Spring Core Concepts

Inversion of Control (IoC)

Inversion of Control is a design principle in which the control of objects or the flow of a program is inverted. Instead of the developer manually creating objects or managing their dependencies, an external entity (like a framework) takes responsibility for managing these.

Example Without IoC (Tightly Coupled):

```
class DatabaseService {
    public void connect() {
        System.out.println("Database connected!");
    }
}

class Application {
    private DatabaseService databaseService;

    public Application() {
        databaseService = new DatabaseService(); // Tightly co
upled
    }

    public void start() {
        databaseService.connect();
    }
}

public class Main {
    public static void main(String[] args) {
        Application app = new Application();
        app.start();
    }
}
```

Here:

- `Application` class directly creates an instance of `DatabaseService`, making it tightly coupled.
- Any changes to `DatabaseService` require changes in the `Application` class.

Dependency Injection (DI)

Dependency Injection is a specific implementation of IoC. It provides the required dependencies to an object instead of the object creating them itself. DI can be done through:

Types of DI

1. **Constructor Injection**
2. **Setter Injection**
3. **Field Injection**

---

1. Constructor Injection

In **Constructor Injection**, the dependency is provided through the class constructor.

Example:

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

// Dependency
@Component
class DatabaseService {
    public void connect() {
        System.out.println("Database connected via Constructor
Injection!");
    }
}

// Dependent class
@Component
```

```
class Application {
    private final DatabaseService databaseService;

    // Constructor Injection
    @Autowired
    public Application(DatabaseService databaseService) {
        this.databaseService = databaseService;
    }

    public void start() {
        databaseService.connect();
    }
}

// Main Class
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplic
ation;

@SpringBootApplication
public class Main {
    public static void main(String[] args) {
        var context = SpringApplication.run(Main.class, args);
        var app = context.getBean(Application.class);
        app.start();
    }
}
```

If you use xml configurations:

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

    <!-- Bean for DatabaseService -->
    <bean id="databaseService" class="com.example.DatabaseService
```

```
    <!-- Bean for Application with constructor injection -->
    <bean id="application" class="com.example.Application">
        <constructor-arg ref="databaseService"/>
    </bean>
</beans>
```

2. Setter Injection

In **Setter Injection**, the dependency is provided through a setter method.

Example:

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

// Dependency
@Component
class DatabaseService {
    public void connect() {
        System.out.println("Database connected via Setter Inje
ction!");
    }
}

// Dependent class
@Component
class Application {
    private DatabaseService databaseService;

    // Setter Injection
    @Autowired
    public void setDatabaseService(DatabaseService databaseSer
vice) {
        this.databaseService = databaseService;
    }

    public void start() {
        databaseService.connect();
    }
}
```

```
// Main Class
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplic
ation;

@SpringBootApplication
public class Main {
    public static void main(String[] args) {
        var context = SpringApplication.run(Main.class, args);
        var app = context.getBean(Application.class);
        app.start();
    }
}
```

If you use xml configuraitons:

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

    <!-- Bean for DatabaseService -->
    <bean id="databaseService" class="com.example.DatabaseService

    <!-- Bean for Application with setter injection -->
    <bean id="application" class="com.example.Application">
        <property name="databaseService" ref="databaseService"/>
    </bean>
</beans>
```

3. Field Injection

In **Field Injection**, the dependency is directly injected into the class field using the `@Autowired` annotation.

Example:

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

// Dependency
@Component
class DatabaseService {
    public void connect() {
        System.out.println("Database connected via Field Injec
tion!");
    }
}

// Dependent class
@Component
class Application {
    @Autowired
    private DatabaseService databaseService; // Field Injectio
n

    public void start() {
        databaseService.connect();
    }
}

// Main Class
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplic
ation;

