Spring Configuration

What is Spring Configuration?

Spring is a framework used to build Java applications, and configuration in Spring is a way to set up and customize how your application behaves. It's essentially telling Spring how you want your application to work and how it should manage things like your objects (beans), databases, etc.

There are **three types of configuration** in Spring:

1. **XML Configuration**
2. **Annotation-based Configuration**
3. **Java-based Configuration (Java Config)**

Why are we using configuration in Spring?

Configuration in Spring is needed to set up the **beans** (objects or components) that Spring manages for you. It helps Spring understand how to create, wire, and manage these beans so that your application runs as expected. Instead of manually creating and managing objects yourself, Spring does it for you in a more organized way.

Why Do We Use Beans in Spring?

In Spring Framework, a **bean** is an object that is managed by the **Spring IoC (Inversion of Control) container**. These beans form the backbone of a Spring application, as they are the objects that Spring instantiates, configures, and manages. Using beans has several advantages, and understanding why they are needed helps grasp the fundamentals of Spring.

Key Reasons for Using Beans in Spring

1. Centralized Object Management

- **Without Spring:** You manually create and manage objects in your application code.
- **With Spring Beans:** Spring takes responsibility for creating and managing the lifecycle of objects, simplifying development.
- **Benefit:** Centralized management allows consistent behavior and efficient resource handling.

Example Without Spring:

```
java

GreetingService greetingService = new GreetingService();
```

With Spring:

```
java

@Component
public class GreetingService { }
// Spring container creates and manages the GreetingService bean.
```

---

2. Dependency Injection (DI)

- Beans enable Spring to perform **DI**, which is a design pattern where dependencies are provided to an object instead of the object creating them itself.
- **Benefit:** This reduces tight coupling between objects and promotes a modular, maintainable, and testable application design.

Example:

```
java

@Component
public class Person {
    @Autowired
    private GreetingService greetingService;  // Dependency injected by
Spring
}
```

---

3. Lifecycle Management

- Spring beans have a **lifecycle**: creation, initialization, and destruction.
- You can define custom initialization and destruction logic using:
  - XML configuration: `<init-method>` and `<destroy-method>`.
  - Annotations: `@PostConstruct` and `@PreDestroy`.
- **Benefit:** You can control how beans are created, initialized, and destroyed.

---

4. Singleton and Other Scopes

- Beans in Spring can be created with different **scopes**, such as:
  - **Singleton** (default): Only one instance of the bean is created and shared.
  - **Prototype**: A new instance is created each time the bean is requested.
  - Other scopes like `request`, `session`, etc., for web applications.
- **Benefit:** Scope flexibility ensures optimal resource usage.

Example (Singleton):

```
java

@Component
public class GreetingService {
    // Singleton: Shared across the application.
```

```
}
```

---

5. Loose Coupling

- Beans reduce **tight coupling** between components by relying on interfaces and DI rather than directly instantiating objects.
- **Benefit:** Loose coupling ensures easier modifications, testing, and scaling of applications.

Example of Tight Coupling:

```
java

public class Person {
    private GreetingService greetingService = new GreetingService();
}
```

Loose Coupling with Beans:

```
java

@Component
public class Person {
    @Autowired
    private GreetingService greetingService;
}
```

---

6. Configuration Flexibility

- Beans can be defined in multiple ways:
  - **XML-based Configuration**
  - **Annotation-based Configuration**
  - **Java-based Configuration**
- **Benefit:** You can choose the configuration style that suits your project or team preferences.

---

7. AOP (Aspect-Oriented Programming) Support

- Beans in Spring can be enhanced using **AOP** to add cross-cutting concerns like logging, security, or transaction management without modifying the core logic.
- Benefit: Keeps business logic clean and separates concerns effectively.

1. Using XML for Dependency Injection

Concept:

- In XML configuration, you define the beans and their dependencies in an XML file. Spring reads this file and knows how to inject dependencies.

Example: XML Configuration for DI

Imagine we have two classes:

`GreetingService`:

```java
public class GreetingService {
    public String getGreeting()
{       return "Hello";
    }
}
```

`Person`:

```java
public class Person {
    private GreetingService greetingService;

    // Setter method for Dependency Injection
    public void setGreetingService(GreetingService greetingService) {
        this.greetingService = greetingService;
    }

    public void greet() {
        System.out.println(greetingService.getGreeting() + ", John!");
    }
}
```

Here:

- The `Person` class depends on `GreetingService`.
- Instead of creating the `GreetingService` object manually in `Person`, we'll use Spring to inject it.

