 4.1. What is REST?

**REST (Representational State Transfer)** is an architectural style for developing web services (APIs). It is not a specific technology or framework, but rather a collection of principles and best practices that describe how resources should be accessed over the HTTP protocol. REST is based on the idea that everything is considered a resource (e.g., users, products, orders) that is identified by a unique URL (Uniform Resource Locator). Interaction with these resources occurs through HTTP methods like GET, POST, PUT, and DELETE, which correspond to the **CRUD** operations (Create, Read, Update, Delete).

CRUD (Create, Read, Update, Delete) is a fundamental concept in data management and is used in almost every application that stores and manipulates data. In RESTful APIs, CRUD operations are mapped to HTTP methods:

- Create (POST): A new resource is created (e.g., a new user).
- Read (GET): A resource or a list of resources is retrieved (e.g., all users or a specific user).
- Update (PUT/PATCH): An existing resource is updated (e.g., modifying user data).
- Delete (DELETE): A resource is deleted (e.g., removing a user).

REST is characterized by the following properties:

- Resource Orientation: Everything is considered a resource, identified by URLs.
- Use of HTTP Methods: CRUD operations are mapped to standard HTTP methods.
- Statelessness: Each request to the server contains all the information needed for processing. The server does not store session data between requests.
- Representations: Resources can be represented in different formats (e.g., JSON, XML). JSON is the most commonly used format.
- Scalability: Because REST is stateless, it can easily be scaled by adding more servers.

REST is implemented in web applications through frameworks and libraries like Spring Boot. In Spring Boot, a Java development platform based on the Spring Framework, there is extensive support for building RESTful web services. With Spring Boot, you can define REST endpoints (API routes) and use the corresponding HTTP methods to access and manipulate resources.

Implementing a RESTful web service in Spring Boot is simple. You create a controller class that works with HTTP requests (such as GET, POST, PUT, DELETE) and implements the corresponding methods to access or modify resources. Spring Boot uses the Spring Web library, which enables you to create REST APIs easily.

# 5. Application Properties

## 5.1. What is application.properties?

In Spring Boot, application.properties (or application.yml) files are used for configuring your application. These configuration files allow you to define various settings such as server ports, database connections, logging levels, and custom variables. Spring Boot automatically reads these files at runtime and applies the configurations to the application. The application.properties file uses a key-value pair format, where each property is defined on a new line. For example:

```
server.port=8080
spring.datasource.url=jdbc:mysql://localhost:3306/mydatabase
spring.datasource.username=root
spring.datasource.password=secret
logging.level.org.springframework=DEBUG
```

## 5.2. Custom Properties

Define your own properties and access them in your application using the @Value annotation or @ ConfigurationProperties. For example:

```
app.name=My Spring Boot Application
app.version=1.0.0
used in:
@Value("${app.name}")
private String appName;

@Value("${app.version}")
private String appVersion;
```