## 1. What is Spring?

Spring is a comprehensive framework for developing Java applications, designed to simplify the development of enterprise applications by providing a programming model based on POJOs (Plain Old Java Objects) and a consistent way to manage dependencies and transactions.

Key Features of Spring:

- Inversion of Control (IoC): Spring manages the dependencies of objects and injects them where needed, facilitating decoupling and testing of components.
- Aspect-Oriented Programming (AOP): Allows for the separation of cross-cutting concerns such as security, transactions, and logging.
- Declarative Transactions: Simplifies transaction management in Java applications.
- MVC Framework: Provides a Model-View-Controller framework for developing robust web applications.

Advantages of Using Spring:

- Modularity: Spring is divided into multiple modules, allowing developers to use only the parts they need.
- Flexibility: Supports a wide range of technologies and frameworks.
- Ease of Testing: IoC makes it easy to create modular, testable applications.

### Benefits of Using Spring Boot

- Simplified Configuration: Auto-configuration eliminates the need for extensive manual setup.
- Rapid Development: Offers templates and tools that facilitate quick and efficient development.
- Standalone Applications: Applications can run as standalone JAR files, simplifying deployment.
- Seamless Integration: Easily integrates with other Spring ecosystem components.

### Default Configurations in Spring Boot

Default Embedded Server:

- Spring Boot includes embedded servers like Tomcat, Jetty, and Undertow. By default, it uses Tomcat.
- The embedded server listens on port 8080 by default, which can be configured in `application.properties` with `server.port`.

Thread Configuration:

- HTTP Server (Tomcat):

    - Maximum number of threads: 200 (configurable in `application.properties` with `server.tomcat.max-threads`).
    - Minimum number of threads: 10 (configurable in `application.properties` with `server.tomcat.min-spare-threads`).

Database Connection Configuration:

- HikariCP: The default connection pool in Spring Boot.
    - Maximum pool size: 10 (configurable in `application.properties` with `spring.datasource.hikari.maximum-pool-size`).
    - Minimum idle connections: 1 (configurable in `application.properties` with `spring.datasource.hikari.minimum-idle`).

Other Default Values:

- Connection timeout: 30 seconds (configurable in `application.properties` with `spring.datasource.hikari.connection-timeout`).
- Maximum lifetime of a connection: 30 minutes (configurable in `application.properties` with `spring.datasource.hikari.max-lifetime`).

These default values can be adjusted according to the specific needs of the application, providing a balance between simplicity and flexibility.

## 1. What is Spring?

### 1.2. Aspect-Oriented Programming (AOP)

AOP is a programming paradigm that allows you to modularize cross-cutting concerns of an application, such as security, transaction management, or logging.

How does AOP work in Spring?

1. Join points: These are points in the execution flow of an application, such as the beginning or end of a method.
2. Pointcuts: These are expressions that define a set of join points.
3. Advice: This is the code that is executed before, after, or around a join point.
4. Aspect: Combines a pointcut and an advice to define an aspect.

Spring AOP provides a mechanism for defining aspects and applying them to beans. In this way, you can add functionality to objects without modifying their source code.

## 1. What is Spring?

### 1.3. Declarative Transactions

Declarative transactions allow you to manage transactions in Spring in a simple way. Instead of writing explicit code to start, commit, or rollback transactions, you use annotations to indicate where transactions should start and end.

How do declarative transactions work in Spring?

Spring uses the `PlatformTransactionManager` interface to manage transactions. Annotations like `@Transactional` are used to mark methods that should execute within a transaction.

- Transaction propagation: Defines how nested transactions are handled.
- Isolation: Defines the isolation level of the transaction.
- Timeout: Defines the maximum time a transaction can be active.
- Read-only: Indicates if the transaction is read-only.

Benefits of declarative transactions:

- Simplification: Reduces the amount of code required to manage transactions.
- Centralization: Transaction management logic is centralized in one place.
- Flexibility: Allows for easy configuration of transaction properties.

