IoC

Inversion of Control (IoC) refers to the principle where the control of objects or portions of a program is transferred to a container or framework. In simpler terms, it means that the creation and management of objects are handled by the Spring framework rather than being controlled manually by the application code.

Key Points of IoC in Spring:

1. **Dependency Injection (DI):** The most common way IoC is implemented in Spring. DI is a design pattern used to implement IoC, where the Spring container injects the dependencies into a class at runtime rather than at compile time.
2. **Bean Management:** The Spring container manages the lifecycle of beans, including their creation, initialization, and destruction. The configuration of these beans can be done using XML configuration files, Java annotations, or Java-based configuration.
3. **Types of Dependency Injection:**
    - **Constructor Injection:** Dependencies are provided through a class constructor.
    - **Setter Injection:** Dependencies are provided through setter methods.
    - **Field Injection:** Dependencies are directly assigned to fields. (Less preferred due to testing and maintenance difficulties.)

DI

Dependency Injection (DI) in Spring is a design pattern and a core concept that allows the Spring framework to manage dependencies between objects. It promotes loose coupling by injecting dependencies (objects a class needs to perform its functions) rather than creating them internally. This makes the code more modular, easier to test, and maintainable.

Advantages of Dependency Injection

1. **Loose Coupling:** Classes are not tightly coupled, making them easier to test and maintain.
2. **Reusability:** Components can be reused across different parts of the application.
3. **Testability:** Easier to write unit tests by injecting mock dependencies.
4. **Flexibility:** Easy to swap out implementations of a dependency without changing the dependent class.

IoC Container

An IoC (Inversion of Control) container in Spring is a core component responsible for managing the lifecycle and configuration of application objects. The IoC container uses Dependency Injection (DI) to manage object creation, wiring, and destruction, promoting loose coupling and modular design.

Key Responsibilities of the IoC Container

1. **Object Creation:** The container instantiates and manages the lifecycle of beans (objects managed by the Spring container).
2. **Dependency Injection:** The container injects dependencies into beans, typically through constructor, setter, or field injection.
3. **Configuration Management:** The container reads configuration metadata (XML, annotations, or Java configuration) to determine how to assemble and configure beans.
4. **Bean Lifecycle Management:** The container manages the complete lifecycle of beans, including their creation, initialization, and destruction.
5. **Event Handling:** The container can publish and listen to application events, facilitating communication between beans.

In Spring, there are primarily two types of IoC (Inversion of Control) containers: **BeanFactory** and **ApplicationContext**. Each serves different purposes and offers varying levels of functionality.

1. BeanFactory

The `BeanFactory` is the simplest container, providing the basic features of dependency injection. It is suitable for lightweight, non-enterprise applications where only basic DI is required.

Characteristics:

- **Lazy Initialization:** Beans are created only when they are requested, not at startup.
- **Minimal Overhead:** Provides only basic DI functionality, which results in lower memory and CPU usage.

```java
BeanFactory factory = new XmlBeanFactory(new ClassPathResource("beans.xml"));
MyBean myBean = (MyBean) factory.getBean("myBean");
```

2. ApplicationContext

The `ApplicationContext` is a more advanced container that builds on the `BeanFactory` interface. It provides additional enterprise-specific functionality, making it suitable for most Spring-based applications.

Characteristics:

- **Eager Initialization:** By default, beans are created at startup, leading to faster retrieval but potentially higher startup time.
- **Internationalization Support:** Provides support for message resources and internationalization.
- **Event Propagation:** Can publish and listen to application events.
- **Declarative Mechanisms:** Supports declarative mechanisms such as annotations and aspect-oriented programming (AOP).

Common Implementations of `ApplicationContext`:

a. ClassPathXmlApplicationContext

Loads context definition from an XML file located in the classpath.

```java
ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
MyBean myBean = context.getBean(MyBean.class);
```

b. FileSystemXmlApplicationContext

Loads context definition from an XML file located in the file system.

```java
ApplicationContext context = new FileSystemXmlApplicationContext("path/to/applicationConte
MyBean myBean = context.getBean(MyBean.class);
```

c. AnnotationConfigApplicationContext

Loads context definition from annotated classes.

```java
@Configuration
@ComponentScan(basePackages = "com.example")
public class AppConfig {
}

ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
MyBean myBean = context.getBean(MyBean.class);
```

d. WebApplicationContext

A specialized version of `ApplicationContext` for web applications. It integrates with the lifecycle of a web application and provides convenient access to Spring's Web MVC infrastructure.

```java
@WebAppConfiguration
@ContextConfiguration(classes = AppConfig.class)
public class MyWebAppTest {
    @Autowired
    private WebApplicationContext webApplicationContext;
}
```

Singleton Bean

A **singleton** bean is the default scope in Spring. This means that there will be only one instance of the bean created per Spring IoC container. The same instance is shared across the entire application, and it is created when the application context is initialized.

Characteristics:

- **Single Instance:** Only one instance of the bean is created and used throughout the application.
- **Shared State:** All components that depend on the singleton bean share the same instance, which can lead to shared state.
- **Memory Efficiency:** Singleton beans are memory-efficient since only one instance is created.

Example:

Bean Definition in Java Configuration:

```java
@Configuration
public class AppConfig {
    @Bean
    public MySingletonBean mySingletonBean() {
        return new MySingletonBean();
    }
}
```

Bean Definition in XML Configuration:

```xml
<bean id="mySingletonBean" class="com.example.MySingletonBean" scope="singleton"/>
```

Usage:

```java
@Service
public class MyService {
    @Autowired
    private MySingletonBean mySingletonBean;

    // Use the singleton bean
}
```

Prototype Bean

A **prototype** bean is created each time it is requested from the Spring container. This means that a new instance of the bean is created for every call to `getBean()`, resulting in multiple instances of the bean within the same application context.

Characteristics:

- **Multiple Instances:** A new instance is created every time the bean is requested.
- **No Shared State:** Each instance is independent, so no shared state exists between different instances.
- **Less Memory Efficient:** More memory and resources are consumed as multiple instances are created.

Example:

Bean Definition in Java Configuration:

```java
@Configuration
public class AppConfig {
    @Bean
    @Scope("prototype")
    public MyPrototypeBean myPrototypeBean() {
        return new MyPrototypeBean();
    }
}
```

Bean Definition in XML Configuration:

```xml
<bean id="myPrototypeBean" class="com.example.MyPrototypeBean" scope="prototype"/>
```

Usage:

```java
Service
ublic class MyService {
   @Autowired
   private ApplicationContext applicationContext;

   public void usePrototypeBean() {
       MyPrototypeBean myPrototypeBean1 = applicationContext.getBean(MyPrototypeBean.class);
       MyPrototypeBean myPrototypeBean2 = applicationContext.getBean(MyPrototypeBean.class);

       // myPrototypeBean1 and myPrototypeBean2 are different instances
   }
```
