SPRING TRAINING

16.09.2023

Gülümser Aslan & Mustafa Yumurtacı

trendyol learning

PURPOSE

Understanding the Spring concept

WHY

Makes programming in Java

- ❑ quicker
- ❑ easier
- ❑ safer

AGENDA

1. Introducing The Spring Framework
2. Core Concepts: DI & IoC
3. Java Based Configuration
4. Annotation Based Configuration
5. Introducing Aspect Oriented Programming

Overview


dependency injection
inversion of control
service
advice
bean
aspect
configuration
annotation
pointcut
component
spring boot
dependency

1. Introduction to Spring Framework

1. What Is Framework?
2. What Is Spring Framework?
3. History of Spring
4. Spring vs Spring Boot
5. Why Spring Is Successful?

What Is Framework?

Framework

- provides tools and rules.
- helps developers to create applications *more* easily and efficiently.
- saves time and effort by offering pre-built structures and functions

1. Introduction to Spring Framework

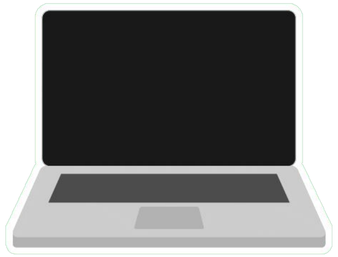
1. What Is Framework?
2. What Is Spring Framework?
3. History of Spring
4. Spring vs Spring Boot
5. Why Spring Is Successful?

What Is Spring Framework?

Spring Framework

- is an application framework part of Java ecosystem.
- focus on *speed*, *simplicity*, *productivity*.
- is the most popular Java framework.
- builds java enterprise applications.
  - open source
  - lightweight
  - dependency injection container

Spring Framework Is Open Source



Code is available at:
https://github.com/spring-projects/spring-framework

Documentation available at:
https://docs.spring.io/spring-framework/reference/

Spring Framework Is Lightweight

Spring applications do not require a Java EE application server

- ❏ You can run your application as a standalone application

Spring is not invasive

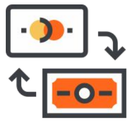
- ❏ Does not require you to extend or implement framework classes

Spring jars are relatively small

Spring Framework Provides a DI Container

Spring serves as a Dependency Injection(DI) container

- ❏ Your objects do not have to worry about finding / connecting to each other



Spring inject dependencies into your objects

Spring serves as a lifecycle manager

Spring Framework: More Than a DI Container

Spring integrates with a wide variety of technologies / platforms:

- ❏ Cloud, Micro-services
- ❏ JDBC, Transactions, JPA, NoSQL
- ❏ Events, Streaming, Reactive, Messaging, RabbitMQ, Kafka
- ❏ Security, OAuth2
- ❏ Monitoring, Observability

Spring simplify working with lower-level technologies

Spring is extensible and customizable

1. Introduction to Spring Framework

1. What Is Framework?
2. What Is Spring Framework?
3. History of Spring
4. Spring vs Spring Boot
5. Why Spring Is Successful?

History of Spring

2003 **Rod Johnson published a book**

Spring 1.0 is released 2004

2006 **Spring 2.0:**
- Simplified Xml configuration
- Java 5 support

Spring 2.5:
- Annotation configuration

2007

2009 **Spring 3.0:**
- Support expression language
- Java based configuration

Spring 4.0:
- SPEL support
- Java 8 support

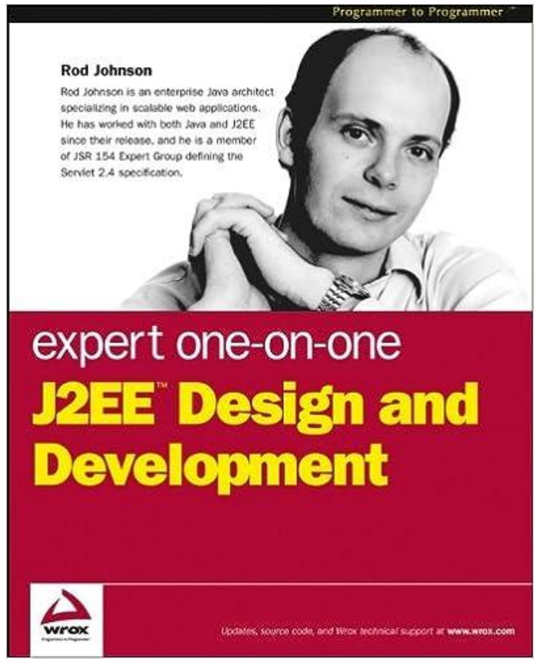
2013

2014 **Spring Boot 1.0 is released**

Spring 5.0:
- Reactive programming
- Java 9 support

2017

2022 **Spring 6.0:**
- Java 17 support



trendyol learning

1. Introduction to Spring Framework

1. What Is Framework?
2. What Is Spring Framework?
3. History of Spring
4. Spring vs Spring Boot
5. Why Spring Is Successful?

Spring vs Spring Boot

SPRING	SPRING BOOT
An open-source lightweight framework.	Built on top of the conventional Spring Framework
Used to develop enterprise applications	Used to develop REST APIs.
The most important feature of the Spring Framework is **dependency injection**.	The most important feature of the Spring Boot is **Autoconfiguration**.
Developers should write configuration codes.	In Spring Boot everything is auto-configured.

💫 You can create a SpringBootApplication from here: https://start.spring.io/

1. Introduction to Spring Framework

1. What Is Framework?
2. What Is Spring Framework?
3. History of Spring
4. Spring vs Spring Boot
5. Why Spring Is Successful?

Why is Spring Successful?

Spring embrace change and continues to improve:

- ❏ new JDK Versions
- ❏ Reactive Programming
- ❏ Stream Processing
- ❏ Kotlin Support
- ❏ Kubernetes

Integration with other open source projects

- ❏ *Hibernate, Quartz*

Created for common enterprise domains:

- ❏ *Spring Security, Batch, Integration*

Spring Boot created for *simplify* DevEx

Spring deals with boilerplate codes.