## 2. Spring Initializr

### What is Spring Initializr?

Spring Initializr is a web-based tool that helps developers quickly generate a Spring Boot project with the necessary dependencies and configurations. It simplifies the initial setup of a Spring Boot project, allowing developers to focus on writing application logic rather than boilerplate code.

Key Features of Spring Initializr:

- Web Interface: Provides a user-friendly web interface to configure and generate a new Spring Boot project.
- Dependency Management: Allows selection of dependencies needed for the project, automatically adding them to the project configuration.
- Project Structure: Sets up the basic structure of a Spring Boot project, including necessary files and directories.
- Build Tool Integration: Supports Maven and Gradle build tools, generating appropriate configuration files.

### Accessing Spring Initializr

You can access Spring Initializr through its web interface at start.spring.io.

## 2. Spring Initializr

### 2.1. Configuring a New Project

When you visit the Spring Initializr website, you will see a form to configure your new project. The following steps outline how to configure and generate a Spring Boot project:

1. Project Metadata:
    - Project: Choose between Maven and Gradle as your build tool.
    - Language: Select the programming language (Java, Kotlin, or Groovy).
    - Spring Boot Version: Choose the Spring Boot version to use.
2. Project Details:
    - Group: The group ID for your project (e.g., `com.example`).
    - Artifact: The artifact ID for your project (e.g., `demo`).
    - Name: The name of your project (e.g., `demo`).
    - Description: A brief description of your project.
    - Package Name: The base package name for your project (e.g., `com.example.demo`).
3. Project Options:
    - Packaging: Choose between `jar` and `war` packaging.
    - Java Version: Select the Java version to use (e.g., `17, 21`).
4. Dependencies:
    - Dependencies: Select the dependencies needed for your project by typing in the search box and selecting from the suggestions. Common dependencies include:
        - Spring Web: For building web applications, including RESTful services.
        - Spring Data JPA: For data access using JPA.
        - H2 Database: For using an in-memory database.
        - Spring Boot DevTools: For developer tools that enhance the development experience.

## 2. Spring Initializr

### 2.2. Generating the Project

Once you have configured the project settings and selected the dependencies, click the "Generate" button. This will download a ZIP file containing your new Spring Boot project.

### Importing the Project into an IDE

After downloading the project ZIP file, you need to import it into your Integrated Development Environment (IDE) to start development. Follow these steps to import the project:

1. Extract the ZIP File: Unzip the downloaded file to a directory of your choice.
2. Open the IDE: Open your preferred IDE (e.g., IntelliJ IDEA, Eclipse, VS Code).
3. Import the Project:

    - For IntelliJ IDEA:
        1. Select `File` > `Open`.
        2. Navigate to the unzipped project directory and select it.
        3. Click `Open` to import the project.
    - For Eclipse:
        1. Select `File` > `Import`.
        2. Choose `Existing Maven Projects` and click `Next`.
        3. Navigate to the unzipped project directory and click `Finish`.
    - For VS Code:
        1. Open the Command Palette (`Ctrl+Shift+P`).
        2. Select `Java: Create Java Project` and choose `Spring Boot`.
        3. Select the unzipped project directory.
4. Build the Project:
    - For Maven: Run `mvn clean install` to build the project.
    - For Gradle: Run `./gradlew build` to build the project.
5. Run the Project:
    - In your IDE, locate the `DemoApplication` class (or the main class in your project).
    - Right-click the class and select `Run` or use the IDE's run configuration to start the application.

By following these steps, you will have a Spring Boot project up and running, ready for further development.

## 3. Build Tools

**Introduction to Build Tools**

Build tools such as Maven and Gradle are essential for managing project dependencies, automating the build process, and simplifying project configurations. They streamline the development workflow by handling tasks like compilation, packaging, deployment, and dependency management.

Why Use Build Tools?

- Automation: Automate repetitive tasks like compilation, testing, and packaging, reducing manual effort and errors.
- Dependency Management: Automatically download and manage libraries and frameworks required by the project.
- Consistency: Ensure consistent builds across different environments by standardizing the build process.
- Efficiency: Speed up the development process by handling complex build processes and configurations.

