Java Technologies

Lecture 3

Beans, Dependency Injection and Cross-Cutting Concerns

Fall, 2025

Agenda

- Inversion of Control, Dependency Injection
- Injection Points, `@Autowired` and `@Inject` annotations
- Spring Beans, `@Component` & `@Bean` annotations
- Bean Scopes, Contexts, `@ApplicationContext`
- Naming Beans, Qualifiers, Alternatives
- Bean Lifecycle Events, SmartLifecycle
- Dependency Resolution, Circular Dependencies
- Loose Coupling via Event Publishing
- SpringEL and Conditional Event Listening
- Aspect Oriented Programinng, Spring AOP
- Interceptor and Decorator Design Patterns

The Problem

- A class **controlling its own dependencies**.

```
class UserService {

  private JdbcUserRepository repo = new JdbcUserRepository(); ❌

  public void registerUser(String name) {
    repo.save(new User(name));
  }
}
```

- **Tight coupling**

  `UserService` ”knows” about `JdbcUserRepository`.
- **Lack of flexibility**

  How to use JPA instead of JDBC, or maybe both?
- **Unit testing challenges**

  How to test without access to the repo's database?
- **Lifecycle responsability**

How to Solve It?

- Delegate the dependency management to a third party.
- First idea: **interfaces** and **object factories**

```
  private UserRepository repo = userRepoFactory.getInstance();
```

- **The Good**: Centralizes creation logic.
- **The Bad**: Boilerplate code (creating the factory); Still tight coupling between objects and factories; Still hard to test.
- **The Solution**:

```
class UserService {
  private final UserRepository repo; // Declare what you need

  public UserService(UserRepository repo) { ✓
      this.repo = repo; // The "third party" provides the repo
  }
  public void registerUser(String name) {
    repo.save(new User(name));
  }
} // This can only happen if we use an application framework
```

Inversion of Control (IoC)

- Implementations should not depend upon other implementations, but instead **they should depend upon abstractions**.
- IoC is a principle that addresses **resolving dependencies** between components: instead of a class controlling its dependencies, something external provides them.

  $\rightarrow$ **IoC container**
- **Advantages**:
  - Reduces coupling
  - Improves testability
  - Promotes flexibility & extensibility
  - Encourages separation of concerns
- The "Hollywood Principle": **"Don't call us, we'll call you."**

Dependency Injection (DI)

- SOLI**D**
- **Dependency Injection** is a design technique / pattern that **implements** the Inversion of Control principle.
  (Service Locator is another pattern that implements DI).
- **Injection** is the passing of an actual object described as an abstract dependency to a dependent object.
- **Terminology**: Service / Component, Interface, Client, Container / Injector / Assembler / Provider / Factory.
- **DI Container**
  - Implemented by application servers or frameworks.
  - Manages object creation: "[managed] beans", "components".
  - Manages object lifecycle: create new, reuse, or destroy.
  - Resolves dependencies, injecting them to the caller.
  - Handles configuration: what kind of implementation to use.

Injection Points

1. **Constructor** (recommended, for mandatory dependencies)

```
class UserService {
    private final UserRepository repo;
    public UserService(UserRepository repo) { this.repo = repo; }
}
```

2. **Field** (the easiest, not recommended: less clean, hard to test)

```
    @Autowired
    private UserRepository repo;
```

3. **Setter** (for optional dependencies) and **Method** (less common)

```
    private final UserRepository repo;

    @Autowired(required = false)
    public void setUserRepository(UserRepository repo) {
        this.repo = repo;
    }
```

Even if it is not recommended, we will use the field injection point quite often on these slides, only because it is more compact.

