Day 7: Deep Dive into Dependency Injection in Spring Framework

🧠 Why Dependency Injection (DI) Matters

Imagine you're building a house. Would you build the plumbing, electricity, and furniture inside the house every time? No — you'd plug them in! Similarly, in software, **Dependency Injection** allows us to "plug in" required components rather than creating them every time.

Dependency Injection is a design pattern used to implement **IoC (Inversion of Control)**, where the control of object creation and dependency management is shifted from the application code to the framework — in this case, Spring Framework.

🔁 Inversion of Control (IoC) in Spring

IoC is the core principle where Spring takes control of object instantiation and lifecycle. Instead of your code controlling the flow, Spring does it — you just tell it *what* you want, and Spring provides *how*.

Spring achieves this through the **ApplicationContext**, which is essentially the Spring IoC container. It:

- Manages the lifecycle of beans.
- Handles DI through annotations or XML.
- Keeps everything loosely coupled and testable.

```java
ApplicationContext context =
  new AnnotationConfigApplicationContext(AppConfig.class);
MyService service = context.getBean(MyService.class);
```

Spring offers different types of ApplicationContext:

- ClassPathXmlApplicationContext: Used when configuring beans in XML.
- AnnotationConfigApplicationContext: Used when configuring beans via annotations and Java Config.
- WebApplicationContext: Specialized context for web applications.

🌱 What is Dependency Injection?

Dependency Injection is a technique where one object receives the dependencies it needs from an external source rather than creating them itself.

In traditional programming, a class is responsible for creating its dependencies. This leads to tight coupling. DI flips that — objects declare their dependencies, and a DI framework like Spring injects them.

Benefits:

- 🔄 Loose coupling between components
- 🔍 Easier unit testing (mock dependencies)
- 🧼 Cleaner, more maintainable code
- ♻️ Better reusability of components

Think of DI as handing a chef all ingredients and tools rather than letting them shop and cook in the same method.

💡 Ways to Inject Dependencies

1. Constructor Injection (Recommended)

Constructor Injection is the preferred way of injecting dependencies in Spring. It ensures that the object is always in a valid state and all required dependencies are available at creation time.

```
@Service
public class OrderService {
    private final PaymentService paymentService;

    public OrderService(PaymentService paymentService) {
        this.paymentService = paymentService;
    }
}
```

Why it's best:

- ✅ Ensures immutability
- ✅ Great for mandatory dependencies
- ✅ Makes unit testing easier
- ✅ Catches missing dependencies at compile time

2. Setter Injection

Used for optional dependencies or when you want to change a dependency after the bean is constructed.

```java
@Service
public class OrderService {
    private PaymentService paymentService;

    @Autowired
    public void setPaymentService(PaymentService
                                  paymentService) {
        this.paymentService = paymentService;
    }
}
```

Use case: Optional dependencies or when dependencies change post-creation.

Pros: Can modify dependency after object creation

Cons: Possible to forget setting the dependency, causing runtime errors

3. Field Injection

```java
@Service
public class OrderService {
    @Autowired
    private PaymentService paymentService;
}
```

Why avoid:

- ❌ Hard to test (no way to pass mocks easily)
- ❌ Breaks encapsulation
- ❌ Not ideal for production code

🏷️ Key Annotations Explained

These annotations are part of Spring's core support for DI and help the container understand what and how to inject.

@Component

Marks a class as a Spring-managed component. During component scanning, Spring detects this class and adds it to the application context.

@Service

A specialization of @Component for service layer classes. It's used for better semantics and clarity — Spring treats it the same as @Component.

@Repository

Also a specialization of @Component, specifically for the persistence layer. It enables exception translation, meaning database exceptions are converted into Spring's DataAccessException hierarchy.

@Controller

Used in web applications. It defines a controller class in Spring MVC that handles HTTP requests and returns responses.

@Autowired

Tells Spring to automatically inject a dependency. Works on constructors, setters, or fields.

@Qualifier

When multiple beans of the same type exist, @Qualifier is used to specify which bean should be injected.

```
@Autowired
@Qualifier("paypalService")
private PaymentService paymentService;
```

@Primary

Specifies the default bean to inject when there are multiple candidates.

```
@Primary
@Bean
public PaymentService stripeService() {
    return new StripeService();
}
```

These annotations are Spring's way of taking over dependency management and reducing boilerplate configuration.

🔄 Switching Implementations at Runtime

Spring supports multiple strategies for changing the implementation of a dependency without changing the consuming code.

You can dynamically switch between multiple implementations of an interface. For instance:

- StripeService for one region
- PaypalService for another

Option 1: Using @Qualifier

```java
@Service
public class OrderService {
    @Autowired
    @Qualifier("paypalService")
    private PaymentService paymentService;
}
```

Option 2: Using @Primary

If no @Qualifier is used, Spring injects the bean marked @Primary.

Option 3: Profiles

You can load different beans based on the environment.

```java
@Profile("dev")
@Bean
public PaymentService mockService() {
    return new MockPaymentService();
}

@Profile("prod")
@Bean
public PaymentService stripeService() {
    return new StripeService();
}
```

This helps in developing environment-specific logic (like mocking APIs in dev).

🧪 Unit Testing with Mocks

DI allows injecting mock versions of dependencies, making unit testing isolated and efficient.

Using Mockito:

```
@RunWith(MockitoJUnitRunner.class)
public class OrderServiceTest {

    @Mock
    private PaymentService paymentService;

    @InjectMocks
    private OrderService orderService;

    @Test
    public void testPlaceOrder() {
        when(paymentService.charge()).thenReturn(true);
        assertTrue(orderService.placeOrder());
    }
}
```

With constructor injection, you can even instantiate the class manually and pass mocks directly:

```
PaymentService mock = Mockito.mock(PaymentService.class);
OrderService service = new OrderService(mock);
```

🧰 Real-World Best Practices

- ✅ Use **constructor injection** for mandatory dependencies.
- 🛑 Avoid field injection in production code.
- 🧪 Write unit tests using constructor/setter injection with mocks.
- 🔀 Use @Qualifier or @Primary to handle multiple beans.
- 🌍 Use Spring Profiles for environment-specific configurations.
- 📚 Keep your services loosely coupled and focused on one responsibility.
- ♻️ Reuse beans rather than manually instantiating dependencies.

📌 Final Thoughts

Dependency Injection isn't just a Spring feature — it's a best practice in designing software that's modular, testable, and maintainable. By understanding how Spring handles DI via annotations and configurations, you empower yourself to write cleaner and more robust code.

Mastering Dependency Injection in Spring helps you:

- Build **testable**, **scalable**, and **loosely coupled** systems
- Improve productivity by letting Spring manage object lifecycles
- Focus more on **what your service does** rather than **how to wire it up**