- Integration: Easily integrate with continuous integration/continuous deployment (CI/CD) pipelines and other tools.

Maven Overview:

- XML Configuration: Uses an XML file (`pom.xml`) to define project dependencies, build plugins, and configurations.
- Convention over Configuration: Follows a standard project structure and build lifecycle, reducing the need for custom configurations.
- Dependency Management: Manages project dependencies and transitive dependencies, ensuring that all required libraries are available.
- Plugins: Provides a wide range of plugins to extend functionality, such as code analysis, documentation generation, and deployment.

Gradle Overview:

- Groovy/Kotlin DSL: Uses a more flexible and readable DSL for configuration (`build.gradle` or `build.gradle.kts`).
- Incremental Builds: Optimizes the build process by only rebuilding parts of the project that have changed.
- Multi-Project Builds: Supports complex project structures with multiple modules and dependencies.
- Customization: Allows extensive customization of the build process through scripting and plugins.
- Dependency Management: Like Maven, it manages project dependencies and handles transitive dependencies.

Advantages of Maven:

- Standardization: Enforces a standardized project structure and build lifecycle.
- Extensive Plugin Ecosystem: Offers a vast array of plugins for different tasks and integrations.
- Large Community Support: Well-established and widely used, with extensive documentation and community support.

Advantages of Gradle:

- Flexibility: Provides more flexibility and customization options compared to Maven.
- Performance: Faster build times due to incremental builds and better dependency resolution.
- Groovy/Kotlin DSL: Easier to write and maintain build scripts using a more expressive DSL.
- Rich API: Offers a rich API for build script customization and plugin development.

By choosing the appropriate build tool and leveraging its features, developers can significantly improve their productivity and ensure consistent, reliable builds for their Spring Boot projects.

## 3. Build Tools

### 3.2. Gradle

**Gradle Dependency Management**

build.gradle is the primary configuration file for Gradle projects. It defines the dependencies, build tasks, and configurations.

Basic Structure of `build.gradle`:

```
plugins {

    id 'org.springframework.boot' version '2.5.4'

    id 'io.spring.dependency-management' version '1.0.11.RELEASE'

    id 'java'

}


group = 'com.example'

version = '0.0.1-SNAPSHOT'

sourceCompatibility = '11'


repositories {

    mavenCentral()

}


dependencies {

    implementation
'org.springframework.boot:spring-boot-starter-web'

    implementation
'org.springframework.boot:spring-boot-starter-data-jpa'
```

```
    runtimeOnly 'com.h2database:h2'

    developmentOnly
'org.springframework.boot:spring-boot-devtools'

    testImplementation
'org.springframework.boot:spring-boot-starter-test'

}



tasks.named('test') {

    useJUnitPlatform()

}
```

Key Elements of `build.gradle`:

- plugins: Specifies the plugins required for the project.
- group: Unique identifier of the project's group.
- version: Project version.
- sourceCompatibility: Java version compatibility.
- repositories: Specifies the repositories from which to download dependencies.
- dependencies: Lists the dependencies required for the project.

## 3. Build Tools

### 3.3. Dependencies

Common Dependencies:

- Spring Boot Starter Web:
  - Purpose: Provides the necessary dependencies for developing web applications, including support for RESTful APIs and embedded servers (like Tomcat or Jetty). It includes dependencies for Spring MVC, Jackson (for converting Java objects to JSON), and other essential web components.
  - Use Case: Creating REST controllers to handle HTTP requests in a web application.
- Spring Boot Starter Data JPA:
  - Purpose: Facilitates data access using JPA (Java Persistence API) and Spring Data JPA. It includes a JPA implementation, such as Hibernate, and