`@Autowired` and `@Inject`

- `@Autowired` is the Spring annotation that allows for the automatic injection of dependencies (autowiring).
- How it works:
  - **Dependency Discovery**: The container scans for components.
  - **Type-based Matching**: It searches for beans that "fit" the types required by clients, using a **resolution** algorithm.
  - **Injection**: The client gets the desired dependency (or not).
- **Contexts and Dependency Injection (CDI)**
  - `@Inject` is the standardized, Java EE version of `@Autowired`.
  - Spring supports both `@Autowired` and `@Inject`.
  - Using `@Inject` makes your code more portable across frameworks, because it relies on the standard CDI specification.

```
    <dependency>
        <groupId>jakarta.inject</groupId>
        <artifactId>jakarta.inject-api</artifactId>
    </dependency>
```

Example: Using DI in Spring

```java
public interface Greeting {
    String sayHello();
}
```

```java
@Component
public class DefaultGreeting implements Greeting {
    @Override
    public String sayHello() { return "Hello!"; }
}// @Component is the generic stereotype annotation in Spring.
```

```java
@Component
public class Client {
    private final Greeting greeting;

    public Client(Greeting greeting) {
        this.greeting = greeting; // Constructor injection
    }
    public void doWork() {
        System.out.println(greeting.sayHello());
    }
}
```

Example: Running the Application

```
@SpringBootApplication
public class ExampleApplication {

    public static void main(String[] args) {
        SpringApplication.run(ExampleApplication.class, args);
    }

    // Run our Client when the app starts
    @Bean
    CommandLineRunner run(Client client) {
        return args -> client.doWork();
    }
}
```

- @Bean annotation tells Spring "the method's return value is a **managed bean** in the **application context**."
- `CommandLineRunner` is a functional interface. Spring Boot automatically detects all beans of this type and executes them right after the **application context is fully initialized**.

Spring Beans

- A **bean** is an object **managed by the Spring IoC** container. Spring keeps track of these beans, handles their lifecycle, and can inject them into other beans.

- Beans are **Plain Old Java Objects (POJOs)**
- They are created via **Annotation-Based Configuration**: `@Component` and its specializations, or `@Bean`.

The @Bean annotation

- @Bean is a **method-level** annotation,
  unlike @Component which is class-level annotation.

```
@Configuration
public class AppConfig {

    @Bean
    public MyService myService() {
        return new MyServiceImpl();
        // The returned object becomes a Spring bean.
        // It's name is 'myService'.
        // It is a singleton by default.
    }
}
```

- You must use @Bean inside a @Configuration class to ensure proper container behavior regarding lifecycle management.
- Used for creating objects having **custom creation logic**.
- The Java EE equivalent for this is using @Producer methods.

Bean Scopes

- The **scope** of a bean determines its lifecycle and visibility.
- **Singleton**: One shared instance per Spring container.

```
@Component
@Scope("singleton") // optional, default
public class UserService { }
```

- **Prototype**: New instance every time it's injected.

```
@Scope("prototype")
```

- **Request**: One instance per HTTP request.
- **Session**: One instance per HTTP session.
- **Application**: One instance per ServletContext.
- **WebSocket**: One instance per WebSocket session.
- Custom scopes can be created using the `Scope` interface.
- Java EE CDI offers type-safe scopes, such as `@RequestScoped`.

Beans With Custom Scopes

- **Singleton**: container manages creation and destruction.
- **Prototype**: creates and injects; it does not destroy them.
- **Request, Session, Application**: specific to Web Contexts.
- How to implement a **custom scope**?

```
public class MyScope implements Scope {
    private final Map<String, Object> beans = new HashMap<>();
    ...
}
```

- Register the new scope.

```
@Configuration
public class ScopeConfig { // Spring Boot custom configuration
  @Bean
  public static CustomScopeConfigurer customScopeConfigurer() {
    var configurer = new CustomScopeConfigurer();
    configurer.addScope("myscope", new MyScope());
    return configurer;
  }
}
```

Using a Custom Scope Bean

- Define a bean / component with the custom scope.

```
@Component
@Scope("myscope")
public class MyBean {
    public void doWork() {
        System.out.println("MyBean working...");
    }
    public void cleanup() {
        System.out.println("MyBean cleaned up!");
    }
}
```

- Create an instance, use it, destroy it. OK, But Why? 😟

```
    @Autowired MyBean mybean;
    ...
    bean.doWork();
    ...
    MyScope scope = (MyScope)
        context.getBeanFactory().getRegisteredScope("myscope");
    scope.destroyBean("myBean");
```

What Exactly Is a "Context"?