@SpringBootApplication
public class Main {
    public static void main(String[] args) {
        var context = SpringApplication.run(Main.class, args);
        var app = context.getBean(Application.class);
        app.start();
    }
}
```

Note: Field Injection is not directly supported in XML configuration. You must use either **constructor** or **setter injection** in XML.

Comparison of Injection Types

Type	Pros	Cons
Constructor Injection	Ensures all dependencies are provided at object creation. Suitable for mandatory dependencies.	Increases verbosity when many dependencies are required.
Setter Injection	Flexible for optional dependencies.	Risk of incomplete initialization if dependencies are not set.
Field Injection	Simple and concise.	Difficult to test and violates immutability principles.

Best Practices

- Use **Constructor Injection** for mandatory dependencies (preferred in most cases).
- Use **Setter Injection** for optional dependencies.
- Avoid **Field Injection** when writing tests or creating immutable classes, as it tightly couples your code to the framework.

Key Points

1. **XML Configuration** is less popular now but is still useful for legacy projects.
2. It is more verbose compared to annotations or Java-based configuration.
3. Recommended for projects that need to define dependencies outside the codebase for better decoupling.

Steps to Configure Package Scanning in XML

1. XML Configuration

Add the following to your `beans.xml` file:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/co
ntext"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans http://
www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/context htt
p://www.springframework.org/schema/context/spring-context.xs
d">

    <!-- Enable component scanning -->
    <context:component-scan base-package="com.example"/>
</beans>
```

Here:

- `base-package="com.example"` specifies the package to scan for annotated components.
- Spring will scan the package `com.example` and all its sub-packages.

---

2. Annotated Classes

Annotate your classes with the appropriate Spring annotations:

- `@Component`: Generic stereotype for components.
- `@Service`: Stereotype for service-layer components.
- `@Repository`: Stereotype for DAO-layer components.
- `@Controller`: Stereotype for Spring MVC controllers.

Example:

```
package com.example.service;

import org.springframework.stereotype.Component;

@Component
```

```
public class DatabaseService {
    public void connect() {
        System.out.println("Database connected!");
    }
}
```

```
package com.example;

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

@Component
public class Application {
    private final DatabaseService databaseService;

    @Autowired
    public Application(DatabaseService databaseService) {
        this.databaseService = databaseService;
    }

    public void start() {
        databaseService.connect();
    }
}
```

3. Main Class

Load the application context from the XML configuration and use the beans:

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplica
tionContext;

public class Main {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicati
```

```
onContext("beans.xml");
        Application app = context.getBean(Application.class);
        app.start();
    }
}
```

Key Points

1. **Base Package**:
   - The `base-package` attribute in `<context:component-scan>` specifies the root package to scan. Spring will scan all sub-packages recursively.
2. **Multiple Packages**:
   - If you want to scan multiple packages, you can use a comma-separated list:

```
<context:component-scan base-package="com.example,com.other"/>
```

3. **Filter Components**:
   - Use the `<context:include-filter>` or `<context:exclude-filter>` to include or exclude specific components during scanning.

     Example:

```
<context:component-scan base-package="com.example">
    <context:exclude-filter type="annotation" expression="org.springframework.stereotype.Repository"/>
</context:component-scan>
```

Bean Lifecycle

The **Spring Bean Lifecycle** describes the process that a Spring-managed bean goes through from creation to destruction. Spring provides a powerful way to manage this lifecycle with hooks and callback methods.

Lifecycle Phases

1. **Bean Instantiation**:

   The container creates an instance of the bean using the no-argument constructor or a static factory method.

2. **Populate Properties**:

   Spring injects dependencies into the bean, either via constructor, setter methods, or field injection.

3. **Bean Name and Factory Awareness (Optional)**:

   If the bean implements any of the `Aware` interfaces, Spring provides additional context like the bean name or the application context.

4. **Pre-Initialization (BeanPostProcessor)**:

   Before the initialization callbacks, `BeanPostProcessor` methods are invoked for any custom processing.

5. **Initialization**:

   The container calls lifecycle callbacks like `@PostConstruct` or `InitializingBean`'s `afterPropertiesSet()` method.

6. **Post-Initialization (BeanPostProcessor)**:

   After initialization, another set of `BeanPostProcessor` methods are invoked.

7. **Ready for Use**:

   The bean is now fully initialized and available for use.

8. **Destruction**:

   When the application shuts down, the container destroys the bean, calling methods annotated with `@PreDestroy` or `DisposableBean`'s `destroy()` method.

Detailed Lifecycle Steps with Examples

1. Implementing Bean Lifecycle Methods

Here is an example demonstrating the entire lifecycle:

XML Configuration

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

    <!-- Bean declaration -->
    <bean id="exampleBean" class="com.example.LifecycleBean" i
nit-method="customInit" destroy-method="customDestroy"/>
</beans>
```