    provides default configurations for interacting with relational databases, simplifying the creation of repositories and managing entities.
  - Use Case: Persisting and querying data in a relational database using JPA entities and Spring Data repositories.
- H2 Database:
  - Purpose: A lightweight, in-memory database mainly used for testing and development. It can be used to store temporary data during development without needing to set up an external database.
  - Use Case: Using an in-memory database for unit tests or rapid development without the overhead of configuring a full-fledged database.
- Spring Boot DevTools:
  - Purpose: Provides tools to enhance the development process, such as automatic application reloads and optimized development configurations. It includes features that improve the development experience, such as hot-swapping of code changes without restarting the application.
  - Use Case: Enhancing productivity during development by allowing automatic reloads of the application when changes are made to the source code.
- Spring Boot Starter Test:
  - Purpose: Includes the necessary dependencies for testing Spring Boot applications, such as JUnit, Mockito, and Spring Test. It provides tools and configurations for writing and running unit and integration tests.
  - Use Case: Writing and executing tests to ensure that the application code functions correctly and meets specified requirements.

Adding and Managing Dependencies

Adding a Dependency: To add a new dependency, include it in the `<dependencies>` section of `pom.xml` for Maven or in the `dependencies` block of `build.gradle` for Gradle.

Example: Adding Lombok

- Maven:

```
<dependency>

    <groupId>org.projectlombok</groupId>

    <artifactId>lombok</artifactId>

    <version>1.18.20</version>

    <scope>provided</scope>

</dependency>
```

- **Gradle:**

```
implementation 'org.projectlombok:lombok:1.18.20'
```

```
annotationProcessor 'org.projectlombok:lombok:1.18.20'
```

Managing Dependencies:

- Maven: Use the `mvn dependency:tree` command to view the dependency tree and detect conflicts.
- Gradle: Use the `./gradlew dependencies` command to list dependencies and check for conflicts.

By understanding and managing dependencies in Maven and Gradle, you ensure your Spring Boot project is correctly configured and all required libraries are included.

## 4. Structure of a Spring Boot Project

When creating a project with Spring Boot, the project structure follows a standard format that facilitates development, maintenance, and deployment. The project's structure is organized to allow a clear separation between configuration, source code, and resources.

### 1. Directory and File Structure

1.1. Project Root Directory

The root directory of the project contains important files and directories that configure and manage the project.

- `src/`: Contains the source code and resources of the application.
    - `main/`: Main source code and resources.
        - `java/`: Contains the source code in Java.
            - `com/example/demo/`: Base package of the project (modifiable based on group and artifact).
                - `DemoApplication.java`: Main class that starts the Spring Boot application. This class has the `@SpringBootApplication` annotation, indicating it is a Spring Boot application and enabling various features.
        - `resources/`: Contains static and configuration resources.
            - `application.properties`: Main configuration file for setting application properties.
            - `static/`: Static resources such as CSS, JavaScript, and images.
            - `templates/`: Templates for template engines (e.g., Thymeleaf).
    - `test/`: Test source code.
        - `java/`: Contains unit and integration tests.

        - `resources/`: Resources used during tests.
- `pom.xml` (for Maven) or `build.gradle` (for Gradle): Build configuration files that define dependencies, plugins, and project configurations.
- `README.md`: Optional file providing a description of the project and how to use it.
- `Dockerfile`: Optional file for creating a Docker image of the project (if Docker is used).

**2. Key Files and Directories**

2.1. `src/main/java/com/example/demo/DemoApplication.java`

- Description: The main class of the application.
- Function: Contains the `main` method that uses `SpringApplication.run()` to start the Spring Boot application.

```
package com.example.demo;


import org.springframework.boot.SpringApplication;

import
org.springframework.boot.autoconfigure.SpringBootApplication;


@SpringBootApplication

public class DemoApplication {

    public static void main(String[] args) {

        SpringApplication.run(DemoApplication.class, args);

    }

}
```

2.2. `src/main/resources/application.properties`

- Description: Configuration file where application properties are defined, such as database configuration, server port, and other parameters.
- Example Configuration:

```
# Server port configuration

server.port=8080
```

```
# Database configuration

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.driver-class-name=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=password
```