- **Context** (dictionary): the circumstances that form the setting for an event, statement, or idea, and in terms of which it can be fully understood and assessed.
- Synonyms: frame of reference, conditions, factors, state of affairs, situation, background.
  "The problem is to decide what this means in the context in which the words are used."
- In our "context", **a context is a container that manages objects** (beans/components) within a defined scope, and provides them with various services such as lifecycle management, dependency injection, and resource access.
- Java EE examples: Servlet Context, JSF Context, EJB Context, JMS Context, Naming Context, SOAP Message Context, etc.

Spring ApplicationContext

- **ApplicationContext** is the heart of Spring. ♥
- It is the central interface to the **Spring IoC container**.
- It inherits from **BeanFactory**, which represents core features.
- It **manages beans**: creates them, wires dependencies, etc.
- It provides **enterprise services**, such as internationalization, event publishing, or resource loading.

```
ApplicationContext ctx = SpringApplication.run(MyApp.class, args);
```

```
@Component
public class MyComponent {
    @Autowired
    private ApplicationContext ctx;

    public void showBeans() {
        MyService service = ctx.getBean(MyService.class);
        var beans = ctx.getBeanDefinitionNames());
    }
} // In most cases, you don't need the ApplicationContext directly.
```

Naming Beans and Qualifiers

- Beans (@Component, @Bean) can be given custom names.

```
@Component("mySpecialService")
public class MyServiceImpl implements MyService { ... }
```

```
@Configuration
public class MyConfig {
    @Bean("mySpecialService", "mySuperService")
    public MyService myService() { return new MyServiceImpl(); }
}
```

- Beans can be injected using their custom names. ⚠

```
@Component
public class MyClient {
    private final MyService myService;
    @Autowired
    public MyClient(
            @Qualifier("mySpecialService") MyService myService) {
        this.myService = myService;
    }
} // Lookup using string-based names may be problematic.
```

Type Safe Qualifiers

- Create a **custom annotation** to describe the qualifier.

```
@Target({FIELD, PARAMETER, TYPE}) // ElementType
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface SpecialService {}
```

- Annotate a bean or a component.

```
@Component
@SpecialService
public class MyServiceImpl implements MyService { }
```

- Inject it **type-safely**.

```
    @Autowired
    public MyClient(@SpecialService MyService myService) {
        this.myService = myService;
    }
```

- The Java EE equivalents are `@Named` and `@Qualifier`.

Alternatives

- Used to provide an **alternative implementation** of a bean, that can be enabled at **deployment** time.
- In CDI, it is implemented using `@Alternative` + `beans.xml`.
- Spring offers **Profile-Based Alternatives** (@Profile)

```
@Service @Profile("prod1")
class CustomService1 implements MyService { ... }
@Service @Profile("prod2")
class CustomService2 implements MyService { ... }

spring.profiles.active=prod1
```

- Components can be selected programmatically.

```
    @Bean
    public MyService myService(Environment env) {
        if (env.acceptsProfiles("prod1")) {
            return new CustomService1();
        } else { ... }
    }
```

Bean Lifecycle Events

- Bean lifecycle events are **controlled hooks** into key moments of a bean's existence, enabling proper resource management.
- **`@PostConstruct`** annotated methods are called after the bean is fully initialized and dependencies are injected.

```
    @PostConstruct // ← Java EE standard CDI annotation
    public void init() {
        System.out.println("Bean initialized");
    } // opening resources (database connections, files)
```

- **`@PreDestroy`** annotated methods are called before the bean is destroyed – when the application context closes.

```
    @PreDestroy    // ← Java EE standard CDI annotation
    public void cleanup() {
        System.out.println("Bean destroyed");
    } // closing resources
```

- Spring specific: `InitializingBean`, `DisposableBean`

Global Lifecycle Hooks

- **BeanPostProcessor** is used to **intercept bean creation**.

```
@Component
public class MyBeanPostProcessor implements BeanPostProcessor {

  @Override
  public Object postProcessBeforeInitialization
        (Object bean, String beanName) throws BeansException {
    System.out.println("Before init: " + beanName);
    return bean;
  } // Initialization refers to @PostConstruct

  @Override
  public Object postProcessAfterInitialization
        (Object bean, String beanName) throws BeansException {
    System.out.println("After init: " + beanName);
    return bean;
  }
} // This is a global interceptor, not tied to a single bean.
```

- It runs before (and after) any bean initialization logic.
- It allows **cross-cutting** behavior (AOP): logging, proxies...