- ❏ You can focus on solving business domain

Spring deals with →

trendyol learning

Homework

1. Please visit --> https://spring.io/
2. Look at Spring Overview --> https://docs.spring.io/spring/docs/current/spring-framework-reference/overview.html#overview
3. Read Why Spring? --> https://spring.io/why-spring
4. Look at the last released versions and features -->

https://github.com/spring-projects/spring-framework/releases

2. Core Concepts: DI & IoC

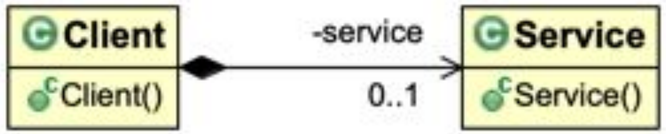
1. What Is Dependency?
2. Managing Dependency
3. What Is Dependency Injection?
4. What Is Inversion of Control?
5. DI and IoC
6. Lab Section: *Step by Step Into Spring DI*

What Is Dependency?

Dependency

- defines a relationship between service - client or consumer - provider
- service : *provides*
- client: *consumes*

Dependency: *Example*




```java
public class Client {

   private Service service;
}
```

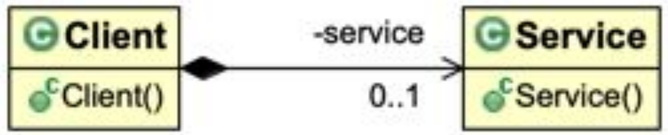
```java
public class Service {

}
```

Client is dependent to service

2. Core Concepts: DI & IoC

1. What Is Dependency?
2. Managing Dependency
3. What Is Dependency Injection?
4. What Is Inversion of Control?
5. DI and IoC
6. Lab Section: *Step by Step Into Spring DI*

Managing Dependency




```java
public class Client {

   private Service service;

   public Client() {
       service = new Service();
   }
}
```

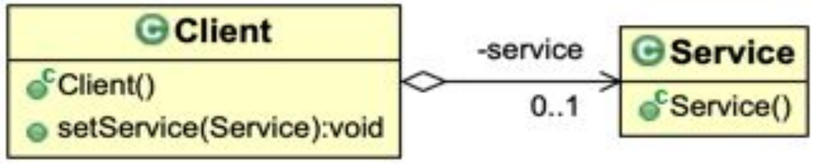
```java
public class Service {

}
```

Client object creates Service object.

High Coupling

trendyol learning

Managing Dependency: A possible solution



1. constructor method

2. setter method

```java
public class Client {

   private Service service;

   public Client() {
       service = new Service();
   }

   public void setService(Service service) {
       this.service = service
   }
}
```

```java
public class Service {

}
```

Creating a Service object and passing it to the Client object

△ better solution for coupling and complexity of Client

trendyol learning

2. Core Concepts: DI & IoC

1. What Is Dependency?
2. Managing Dependency
3. What Is Dependency Injection?
4. What Is Inversion of Control?
5. DI and IoC
6. Lab Section: *Step by Step Into Spring DI*

What Is Dependency Injection?

Dependency Injection

- is creating a dependent object and passing this object.
- aims simpler mechanism for component dependencies
- manages component's life cycles.

Client object does not have to be responsible for creating its parts or its Service objects.
Removing the responsibility of Service object creation allows the Client object to be **highly-cohesive** and **lowly-coupled**.

Now, if the client doesn't create the service, *who* does?
We will see the answer this question in the lab section.

2. Core Concepts: DI & IoC

1. What Is Dependency?
2. Managing Dependency
3. What Is Dependency Injection?
4. What Is Inversion of Control?
5. DI and IoC
6. Lab Section: *Step by Step Into Spring DI*

What Is Inversion of Control?

Inversion of Control

- is about inverting the owner of the control of the flow in an application.
- is a common principle applied by frameworks.
- shortly IoC

A framework when extended by an application takes over the control of the flow of that application and manages it by making calls on its objects and the objects of the application.

2. Core Concepts: DI & IoC

1. What Is Dependency?
2. Managing Dependency
3. What Is Dependency Injection?
4. What Is Inversion of Control?
5. DI and IoC
6. Lab Section: *Step by Step Into Spring DI*

DI and IOC

Dependency Injection (DI) is achieved by IoC. **IoC is used to enable DI.**

IoC is an underlying principle applied in the inner workings of frameworks and DI is one of many techniques where IoC is applied.

IoC is used in many places other than creating and wiring objects such as managing object's life cycle, events, AOP, etc.

2. Core Concepts: DI & IoC

1. What Is Dependency?
2. Managing Dependency
3. What Is Dependency Injection?
4. What Is Inversion of Control?
5. DI and IoC
6. Lab Section: *Step by Step Into Spring DI*

Lab Section


```
public static void main(String[] args) {
    System.out.println("Hello World!");
}
```

- What is wrong with this code?
- *Hint*: Think about dependencies.
- What if we want to say greeting in another language like “Selam”?
- Application must be changed for each different greeting.

Lab Section

Let’s do some refactor and learn what is Spring

3. Java Based Configuration

1. Java Based Configuration Concept
2. Application Context
3. Handling Multiple Configuration
4. Bean Scopes
5. Property Values
6. Spring Profiles
7. Coding Section: *Java Based Configuration*

