### Dependency Injection in Spring Boot

\*\*Dependency Injection (DI)\*\* is a fundamental design pattern in Spring, which is part of a broader concept known as \*\*Inversion of Control (IoC)\*\*. The core idea is that objects do not create their dependencies themselves. Instead, dependencies are provided (injected) by the Spring IoC container, allowing for loose coupling between components and better testability.

In Spring Boot, DI is achieved by annotating fields, constructors, or setter methods with \*\*@Autowired\*\*, and Spring Boot takes care of injecting the dependencies at runtime.

---

### Types of Dependency Injection in Spring Boot

1. \*\*Constructor Injection\*\* (Recommended)

2. \*\*Setter Injection\*\*

3. \*\*Field Injection\*\*

---

### 1. Constructor Injection (Recommended)

With \*\*constructor injection\*\*, Spring Boot injects dependencies via the class's constructor. This is the most recommended approach as it ensures that the class is always in a valid state (dependencies are provided at creation) and makes the class easier to test.

#### Example:

```java

import org.springframework.stereotype.Service;

@Service

public class MyService {

private final MyRepository myRepository;

// Constructor-based injection

public MyService(MyRepository myRepository) {

this.myRepository = myRepository;

}

public void performAction() {

myRepository.doSomething();

}

}

```

Here:

- \*\*`MyService`\*\* depends on \*\*`MyRepository`\*\*.

- The dependency is injected via the constructor.

- The \*\*`@Autowired`\*\* annotation is optional in constructor-based injection from Spring 4.3 onwards if the class has only one constructor.

---

### 2. Setter Injection

With \*\*setter injection\*\*, Spring Boot injects dependencies via setter methods. This approach allows dependencies to be injected after the object is created, which makes it possible to change dependencies after the object is constructed.

#### Example:

```java

import org.springframework.stereotype.Service;

@Service

public class MyService {

private MyRepository myRepository;

// Setter-based injection

@Autowired

public void setMyRepository(MyRepository myRepository) {

this.myRepository = myRepository;

}

public void performAction() {

myRepository.doSomething();

}

}

```

In this example:

- The \*\*`MyRepository`\*\* dependency is injected using a setter method.

- The \*\*`@Autowired`\*\* annotation is required on the setter method to tell Spring to inject the dependency.

---

### 3. Field Injection

With \*\*field injection\*\*, Spring Boot directly injects dependencies into fields. While this approach is convenient, it is generally discouraged for several reasons, such as reduced testability and lack of immutability.

#### Example:

```java

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

@Service

public class MyService {

@Autowired

private MyRepository myRepository;

public void performAction() {

myRepository.doSomething();

}

}

```

In this example:

- The \*\*`@Autowired`\*\* annotation on the field tells Spring to inject \*\*`MyRepository`\*\* directly into the \*\*`MyService`\*\* class.

- Field injection is not recommended because it can make unit testing difficult and the service might be in an invalid state if the dependency is not injected correctly.

---

### Key Concepts of Dependency Injection in Spring Boot

1. \*\*Inversion of Control (IoC)\*\*:

- The control of object creation is given to the Spring IoC container, which manages the lifecycle of objects (beans).

2. \*\*Spring Beans\*\*:

- Any class annotated with \*\*`@Component`\*\*, \*\*`@Service`\*\*, \*\*`@Repository`\*\*, or \*\*`@Controller`\*\* becomes a Spring-managed bean. Spring automatically wires these beans into other components that depend on them.

3. \*\*@Autowired\*\*:

- This annotation is used to mark fields, constructors, or setters to tell Spring that the dependency should be injected by the IoC container.

4. \*\*@Component, @Service, @Repository\*\*:

- These annotations are used to define Spring beans. Once these annotations are added, Spring manages these classes and allows them to be injected into other components using DI.

---

### Example of Dependency Injection in a Spring Boot Project

#### Step 1: Define a Repository Class

```java

import org.springframework.stereotype.Repository;

@Repository

public class MyRepository {

public void doSomething() {

System.out.println("Repository action performed.");

}

}

```

- \*\*`@Repository`\*\* marks this class as a bean to be managed by Spring and indicates that it is responsible for data access logic.

#### Step 2: Define a Service Class that Uses the Repository

```java

import org.springframework.stereotype.Service;

@Service

public class MyService {

private final MyRepository myRepository;

// Constructor-based injection

public MyService(MyRepository myRepository) {

this.myRepository = myRepository;

}

public void performAction() {

System.out.println("Service action performed.");

myRepository.doSomething();

}

}

```

- \*\*`@Service`\*\* indicates that this class contains business logic and is also managed by Spring.

- The repository is injected via the constructor.

#### Step 3: Define a Controller Class

```java

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

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

@RestController

public class MyController {

private final MyService myService;

// Constructor-based injection

public MyController(MyService myService) {

this.myService = myService;

}

@GetMapping("/action")

public String action() {

myService.performAction();

return "Action completed!";

}

}

```

- \*\*`@RestController`\*\* marks this class as a Spring-managed controller that handles HTTP requests.

- The service is injected into the controller using constructor injection.

#### Step 4: Running the Application

```java

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);

}

}

```

Now, when you run the application and access `http://localhost:8080/action` in a browser, the following output will be printed in the console:

```

Service action performed.

Repository action performed.

```

And the browser will display:

```

Action completed!

```

---

### Benefits of Dependency Injection in Spring Boot

1. \*\*Loose Coupling\*\*:

- DI promotes loose coupling between objects, allowing you to focus on developing individual components independently.

2. \*\*Easier Testing\*\*:

- DI makes it easier to test classes by allowing you to inject mock dependencies, making unit testing more effective.

3. \*\*Centralized Object Creation\*\*:

- The Spring IoC container centrally manages the lifecycle of objects, reducing boilerplate code and promoting reusable code.

4. \*\*Improved Code Maintainability\*\*:

- The loose coupling between components improves code maintainability and flexibility, as changes in one component do not affect others.

---

### Conclusion

Dependency Injection in Spring Boot simplifies how components in your application interact with each other. By letting the Spring IoC container manage the creation and wiring of objects, your code becomes cleaner, more maintainable, and easier to test. Whether you use constructor injection, setter injection, or field injection, Spring Boot’s DI mechanism helps build modular, testable, and scalable applications efficiently.