Coordinated Lifecycle Management

- **`SmartLifecycle`** provides a mechanism for beans to **participate in the global startup and shutdown process**.

```java
@Component
public class DatabaseInitializer implements SmartLifecycle {
  private boolean running = false;
  public void start() { this.running = true; }
  public void stop() { this.running = false; } //+Callback
  public boolean isRunning() { return this.running; }
  public int getPhase() { return 0; } //Start first
  public boolean isAutoStartup() { return true; } //After refresh
}
```

```java
@Component
public class DatabaseWorker implements SmartLifecycle {
  ...
  public int getPhase() {
      return 1; // Start AFTER the database initializer
  }
} // At shutdown, the container will stop beans in reverse order.
```

Just Remember: "There Can Be Only One"

```java
@Autowired
private SomeBeanType bean;
```

Dependency Injection Resolution

- When the Spring container starts, it reads all the bean definitions and builds a **graph of dependencies**. 💡
- Resolution strategy: **Type, Qualifier, Name**.
- `BeanDefinitionStoreException`: beans with the same name.
- `NoSuchBeanDefinitionException`: no bean matches.
- `NoUniqueBeanDefinitionException`: multiple beans match.

```
@Component
class DefaultGreeting implements Greeting { ... }
@Component
class CustomGreeting  implements Greeting { ... }

@Autowired
private Greeting greeting; ✗
```

- **Resolving ambiguities**: using `@Primary`, `@Qualifier`

```
@Component
@Primary
class DefaultGreeting implements Greeting { ... }
```

The "Happy-Path" Resolution

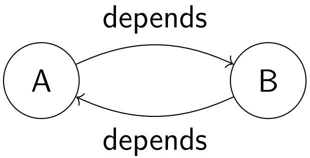
What happens when Spring needs to resolve a dependency.

1. It collects all candidate beans by type.
2. It attempts to instantiate them to see if they're resolvable.
3. If exactly one candidate remains, it is injected.
   $\rightarrow$ This is the happy-path resolution.
4. If none can be instantiated, Spring throws a `NoSuchBeanDefinitionException`.
5. If more than one are instantiable, but no disambiguation is provided (`@Primary`, `@Qualifier`, matching field/parameter name), Spring throws a `NoUniqueBeanDefinitionException` because the dependency is ambiguous.

Circular Dependencies

- **A depends on B**, and **B depends on A**.

```
@Component
class A {
    private final B b;
    public A(B b) { this.b = b; }
}
@Component
class B {
    private final A a;
    public B(A a) { this.a = a; }
} // Constructor based injection doesn't work ❌
```



- **Singleton beans**: Setter injection or field injection circular references are resolved via the "Three-Phase Creation":

  Instantiate → Populate properties → Initialize. 💡

- **Lazy** resolution: `public A(@Lazy B b)` → Proxy
- **Optional** dependencies: `@Autowired(required = false)`

Lazy Bean Initialization

- By default, the IoC container creates all singleton beans **eagerly** at application startup. If some beans have high startup time:
  - the development phase might suffer → 😞
  - if some of those beans will never get used → waste
- **Lazy initialization** is a mode where the container only creates beans when they are first requested, not at startup.

```
# application.properties
# Enables lazy initialization globally for the entire app
spring.main.lazy-initialization=true
```

- It can be used only on specific beans.

```
@Component @Lazy public class MyExpensiveComponent {...}
@Bean @Lazy public MyExpensiveService myExpensiveService() { ... }
```

- **Drawbacks**: Delayed discovery of errors, latency in the first request, can mask configuration problems, background tasks won't start, controllers will return 404 on their first request.