Java Based Configuration Concept

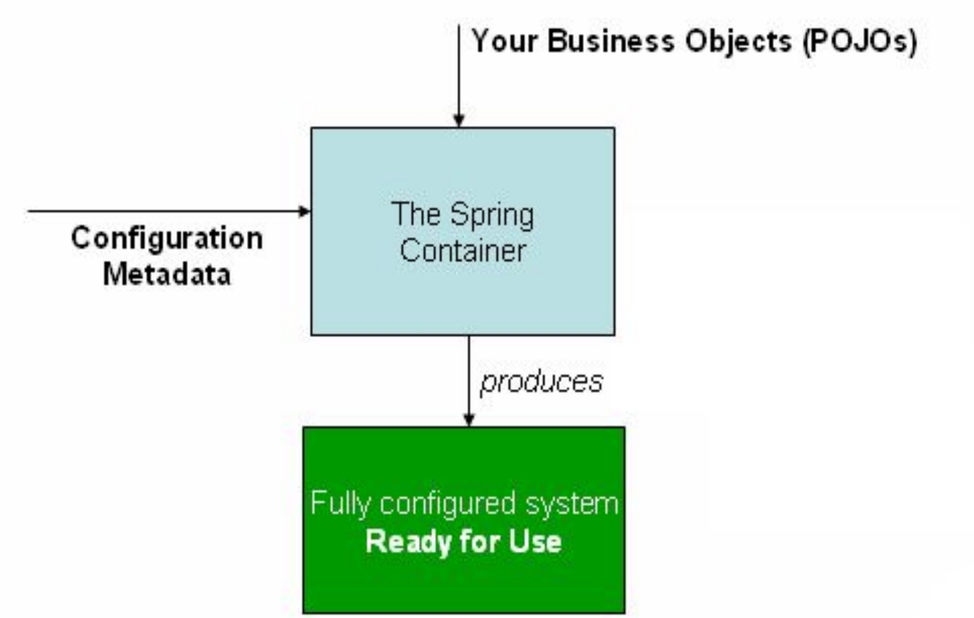


Figure 3.1: *The following diagram shows a high-level view of how Spring works. Application classes are combined with configuration metadata so that, after the ApplicationContext is created and initialized, you have a fully configured and executable system or application.*

- How can we initialize POJOs?
- How can we configure the dependencies they have?

Java Based Configuration Concept

No new operator
No service located
AccountRepository dependency

```
public class TransferServiceImpl implements TransferService {
    public TransferServiceImpl(AccountRepository ar) {
        this.accountRepository = ar;
    }
    ...
}
```

Dependency: Needed to perform money transfers between accounts

```
public class JdbcAccountRepository implements AccountRepository {
    public JdbcAccountRepository(DataSource ds) {
        this.dataSource = ds;
    }
    ...
}
```

Dependency: Needed to access account data in the database

No new operator
No service located
DataSource dependency

What we need?
Decoupling these components
Configuring our own components

Configuration Instructions with Dependencies

@Configuration: Tells the framework that we have definitions inside this class.
@Bean: Tells spring that defines a bean in ApplicationContext.

Method Invocation

```
@Configuration
public class ApplicationConfig {
  @Bean public TransferService transferService() {
    return new TransferServiceImpl( accountRepository() );
  }
  @Bean public AccountRepository accountRepository() {
    return new JdbcAccountRepository( dataSource() );
  }
  @Bean public DataSource dataSource() {
    BasicDataSource dataSource = new BasicDataSource();
    dataSource.setDriverClassName("org.postgresql.Driver");
    dataSource.setUrl("jdbc:postgresql://localhost/transfer" );
    dataSource.setUsername("transfer-app");
    dataSource.setPassword("secret45");
    return dataSource;
  }
}
```

Represents a dependency

Pass as Reference

```
@Configuration
public class ApplicationConfig {
  @Bean public TransferService transferService(AccountRepository repository) {
    return new TransferServiceImpl( repository );
  }
  @Bean public AccountRepository accountRepository(DataSource dataSource) {
    return new JdbcAccountRepository( dataSource );
  }
  @Bean public DataSource dataSource() {
    BasicDataSource dataSource = new BasicDataSource();
    ...
    return dataSource;
  }
}
```

Represents a dependency

3. Java Based Configuration

1. Java Based Configuration Concept
2. Application Context
3. Handling Multiple Configuration
4. Bean Scopes
5. Property Values
6. Spring Profiles
7. Coding Section: *Java Based Configuration*

Application Context

```
// Create application context from the configuration
ApplicationContext context =
    SpringApplication.run( ApplicationConfig.class );

// Look up a bean from the application context
TransferService service =
        context.getBean("transferService", TransferService.class);

// Use the bean
service.transfer(new MonetaryAmount("300.00"), "1", "2");
```

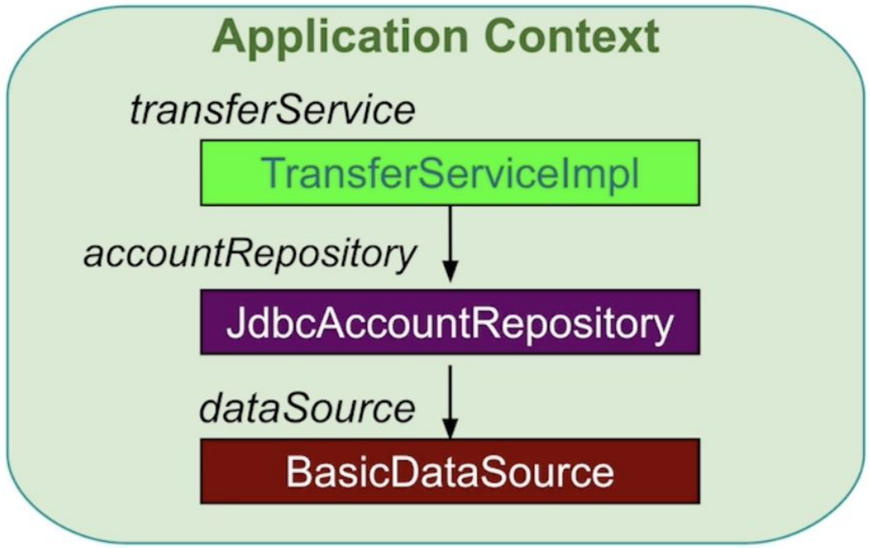
What configuration to use to define beans

Bean ID
Based in method name

use SpringApplication.run()

Inside the Spring ApplicationContext

```
// Create application context from the configuration
ApplicationContext context = SpringApplication.run( ApplicationConfig.class )
```



Application Context: Configuring with XML

```
// Create application context with ClassPathXmlApplicationContext
ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.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">

    <bean id="transferService" class="com.trendyol.bootcamp.service
        .TransferService"/>
</beans>
```

use ClassPathXmlApplicationContext with xml file

Accessing a Bean From Application Context

Multiple options

```
ApplicationContext context = SpringApplication.run(...);

// Use bean id, a cast is needed
TransferService ts1 = (TransferService)  context.getBean("transferService");

// Use typed method to avoid casting
TransferService ts2 =  context.getBean("transferService", TransferService.class);

// No need for bean id if type is unique - recommended (use type whenever possible)
TransferService ts3 =  context.getBean(TransferService.class );
```

cast because it is not type safe

a safe way with name and type

an ambiguity situation

Summary 

Spring manages your application objects

- ❑ Creating them in the correct dependency order
- ❑ Ensuring they are fully initialized before use