- `spring.jpa.database-platform=org.hibernate.dialect.H2Dialect`
- 2.3. `src/main/resources/static/`
  - Description: Contains static files such as CSS, JavaScript, and images that are served directly by the server.
  - Example: `style.css`, `script.js`, `logo.png`
- 2.4. `src/main/resources/templates/`
  - Description: Contains template files for template engines, such as Thymeleaf, used to generate dynamic content in web views.
  - Example: `index.html`
- 2.5. `src/test/java/com/example/demo/`
  - Description: Contains unit and integration tests for the project. Tests are organized in packages that reflect the structure of the main source code packages.
  - Example: `DemoApplicationTests.java`

```java
package com.example.demo;


import org.junit.jupiter.api.Test;

import org.springframework.boot.test.context.SpringBootTest;


@SpringBootTest

class DemoApplicationTests {


    @Test

    void contextLoads() {

    }
}
```

## 4. Structure of a Spring Boot Project

### 4.1. Conventions and Practices

Package Naming

- Domain-Based Structure: Use a package structure that follows the format of the reverse of the company's domain name. For example, if the company domain is `example.com`, the base package should be `com.example`. This convention helps avoid name conflicts with other libraries and applications.
  - Example: `com.example.myapp`
- Functional Packages: Within the base package, organize packages by functionality or module. For example, you might have packages for controllers, services, repositories, etc.
  - Example:
    - `com.example.myapp.controller`
    - `com.example.myapp.service`
    - `com.example.myapp.repository`
    - `com.example.myapp.model`
- Test Packages: Organize tests into packages that reflect the structure of the main source code packages. This makes it easier to locate tests related to a specific component.
  - Example:
    - `com.example.myapp.controller` (for tests of controllers)
    - `com.example.myapp.service` (for tests of services)
    - `com.example.myapp.repository` (for tests of repositories)
    - `com.example.myapp.model` (for tests of models)
- Package Naming: Use lowercase names for packages and avoid using underscores (`_`). Names should be descriptive and clearly represent the functionality of the package.
  - Example: `com.example.myapp.service.user`

Class Naming

- Class Names: Use nouns or noun phrases that clearly describe the class's functionality or purpose. Use PascalCase (also known as UpperCamelCase) for class names.
  - Example: `UserController`, `OrderService`, `ProductRepository`
- Controllers: Classes that handle HTTP requests and serve as entry points for APIs should be named with the suffix `Controller`.
  - Example: `UserController`, `OrderController`
- Services: Classes containing business logic should be named with the suffix `Service`.
  - Example: `UserService`, `OrderService`

- Repositories: Classes interacting with the database and performing CRUD operations should be named with the suffix `Repository`.
  - Example: `UserRepository`, `OrderRepository`
- Models (Entities): Classes representing domain entities or data should be named with a singular noun and follow PascalCase.
  - Example: `User`, `Order`

Method and Variable Naming

- Methods: Use verbs or verb phrases for method names and follow camelCase (also known as lowerCamelCase). Names should clearly describe the action performed by the method.
  - Example: `getUserById()`, `createOrder()`, `updateProduct()`
- Variables: Use descriptive names and follow camelCase for variables. Names should clearly indicate the purpose of the variable.
  - Example: `userList`, `orderAmount`, `productName`
- Constants: Constants should be in uppercase with words separated by underscores (`_`).
  - Example: `MAX_LOGIN_ATTEMPTS`, `DEFAULT_PAGE_SIZE`

Configuration Files

- Property Files: Use descriptive names for configuration files and follow the format `application.properties` or `application.yml` for the main configuration.
  - Example: `application.properties`, `application-dev.properties`, `application-prod.yml`
- Profile-Specific Configuration Files: For profile-specific configurations, use filenames that include the profile name.
  - Example: `application-dev.properties`, `application-test.properties`

Documentation and Comments

- Code Comments: Use clear and concise comments to explain complex logic and design decisions. Comments should be relevant and helpful for other developers reading the code.
  - Example:

```
// Calculate the total amount after applying discounts

public double calculateTotalAmount(Order order) {

    // implementation

}
```

- Class and Method Documentation: Use Javadoc to document classes and methods. Include descriptions of functionality, parameters, and return values.