Java Class

```
package com.example;

import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

public class LifecycleBean implements InitializingBean, Dispos
ableBean {

    public LifecycleBean() {
        System.out.println("1. Bean Instantiation");
    }

    // Dependency injection happens here (if any)

    @PostConstruct
    public void postConstruct() {
        System.out.println("2. @PostConstruct - Called after d
ependencies are injected.");
```

```
    }

    @Override
    public void afterPropertiesSet() {
        System.out.println("3. InitializingBean's afterPropert
iesSet() - Custom initialization logic here.");
    }

    public void customInit() {
        System.out.println("4. Custom init-method - Declared i
n XML or Java config.");
    }

    @PreDestroy
    public void preDestroy() {
        System.out.println("5. @PreDestroy - Cleanup before de
struction.");
    }

    @Override
    public void destroy() {
        System.out.println("6. DisposableBean's destroy() - Ad
ditional cleanup logic here.");
    }

    public void customDestroy() {
        System.out.println("7. Custom destroy-method - Declare
d in XML or Java config.");
    }
}
```

Key Interfaces and Annotations

1. `InitializingBean` and `DisposableBean`

- `InitializingBean`: Provides the `afterPropertiesSet()` method for initialization logic.
- `DisposableBean`: Provides the `destroy()` method for cleanup logic.

2. Annotations: `@PostConstruct` and `@PreDestroy`

- `@PostConstruct`: Marks a method to be executed after dependency injection and initialization.
- `@PreDestroy`: Marks a method to be executed before bean destruction.

Using BeanPostProcessor for Custom Logic

`BeanPostProcessor` is used to perform actions before and after a bean's initialization.

Example:

```
package com.example;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProces
sor;
import org.springframework.stereotype.Component;

@Component
public class CustomBeanPostProcessor implements BeanPostProces
sor {

    @Override
    public Object postProcessBeforeInitialization(Object bean,
String beanName) throws BeansException {
        System.out.println("Before Initialization: " + beanNam
e);
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean,
String beanName) throws BeansException {
        System.out.println("After Initialization: " + beanNam
e);
        return bean;
    }
}
```

Output (with `LifecycleBean`):

```
markdown
Copy code
1. Bean Instantiation
Before Initialization: exampleBean
2. @PostConstruct - Called after dependencies are injected.
3. InitializingBean's afterPropertiesSet() - Custom initializa
tion logic here.
4. Custom init-method - Declared in XML or Java config.
After Initialization: exampleBean
...
5. @PreDestroy - Cleanup before destruction.
6. DisposableBean's destroy() - Additional cleanup logic here.
7. Custom destroy-method - Declared in XML or Java config.
```

Phase	Callback Method	Example
Instantiation	Constructor	`new LifecycleBean()`
Dependency Injection	N/A	Dependencies are injected
Pre-Initialization	`BeanPostProcessor.postProcessBeforeInitialization()`	Customize pre-initialization logic
Initialization	`@PostConstruct`, `afterPropertiesSet()`, `init-method`	Initialization logic
Post-Initialization	`BeanPostProcessor.postProcessAfterInitialization()`	Customize post-initialization logic
Destruction	`@PreDestroy`, `destroy()`, `destroy-method`	Cleanup logic before bean removal

Bean Scopes in Spring

Bean Scope in Spring defines the lifecycle and visibility of a bean in the Spring context. It determines how and when a bean is created, how many instances are created, and how the bean is shared within the application.

Types of Bean Scopes

1. Singleton (Default Scope)

- **Description**: The Spring container creates a single instance of the bean, and it is shared across the application.
- **Scope Name**: `"singleton"`
- **Use Case**: Stateless beans, configuration, or shared resources.