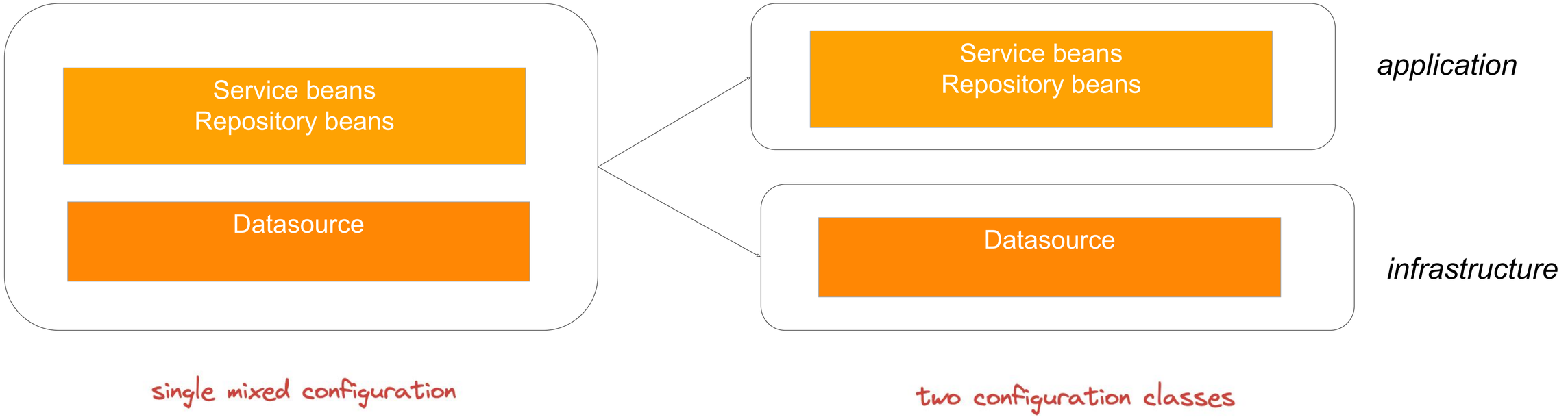
Ice Breaker 🥶

If you could travel anywhere in the world, where would you go?

3. Java Based Configuration

1. Java Based Configuration Concept
2. Application Context
3. Handling Multiple Configuration
4. Bean Scopes
5. Property Values
6. Spring Profiles
7. Coding Section: *Java Based Configuration*

Creating an Application Context From Multiple Files



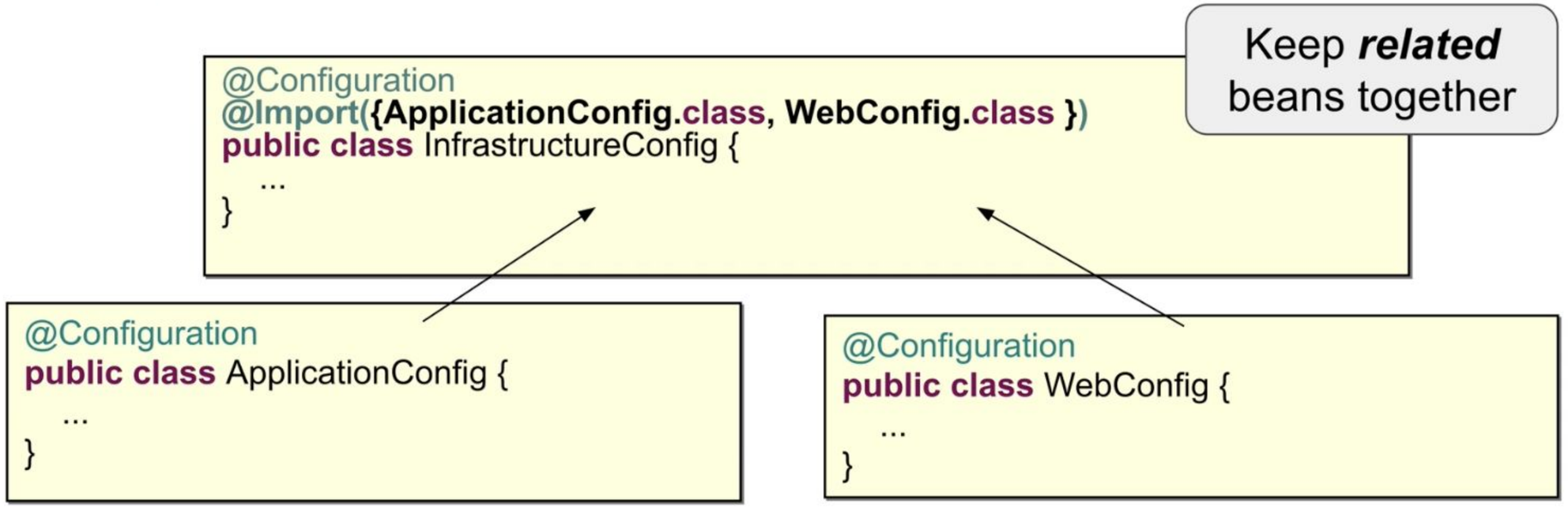
Separation of Concerns principle

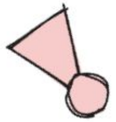
- ❏ Keep related beans in the same **@Configuration** file

Best Practice: separate “application” and “infrastructure”