XML Configuration:

The XML file (`applicationContext.xml`) defines the beans and their relationships:

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- Define the GreetingService bean -->
    <bean id="greetingService" class="com.example.GreetingService"/>

    <!-- Define the Person bean and inject the dependency -->
    <bean id="person" class="com.example.Person">
        <property name="greetingService" ref="greetingService"/>
    </bean>

</beans>
```

Explanation of XML:

1. The `<bean>` tag defines a bean (an object managed by Spring).
    - The `id` is the bean name.
    - The `class` is the fully qualified class name of the object to create.
2. The `<property>` tag specifies which property of the class to inject, using the `name` attribute (matches the setter method name).
3. The `ref` attribute specifies the bean to inject (`greetingService`).

Main Method to Test:

```java
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {
    public static void main(String[] args) {
        // Load the XML configuration file
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

        // Get the Person bean
        Person person = context.getBean("person", Person.class);

        // Call the greet method
        person.greet();
    }
}
```

Output:

```
Hello, John!
```

How Dependency Injection Happens in XML:

1. Spring reads the `applicationContext.xml` file.
2. It creates the `GreetingService` bean.
3. It creates the `Person` bean and **injects the `GreetingService` bean** into it using the setter method (`setGreetingService()`).
4. When you call `person.greet()`, the `Person` object already has the `GreetingService` injected.

---

2. Using Annotations for Dependency Injection

Concept:

- In annotation-based configuration, you use annotations like `@Component`, `@Autowired`, and `@Configuration` in your Java classes.
- This approach eliminates the need for XML files.

1. @Component

- **What it does**: It tells Spring, "This class is something you need to manage (create an object for it and handle it)."
- **Where we use it**: On classes that should be treated as beans by Spring.
- **Why we use it**: To make the class available for dependency injection.

Example:

```
java
```

```
@Component
public class GreetingService {
    public String sayHello() {
        return "Hello!";
    }
}
```

---

2. @Autowired

- **What it does**: It tells Spring, "Put the object I need here automatically."
- **Where we use it**: On fields, constructors, or setter methods.
- **Why we use it**: To connect one bean to another without writing `new Object()` manually.

Example (Field Injection):

```
java

@Component
public class Person {

    @Autowired
    private GreetingService greetingService; // Spring puts the
GreetingService object here.

    public void greet() {
        System.out.println(greetingService.sayHello());
    }
}
```

---

3. @Configuration

- **What it does**: It tells Spring, "This class has the instructions to create beans (objects)."
- **Where we use it**: On a class that provides methods to define beans.
- **Why we use it**: To replace XML configuration with Java code.

Example:

```
java

@Configuration
public class AppConfig {
    @Bean
    public GreetingService greetingService() {
        return new GreetingService(); // This method tells Spring how to
create a GreetingService object.
    }
}
```

---

4. @Bean

- **What it does**: It tells Spring, "This method provides the object you need to manage."
- **Where we use it**: Inside a `@Configuration` class.
- **Why we use it**: To manually define a bean when annotations like `@Component` are not used.

Example:

```
Java

@Bean
public Person person(GreetingService greetingService) {
    return new Person(greetingService); // Spring will call this method and
use its result as a bean.
}
```

5. @Qualifier

- **What it does**: It tells Spring, "Use this specific bean when there are multiple options."
- **Where we use it**: On fields, constructors, or setter methods.
- **Why we use it**: To choose between beans with the same type.

Example:

```
java

@Component("morningService")
public class MorningGreetingService implements GreetingService { }

@Component("eveningService")
public class EveningGreetingService implements GreetingService { }

@Component
public class Person {
    @Autowired
    @Qualifier("morningService")
    private GreetingService greetingService;
}
```

Example: Annotation Configuration for DI

The same `GreetingService` and `Person` classes, but now using annotations.

`GreetingService`:

```java
import org.springframework.stereotype.Component;

@Component  // Marks this class as a Spring-managed bean
public class GreetingService {
    public String getGreeting() {
        return "Hello";
    }
}
```

Person:

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component  // Marks this class as a Spring-managed bean
public class Person {
    private GreetingService greetingService;

    @Autowired  // Tells Spring to inject the GreetingService bean here
    public void setGreetingService(GreetingService greetingService) {
        this.greetingService = greetingService;
    }

    public void greet() {
        System.out.println(greetingService.getGreeting() + ", John!");
    }
}
```

Configuration Class:

We don’t need an XML file. Instead, we use a Java class to configure Spring:

```java
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan(basePackages = "com.example")  // Scans for @Component classes
public class AppConfig {
}
```

Main Method to Test:

```java
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Main {
    public static void main(String[] args) {
        // Load the Java-based configuration
        ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

        // Get the Person bean
        Person person = context.getBean(Person.class);

        // Call the greet method
        person.greet();
    }
}
```

Output:

```
Hello, John!
```

How Dependency Injection Happens with Annotations:

1. Spring scans the `com.example` package for classes annotated with `@Component`.
2. It finds `GreetingService` and `Person` and creates beans for them.
3. The `@Autowired` annotation on the `setGreetingService()` method tells Spring to inject the `GreetingService` bean into the `Person` bean.
4. When you call `person.greet()`, the `Person` object already has the `GreetingService` injected.