Example:

```
@Component
@Scope("singleton")  // Optional as it's the default scope
public class SingletonBean {
    public SingletonBean() {
        System.out.println("SingletonBean instance created!");
    }
}
```

Output:

If this bean is retrieved multiple times:

```
ApplicationContext context = new AnnotationConfigApplicationCo
ntext(AppConfig.class);
SingletonBean bean1 = context.getBean(SingletonBean.class);
SingletonBean bean2 = context.getBean(SingletonBean.class);
System.out.println(bean1 == bean2); // true
```

2. Prototype

- **Description**: A new instance of the bean is created every time it is requested.
- **Scope Name**: `"prototype"`
- **Use Case**: Stateful beans or beans requiring a unique instance for every use.

Example:

```
@Component
@Scope("prototype")
public class PrototypeBean {
    public PrototypeBean() {
        System.out.println("PrototypeBean instance created!");
    }
}
```

Output:

If this bean is retrieved multiple times:

```
PrototypeBean bean1 = context.getBean(PrototypeBean.class);
PrototypeBean bean2 = context.getBean(PrototypeBean.class);
System.out.println(bean1 == bean2); // false
```

---

3. Request (Web Application Scope)

- **Description**: A new bean instance is created for each HTTP request.
- **Scope Name**: `"request"`
- **Use Case**: Beans specific to HTTP requests in web applications.

Example:

```
@Component
@Scope("request")
public class RequestBean {
    public RequestBean() {
        System.out.println("RequestBean instance created!");
    }
}
```

How to use:

This scope works in a Spring MVC or Spring Web application. Each HTTP request will create a new `RequestBean`.

---

4. Session (Web Application Scope)

- **Description**: A new bean instance is created for each HTTP session and shared within that session.
- **Scope Name**: `"session"`
- **Use Case**: Beans tied to user sessions.

Example:

```
@Component
@Scope("session")
public class SessionBean {
    public SessionBean() {
        System.out.println("SessionBean instance created!");
    }
}
```

---

5. Application (Web Application Scope)

- **Description**: A single instance of the bean is created for the lifecycle of the `ServletContext`.
- **Scope Name**: `"application"`
- **Use Case**: Beans that store application-wide state or configuration.

Example:

```
@Component
@Scope("application")
public class ApplicationBean {
    public ApplicationBean() {
        System.out.println("ApplicationBean instance create
d!");
    }
```

```
}
```

6. WebSocket (Web Application Scope)

- **Description**: A new bean instance is created and tied to a WebSocket session.
- **Scope Name**: `"websocket"`
- **Use Case**: Beans required for WebSocket communication.

Example:

```
@Component
@Scope("websocket")
public class WebSocketBean {
    public WebSocketBean() {
        System.out.println("WebSocketBean instance created!");
    }
}
```

Configuring Scopes in XML

Scopes can also be configured in an XML file:

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

    <!-- Singleton scope (default) -->
    <bean id="singletonBean" class="com.example.SingletonBean"
scope="singleton"/>

    <!-- Prototype scope -->
    <bean id="prototypeBean" class="com.example.PrototypeBean"
```

```
scope="prototype"/>

    <!-- Request scope -->
    <bean id="requestBean" class="com.example.RequestBean" sco
pe="request"/>

    <!-- Session scope -->
    <bean id="sessionBean" class="com.example.SessionBean" sco
pe="session"/>
</beans>
```

Scope

	Description	Instance Per	Use Case
Singleton	Default scope; single instance shared.	Spring container	Shared, stateless beans.
Prototype	New instance created every request.	Bean request	Stateful, non-shared beans.
Request	New instance per HTTP request (Web apps).	HTTP request	Request-scoped beans in MVC apps.
Session	New instance per HTTP session (Web apps).	HTTP session	Session-specific beans.
Application	Single instance for the entire application context.	Servlet context	Application-wide beans.
WebSocket	New instance per WebSocket session.	WebSocket session	WebSocket communication.