- ❏ In real world, infrastructure often changes between environments

Handling Multiple Configurations





Files combined with @Import

Defines a single Application Context

Beans sourced from multiple files

trendyol learning

Mixed Configuration

```
@Configuration
public class ApplicationConfig {

   @Bean public TransferService transferService(AccountRepository repository)
      { return new TransferServiceImpl( repository ); }

   @Bean public AccountRepository accountRepository(DataSource dataSource)
      { return new JdbcAccountRepository( dataSource );  }

   @Bean public DataSource dataSource() {
      BasicDataSource dataSource = new BasicDataSource();
      dataSource.setDriverClassName("org.postgresql.Driver");
      dataSource.setUrl("jdbc:postgresql://localhost/transfer" );
      dataSource.setUsername("transfer-app");
      dataSource.setPassword("secret45" );
      return dataSource;
   }
}
```

application beans

infrastructure bean

Coupled to a local Postgres environment

```
@Configuration
public class ApplicationConfig {
   @Bean public TransferService transferService(AccountRepository repository) {
      return new TransferServiceImpl ( repository );
   }

   @Bean public AccountRepository accountRepository(DataSource dataSource) {
      return new JdbcAccountRepository( dataSource );
   }
}
```

```
@Configuration
@Import(ApplicationConfig.class )
public class TestInfrastructureConfig {
   @Bean public DataSource dataSource() {
      ...
   }
}
```

```
ApplicationContext ctx = SpringApplication.run( TestInfrastructureConfig.class )
```

trendyol learning

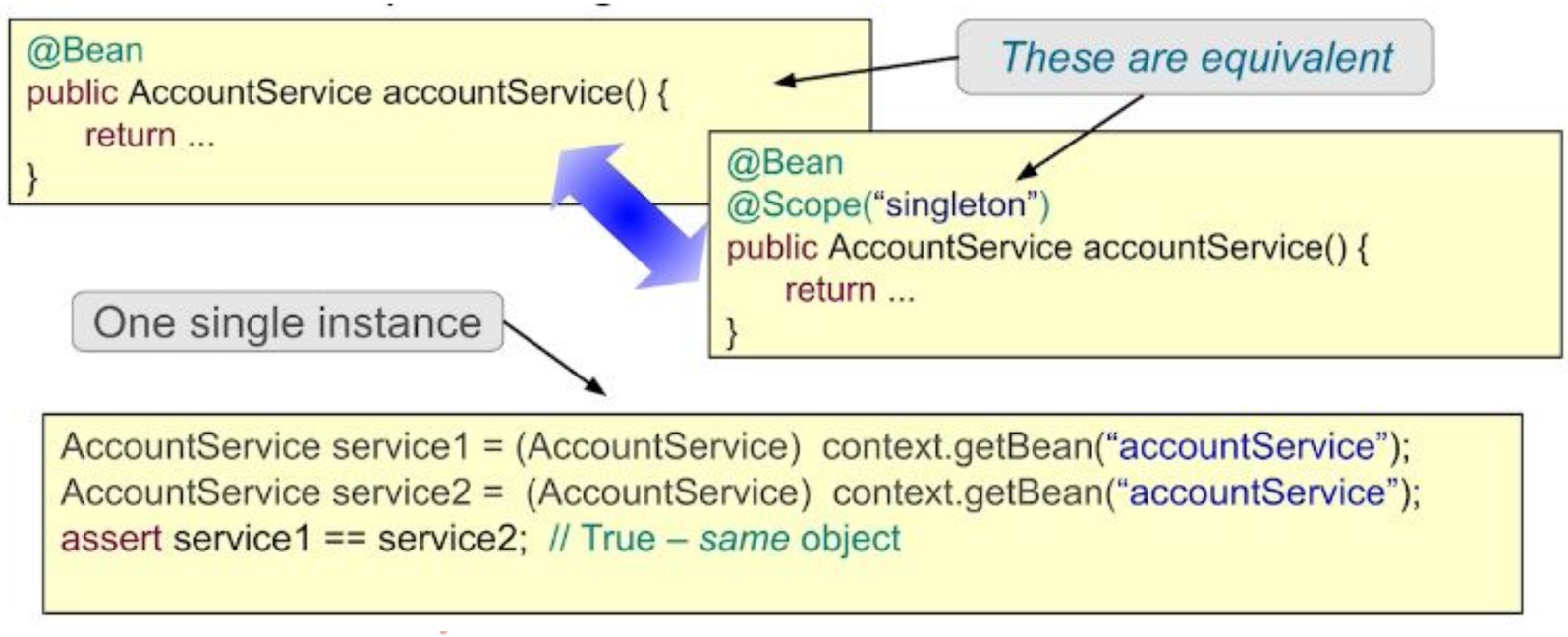
3. Java Based Configuration

1. Java Based Configuration Concept
2. Application Context
3. Handling Multiple Configuration
4. Bean Scopes
5. Property Values
6. Spring Profiles
7. Coding Section: *Java Based Configuration*

Bean Scope: *singleton*

The **scope** of a bean defines the life cycle and visibility of that bean in the contexts we use it.

- Default scope is *singleton.*
- **Singleton**: same instance used every time bean is referenced



Multiple requests problem:

Multiple threads accessing singleton beans at the same time

Bean Scope: *prototype*

- **Prototype**: new instance created every time bean is referenced

```
@Bean
@Scope("prototype")
public Action deviceAction() {
    return …
}
```

@Scope(scopeName="prototype")

```
Action action1 = (Action)  context.getBean("deviceAction");
Action action2 =  (Action)  context.getBean("deviceAction");
assert action1 != action2;  // True – different objects
```

TWO instances

trendyol learning

Common Spring Scopes

Singleton

A single instance is used

Prototype

A new instance is created each time the bean is referenced

Session

A new instance is created per user session

Request

A new instance is created once per request

web environment only

Summary 

- In Java configuration there are two important annotation: **@Configuration** and **@Bean**
- You should define beans in configuration files.
- You can use single configuration file or multiple configuration file.
- You can use multiple configuration files with **@Import** annotation**.**

- Dependency injection aim is to decrease coupling
- Components don’t need to find their dependencies
- Container will inject those dependencies for them

- Application context manages the lifecycle of components
- The default scope is **Singleton**. It creates one instance and uses it
- In **Prototype** scope, a new instance is created every time

trendyol.com

3. Java Based Configuration

1. Java Based Configuration Concept
2. Application Context
3. Handling Multiple Configuration
4. Bean Scopes
5. Property Values
6. Spring Profiles
7. Coding Section: *Java Based Configuration*

Setting Property Values

Hard-coding these properties is a Bad practice

- ❏ Better practice is to “externalize” these properties.
- ❏ One way to “externalize” them is by using property files.