Object Provider

- If you have multiple beans of the same type and try to autowire them → `NoUniqueBeanDefinitionException`. How to choose one of them in a more flexible, programmatic way?
- A singleton bean is created at startup, so its dependencies are resolved also at startup. What if we want to inject in a singleton bean a prototype bean or a request sope bean on demand?
- ObjectProvider offers **a more flexible way** to access beans compared to direct dependency injection.

```java
@Service
public class MyService {
  @Autowired
  private ObjectProvider<MyDependency> dependencyProvider;

  public void doSomething() {
      var dependency = dependencyProvider.getObject();
      // .getIfAvailable(), .getIfUnique(), .stream()
  }
}
```

Loose Coupling Revisited

- Designing components so they are interconnected but depend on each other as little as possible.
- **Implementation level**: Decoupling the service and the client by means of abstract types and qualifiers, so that the service implementation may vary.
- **Communication level**
  - Decoupling the lifecycles of collaborating components by making them contextual, with automatic lifecycle management.
  - Decoupling message producers from consumers using **events**.
- Decoupling **orthogonal concerns** by means of **interceptors**.
- Separating an object's core functionality from its additional, **cross-cutting concerns** using **decorators**.

Application Events

- **Events** allow beans to interact with no compile time dependency at all, through Spring's application context.
- This is based on the **Observer** pattern (Publisher/Subscriber).
  - Event producers raise events.
  - Events are delivered to observers by the container.
  - The event object (POJO) carries state.
  - Producers (services) and consumers (listeners) are decoupled.
- Events have advanced features:
  - **Conditional**: handle only some events of a specific type.
  - **Transactional**: publish events after transaction commits.
  - **Asynchronous**: listener executed in a separate thread.
  - **Ordered**: listeners executed in a specific order.
- Encourage domain-driven design (events as domain signals).
- Spring Boot provides several built-in events, such as `ApplicationStartingEvent` or `ApplicationReadyEvent`.

Example: Using Events

- Define a custom event (POJO).

```
public record OrderCreatedEvent(String orderId) { }
```

- Publish the event.

```
@Service
public class OrderService {
    @Autowired
    private ApplicationEventPublisher publisher;
    public void createOrder(String orderId) {
        System.out.println("Creating order: " + orderId);
        publisher.publishEvent(new OrderCreatedEvent(orderId));
    }
} // ApplicationContext extends ApplicationEventPublisher
```

- Listen for events in any component.

```
    @EventListener
    public void handleOrderCreated(OrderCreatedEvent event) {
        System.out.println("Order notification: " + event);
    }
```

Conditional Event Listening

- Conditional event listening is implemented using **SpEL**.
- Match by the value of a property.

```
@EventListener(condition = "#event.status == 'important'")
public void handleImportantEvent(CustomEvent event) {
    // Handle only important events
}
```

- Match by an expression.

```
@EventListener(condition = "#event.message.length() > 5")
public void handleLongMessages(CustomEvent event) { ... }
```

- Use environment properties.

```
@EventListener( condition =
  "@environment.getProperty('events.enabled') == 'true'")
public void handleIfEnabled(CustomEvent event) { ... }
```

Spring Built-in Events

- **ContextRefreshedEvent** - published when the ApplicationContext is initialized or refreshed; useful for post-initialization logic.
- **ContextStartedEvent**
- **ContextStoppedEvent**
- **ContextClosedEvent**
- **RequestHandledEvent** (web only) - published after an HTTP request is processed in Spring MVC.

Note: Spring Boot extends this set (e.g., `ApplicationStartingEvent`, `ApplicationReadyEvent`, `ApplicationFailedEvent`) during the application lifecycle.

Aspect Oriented Programming (AOP)

- An **aspect** is a feature that's typically scattered across methods, classes, object hierarchies, or even entire object models.
  $\rightarrow$ Cross-cutting concern
- It is a behavior that "looks and smells" like it should have structure, but you can't find a way to express this structure with traditional object-oriented techniques.
  $\rightarrow$ Logging, Profiling, Transactions, Caching, Security.
- How to separate the cross-cutting concerns from the model?
- AOP provides a way to encapsulate this type of functionality by adding behavior, such as logging, "around" your code.
- AOP is a **programming paradigm** that complements OOP.
- **Implementations**: AspectJ, Spring AOP, CDI (a little).