3. Java-based Configuration (Java Config)

This is a variant of annotation-based configuration. Instead of using annotations like `@ComponentScan`, you write Java code in a class and define methods that return beans. You're still writing Java code, but now Spring can manage your beans and dependencies for you.

Step 1: Modify the `Person` Class to Have a Dependency

Let's introduce a `GreetingService` that will handle greeting logic.

```java
package com.example;

public class Person {
    private String name;
    private GreetingService greetingService;

    // Constructor that injects the GreetingService
    public Person(String name, GreetingService greetingService) {
        this.name = name;
        this.greetingService = greetingService;
    }

    // Greet method now uses the GreetingService
    public String greet() {
        return greetingService.getGreeting() + ", " + name + "!";
    }
}
```

Here, the `Person` class now depends on `GreetingService` to provide the greeting logic. We pass the `GreetingService` through the constructor to **inject** the dependency.

Step 2: Create the `GreetingService` Class

Now, let's create a simple `GreetingService` class.

```java
package com.example;

public class GreetingService {

    public String getGreeting()
{        return "Hello";
    }
}
```

Step 3: Modify the `AppConfig` Class to Configure the Dependencies

We need to modify the Spring configuration class (`AppConfig`) to tell Spring how to inject the `GreetingService` into the `Person` object.

Now, Spring will:

1. **Create a `GreetingService` bean** because of the `@Bean` method `greetingService()`.

```java
package com.example;

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

@Configuration
public class AppConfig {

    @Bean
    public GreetingService greetingService() {
        return new GreetingService(); // Spring will manage this object
    }

    @Bean
    public Person person() {
        // Spring injects the GreetingService bean into the Person constructor
        return new Person("John", greetingService());
    }
}
```

2. **Inject the `GreetingService` bean** into the `Person` constructor. This is **Dependency Injection**—Spring automatically provides the required `GreetingService` object when creating the `Person` bean.

Step 4: Modify the `MainApp` Class

The `MainApp` class doesn't need to change; it will still retrieve the `Person` bean from the Spring context and call the `greet()` method.

```java
package com.example;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class MainApp {

    public static void main(String[] args) {
        // Create the Spring container using the configuration
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

        // Retrieve the Person bean (with GreetingService injected)
        Person person = context.getBean(Person.class);

        // Call the greet method, which uses GreetingService
        System.out.println(person.greet());

        // Close the context
        context.close();
    }
}
```

Where is Dependency Injection Happening?

Let's go step by step with debugging to understand **exactly** where Dependency Injection happens:

1. **Creating `AnnotationConfigApplicationContext`:**
   - Spring reads the configuration from `AppConfig.class`.
   - Spring finds the `@Bean` methods: `greetingService()` and `person()`.
2. **Creating `GreetingService` Bean:**
   - Spring calls the `greetingService()` method and creates an instance of `GreetingService`.
3. **Creating `Person` Bean with Dependency Injection:**
   - Spring calls the `person()` method.
   - Spring sees that `Person` has a constructor that requires a `GreetingService` instance.
   - **This is where Dependency Injection happens**: Spring already knows how to create the `GreetingService` bean (from step 2), so it **injects** the `GreetingService` into the `Person` constructor.
4. **Person Bean is Fully Constructed:**
   - Spring creates the `Person` object with the name "John" and the `GreetingService` object that was injected.
5. **Using the `greet()` Method:**
   - When you call `person.greet()`, Spring has already injected the `GreetingService` into `Person`. The `greet()` method uses this service to generate a greeting.

Output

The output will now be:

```
Hello, John!
```

Recap of Dependency Injection in the Example

- **`GreetingService` is injected into `Person`** via the constructor.
- **Spring is responsible for creating and injecting** the `GreetingService` bean into the `Person` bean. This is done automatically by Spring, and you don't need to manually create or pass the `GreetingService` when creating a `Person`.

Summary

- **Dependency Injection** is the process where Spring injects the required dependencies (like `GreetingService`) into a class (like `Person`).
- In this case, the `Person` class **depends on** the `GreetingService`, and Spring **injects** it when creating the `Person` bean.
- The key part where DI happens is when Spring calls the `person()` method in `AppConfig` and injects the `GreetingService` into the `Person` constructor.

Comparison Table

Feature	XML Configuration	Annotation-based Configuration	Java-based Configuration
Setup	Uses XML files to define beans and dependencies.	Uses annotations like `@Component` and `@Autowired`.	Uses Java classes and `@Bean` methods.
Ease of Use	Verbose and harder to read.	Cleaner, less boilerplate than XML.	Cleaner, with full control using Java.
Flexibility	Good for externalized configuration.	Tightly coupled to Java code.	Fully Java-based and flexible.
Readability	XML is harder to maintain in large projects.	Code is easier to read and understand.	Highly readable and maintainable.
Dependencies	Requires XML parsers and files.	Requires annotations in the source code.	Requires no external files, only Java.
Use Case	Useful for legacy or external configurations.	Good for new, annotation-friendly projects.	Best for modern, pure Java projects.