Environment bean represents loaded properties from runtime environment.
Properties derived from various sources:

- ❏ JVM System properties → **System.getProperty()**
- ❏ System Environment Variables → **System.getenv()**
- ❏ Java Properties Files


```
@Bean
public DataSource dataSource() {
  BasicDataSource ds = new BasicDataSource();
  ds.setDriverClassName("org.postgresql.Driver");
  ds.setUrl("jdbc:postgresql://localhost/transfer" );
  ds.setUser("transfer-app");
  ds.setPassword("secret45" );
  return ds;
}
```

Spring Environment Abstraction

Inject **Environment**

- ❏ package: *org.springframework.core.env*
- ❏ Call *getProperty()* method
- ❏ If you want to change any property value, change it from application.yml file

```
@Configuration
public class DbConfig {

    @Bean
    public DataSource dataSource(Environment env) {
        BasicDataSource ds = new BasicDataSource();
        ds.setDriverClassName(env.getProperty("spring.datasource.driver"));
        ds.setUrl(env.getProperty("spring.datasource.url"));
        ds.setUser(env.getProperty("spring.datasource.username"));
        ds.setPassword(env.getProperty("spring.datasource.password"));
        return ds;
    }

}
```

Inject Environment bean like any other Spring bean

```
spring:
  datasource:
    driver: org.postgresql.Driver
    url: jdbc:postgresql://localhost/transfer
    username: transfer-app
    password: secret45
```

application.yml file

Property Source

Environment bean obtains values from “property sources”

- ❏ *Environment variables* and *Java System Properties* always populated automatically.
- ❏ **@PropertySource** contributes additional properties.
- ❏ Available resource prefixes: **classpath: file: http:**

```
@Configuration
@PropertySource (“classpath:/com/trendyol/bootcamp/config/app.properties”)
@PropertySource (“file:config/local.properties”)
public class ApplicationConfig {

  ...
}
```

Add properties to these files in addition to environment variables and system properties

Accessing Properties Using @Value

use **@Value** annotation

- ❏ *package*: org.springframework.beans.factory.annotation
- ❏ No need to reference Environment
- ❏ 💥The *most* used way in Trendyol for accessing properties

```
@Configuration
   public class DbConfig {

      @Value("${spring.datasource.driver}")
       private String driver;

      @Value("${spring.datasource.url}")
       private String url;

      @Value("${spring.datasource.username}")
       private String username;

      @Value("${spring.datasource.password}")
       private String password;

      @Bean
      public DataSource dataSource() {
         BasicDataSource ds = new BasicDataSource();
         ds.setDriverClassName(driver);
         ds.setUrl(url);
         ds.setUser(username);
         ds.setPassword(password);
         return ds;
      }
}
```

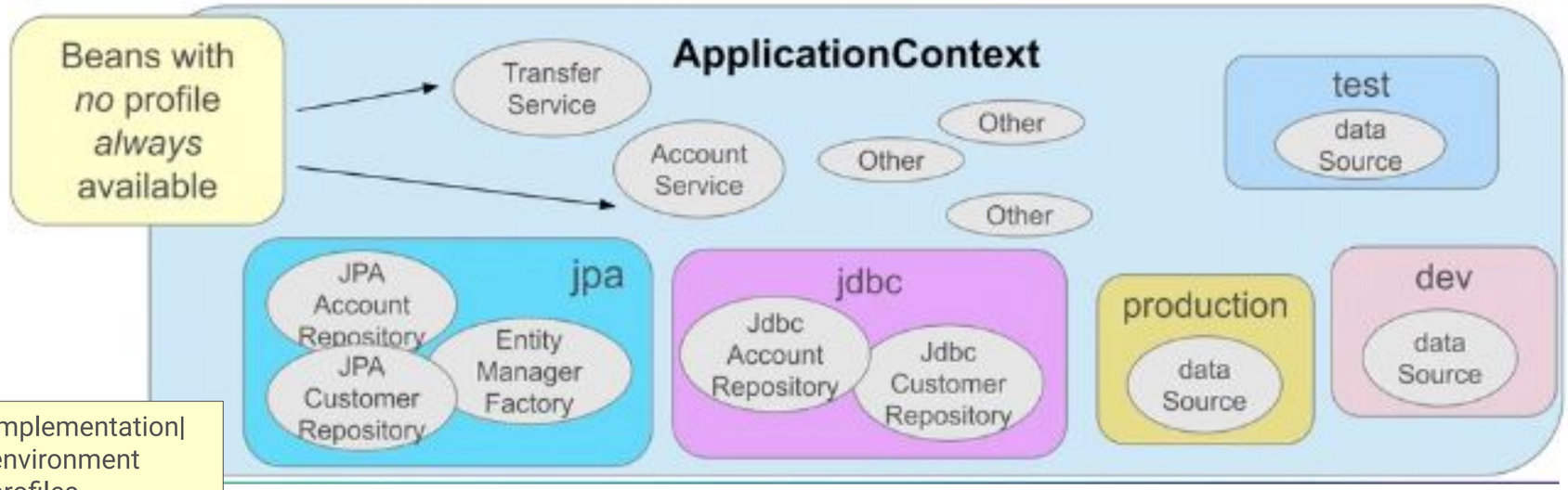
3. Java Based Configuration

1. Java Based Configuration Concept
2. Application Context
3. Handling Multiple Configuration
4. Bean Scopes
5. Property Values
6. Spring Profiles
7. Coding Section: *Java Based Configuration*

Spring Profiles

Profiles can represent

- ❏ Environment: *dev, test, production*
- ❏ Implementation: *jdbc, jpa*
- ❏ Deployment platform: *on-premise, cloud*
- ❏ Beans included / excluded based on profile membership



implementation| environment profiles

trendyol learning

Defining Profiles

Using **@Profile** annotation on configuration class

- org.springframework.context.annotation
- Everything in Configuration belong to the profile
- If Profile is not activated, bean is not initialized

```
@Configuration
@Profile("embedded")
public class DevConfig {

   @Bean
   public DataSource dataSource() {
      EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
      return builder.setName("testdb")
                  .setType(EmbeddedDatabaseType.HSQL)
                  .addScript("classpath:/testdb/schema.db")
                  .addScript("classpath:/testdb/test-data.db").build();
   }
   ...
```