Core Concepts of Spring AOP

- **Aspect**: A module that encapsulates a cross-cutting concern.
- **Join Point**: A point in the execution of the program where an aspect can be applied (a method execution).
- **Advice**: Code that runs at a join point.
  $\rightarrow$ Before, After, AfterReturning, AfterThrowing, Around
- **Pointcut**: A predicate or expression that matches join points.
- **Target Object**: The object on which an advice is applied.
- **Weaving**: The process of applying aspects to the target code.
  $\rightarrow$ Compile-time, Load-time, Runtime.
- **Proxy**: Spring AOP works by creating at runtime a proxy object that wraps the target object.
  - JDK Dynamic Proxies: if the target implements interfaces.
  - CGLIB Proxies: if the target does not implement interfaces.

Example: Using Spring AOP

- Create the target (any type of component).

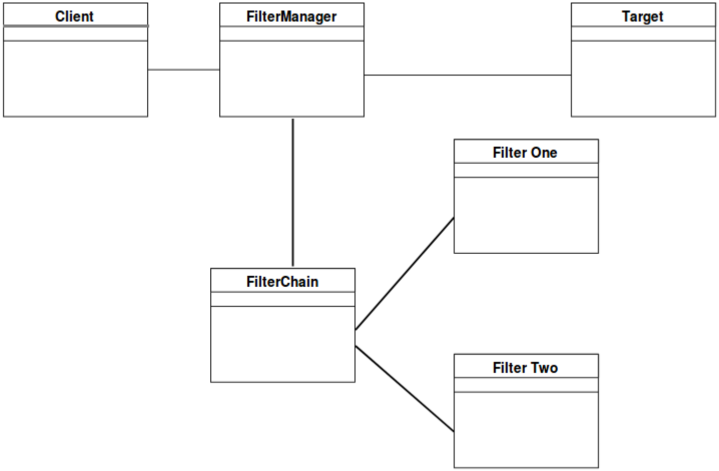
```java
@Service
public class MyService {
    public void hello() { System.out.println("Hello!"); }
}
```

- Create the aspect, defining the advice and a pointcut. The pointcut is specified using an **AspectJ-style expression**.

```java
@Aspect
@Component
public class LoggingAspect {
    @Around("execution(* com.example.demo.MyService.hello(..))")
    public Object aroundHello(ProceedingJoinPoint joinPoint)
            throws Throwable {
        System.out.println("Before: " + joinPoint.getSignature());
        Object result = joinPoint.proceed();
        System.out.println("After: " + joinPoint.getSignature());
        return result;
    } // The advice is the content of this method.
} // Add in pom.xml the starter spring-boot-starter-aop
```

Interceptor Design Pattern

Insert **processing logic before and/or after** the actual processing of a request, without modifying the core business logic.



Implementing an Interceptor

- Define an **interceptor binding** as a marker annotation.

```
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface LogExecution { }
```

- Create the aspect defining the interceptor.

```
@Around("@annotation(com.example.demo.LogExecution)")
public Object around(ProceedingJoinPoint joinPoint)
        throws Throwable {...}
```

- Apply the interceptor binding to a class or to a method.

```
    @LogExecution
    public void hello() { ... }
```

- @Around advice + custom annotation in Spring AOP is equivalent to `@Interceptor` + `@AroundInvoke` in CDI.

Decorator Design Pattern

- Decorator is a structural design pattern that allows you to add responsibilities to an existing bean dynamically at runtime.

```
var reader = new BufferedReader(new FileReader("file"));
```

- It can be implemented in Spring AOP, just like an interceptor.
- It can also be implemented using **bean wrapping**

```
@Service @Primary
public class GreetingServiceDecorator implements GreetingService {
    @Autowired
    private GreetingService delegate;

    @Override
    public String greet(String name) {
        String original = delegate.greet(name); // decorate it
        return "Decorated: " + original;
    }
}
```

- The CDI equivalent is `@Decorator` + `@Delegate`.

Interceptors vs. Decorators

- **Interceptors**
  - Offer a powerful way to capture and separate concerns which are orthogonal to the application.
  - An interceptor may intercept invocations of any type.
  - They are perfect for solving technical concerns such as transaction management, security and logging.
- **Decorators**
  - Used for separating business-related concerns.
  - Do not have the generality of interceptors.
  - A decorator should intercepts invocations only for a certain type, being aware of all the semantics attached to that type.
- Interceptors and decorators, though similar in many ways, are complementary.