  - Example:

```
/**
 * Creates a new user with the specified details.
 * @param userDTO The user details.
 * @return The created user.
 */
public User createUser(UserDTO userDTO) {
    // implementation
}
```

These conventions and practices help keep the code organized, readable, and maintainable, which is especially important in collaborative and long-term projects.

## 5. Basic Controllers (GET, POST)

Controllers in Spring Boot handle HTTP requests and map them to service methods or return responses. They are a crucial part of building web applications and APIs.

### 1. Introduction to Controllers

Controllers are responsible for processing incoming HTTP requests, interacting with the service layer, and returning responses. They are annotated with `@Controller` or `@RestController`.

- `@Controller`: Indicates that the class serves as a web controller. It is often used in combination with view templates.
- `@RestController`: A specialized version of `@Controller` that combines `@Controller` and `@ResponseBody`. It is used for RESTful web services where responses are typically JSON or XML.

### 2. Creating Basic Controllers

2.1. GET Requests

The `@GetMapping` annotation is used to map HTTP GET requests to a method. It is used for fetching data or displaying information.

Example:

```
package com.example.myapp.controller;
```

```
import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;


@RestController

@RequestMapping("/api")

public class MyController {


    @GetMapping("/greeting")

    public String getGreeting() {

        return "Hello, World!";

    }

}
```

- `@RequestMapping("/api")`: Defines the base URL for all methods in the controller.
- `@GetMapping("/greeting")`: Maps GET requests to `/api/greeting` to the `getGreeting` method.
- `getGreeting`: Returns a simple greeting message.

### 2.2. POST Requests

The `@PostMapping` annotation is used to map HTTP POST requests to a method. It is typically used for creating or submitting data.

Example:

```
package com.example.myapp.controller;


import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;
```

```java
@RestController
@RequestMapping("/api")
public class MyController {

    @PostMapping("/submit")
    public String submitData(@RequestBody String data) {
        // Process the submitted data
        return "Data received: " + data;
    }
}
```

- `@PostMapping("/submit")`: Maps POST requests to `/api/submit` to the `submitData` method.
- `@RequestBody`: Binds the request body to the `data` parameter.
- `submitData`: Receives data from the request body and returns a confirmation message.

**3. Handling Path Variables and Request Parameters**

3.1. Path Variables

Path variables are used to pass data in the URL. They are specified using curly braces `{}` in the URL pattern and accessed via method parameters.

Example:

```java
package com.example.myapp.controller;

import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
```

```
@RestController
@RequestMapping("/api")
public class MyController {

    @GetMapping("/user/{id}")
    public String getUserById(@PathVariable("id") Long id) {
        // Fetch user by ID
        return "User ID: " + id;
    }
}
```

- `@PathVariable("id")`: Binds the path variable `id` from the URL to the method parameter.

3.2. Request Parameters

Request parameters are used to pass data in the query string of the URL. They are accessed using `@RequestParam`.

Example:

```
package com.example.myapp.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api")
public class MyController {
    @GetMapping("/search")
    public String search(@RequestParam(name = "query",
defaultValue = "") String query) {
        // Perform search with the query parameter
        return "Search results for: " + query;
    }
}
```

- `@RequestParam(name = "query")`: Binds the query parameter `query` from the URL to the method parameter. The `defaultValue` attribute provides a default value if the parameter is not present.

**4. Error Handling in Controllers**

To handle errors gracefully, you can use `@ExceptionHandler` methods within your controllers.

Example:

```
package com.example.myapp.controller;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api")
public class MyController {
    @GetMapping("/error")

    public String triggerError() {

        throw new RuntimeException("Something went wrong!");

    }


    @ExceptionHandler(RuntimeException.class)

    public ResponseEntity<String>
handleRuntimeException(RuntimeException ex) {

        return new ResponseEntity<>(ex.getMessage(),
HttpStatus.INTERNAL_SERVER_ERROR);

    }

}
```

- `@ExceptionHandler(RuntimeException.class)`: Handles `RuntimeException` and returns an appropriate HTTP status and message.