Nothing in this configuration will be used *unless* "embedded" profile is chosen as one of the active profiles

H2, Derby are also supported

Defining Profiles

You can use **@Profile** annotation on **@Bean** methods
You can use exclamation **!** with **@Profile**

```
@Configuration
@Profile("cloud")
public class DevConfig {
    …
}
```

If *cloud* is **active** profile

```
@Configuration
@Profile("!cloud")
public class ProdConfig {
    …
}
```

If *cloud* is **inactive** profile

Not cloud – use exclamation !

```
@Configuration
public class DataSourceConfig {

    @Bean(name="dataSource")
    @Profile("embedded")
    public DataSource dataSourceForDev() {
        EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
        return builder.setName("testdb") ...
    }

    @Bean(name="dataSource")
    @Profile("!embedded")
    public DataSource dataSourceForProd() {
        BasicDataSource dataSource = new BasicDataSource();
        ...
        return dataSource;
    }
}
```

Explicit bean-name overrides method name

Both profiles define *same* bean name, so only *one* profile should be activated at a time.

Ways to Activate Profiles

Profiles must be activated at run-time

- System property via command-line

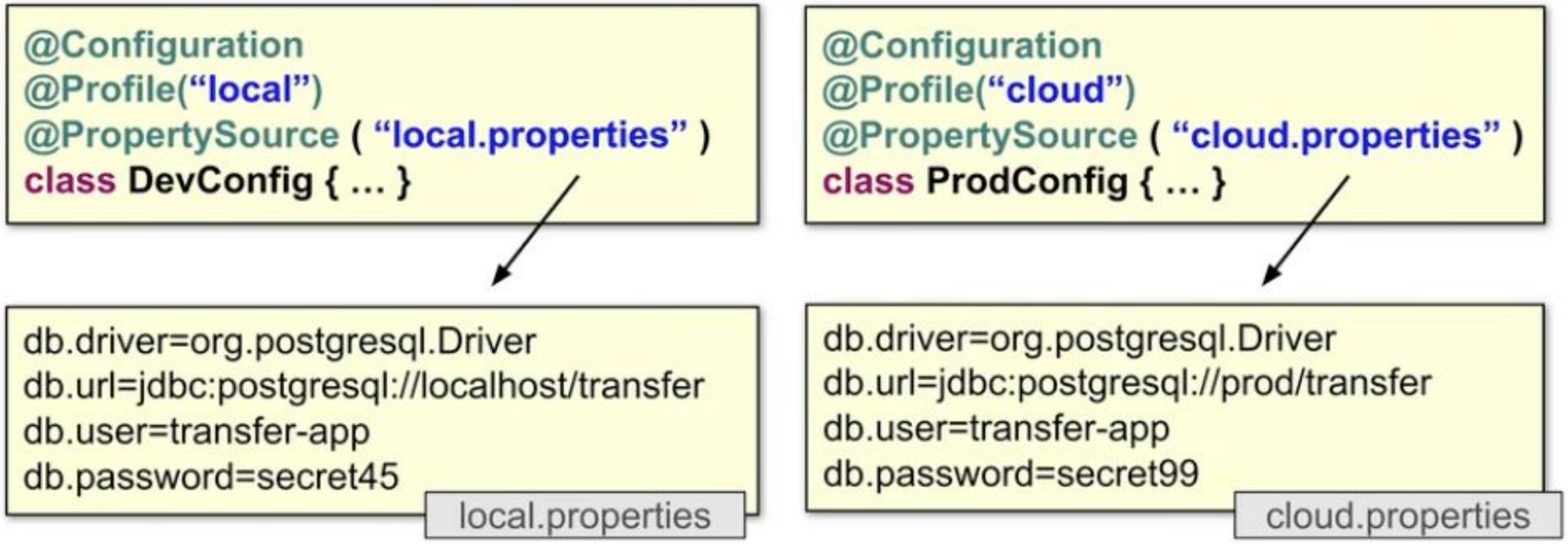
```
-Dspring.profiles.active=embedded,jpa
```

- System property programmatically

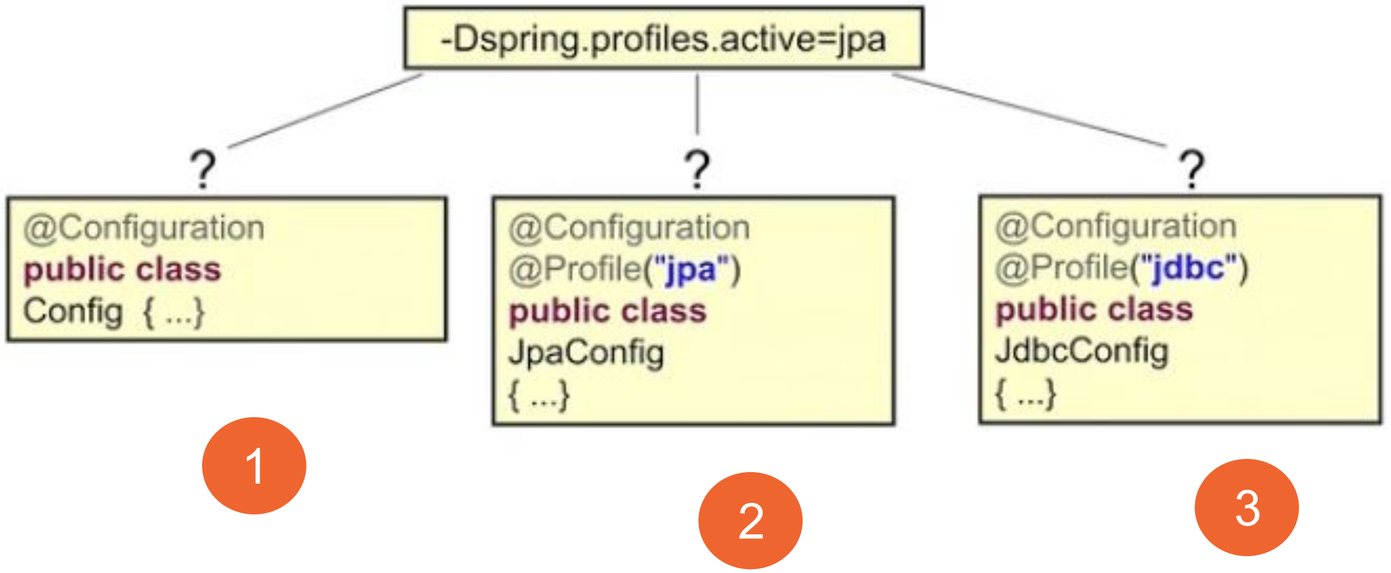
```
System.setProperty("spring.profiles.active", "embedded,jpa");
SpringApplication.run(AppConfig.class);
```

Property Source Selection

@Profile can control which **@PropertySource** are included in the Environment



Question



trendyol learning

3. Java Based Configuration

1. Java Based Configuration Concept
2. Application Context
3. Handling Multiple Configuration
4. Bean Scopes
5. Property Values
6. Spring Profiles
7. Lab Section: *Java Based Configuration*

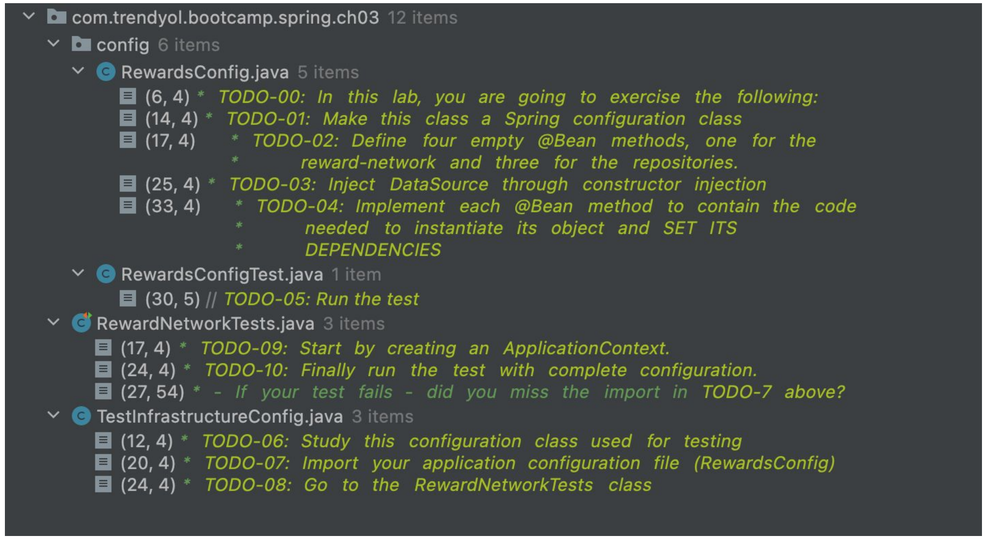
Lab Section

What we will learn:

1. How to create a Spring ApplicationContext and get a bean from it
2. Spring Java configuration syntax

Todos:

1. Please, Switch branch to `feature/java-based-configuration`
2. There are **12 TODOs** in the project files. Look at these TODOs
3. Lets try to do each TODO together



4. Annotation Based Configuration

1. Annotation Based Configuration Concept
2. Usage of Autowired
3. Component Scan
4. Lifecycle Annotations
5. Stereotype Annotations
6. Lab Section: *Annotation Based Configuration*

Annotation Based Configuration Concept

Configuration is external to bean-class
- Java based configuration
- Explicit configuration

```
@Configuration
public class TransferModuleConfig {

    @Bean public TransferService transferService() {
        return new TransferServiceImpl( accountRepository() );
    }

    @Bean public AccountRepository accountRepository() {
        ...
    }
}
```

Dependency Injection

java based configuration

Configuration is *within* the bean-class, embedded to class
- Annotation based configuration
- Implicit configuration
- Component scanning

Annotations embedded with POJOs

```
@Component
public class TransferServiceImpl implements TransferService {

    public TransferServiceImpl(AccountRepository repo) {
        this.accountRepository = repo;
    }
}
```

Bean id/name derived from classname: *transferServiceImpl*

```
@Configuration
@ComponentScan ( "com.bank" )
public class AnnotationConfig {
   // No bean definition needed any more
}
```

Find @Component annotated classes within designated (sub)packages

annotation based configuration

4. Annotation Based Configuration

1. Annotation Based Configuration Concept
2. Usage of Autowired
3. Component Scan
4. Lifecycle Annotations
5. Stereotype Annotations
6. Lab Section: *Annotation Based Configuration*

Usage of @Autowired

- Spring can resolve collaborators automatically by inspecting the content of the ApplicationContext.
- This is called autowiring.
- Autowiring allows cleaner DI management.

constructor injection

```
@Autowired  // Optional if this is the only constructor
public TransferServiceImpl(AccountRepository a) {
   this.accountRepository = a;
}
```

method injection

```
@Autowired
public void setAccountRepository(AccountRepository a) {
   this.accountRepository = a;
}
```

field injection

```
@Autowired
private AccountRepository accountRepository;
```

Required or Optional Autowired

- Default behavior: *required*

```
@Autowired
public void setAccountRepository(AccountRepository a) {
  this.accountRepository = a;
}
```

Exception if no dependency found

→ org.springframework.beans.factory.BeanInitializationException

- Use required=false attribute to override default behavior

```
@Autowired(required=false)
public void setAccountRepository(AccountRepository a) {
  this.accountRepository = a;
}
```

Only inject *if* dependency exists

- Another way to inject optional dependencies

```
@Autowired
public void setAccountService(Optional<AccountService>
                    accountService){
  this.accountService = accountService;
}
```

→

```
@Autowired
public void setAccountService(Optional<AccountService>
                    accountService){
  this.accountService = accountService;
}

public void doSomething() {
   accountService.ifPresent( s ->  {
      // s is the AccountService instance, use s to do something
   });
}
```

Note the use of the Lambda

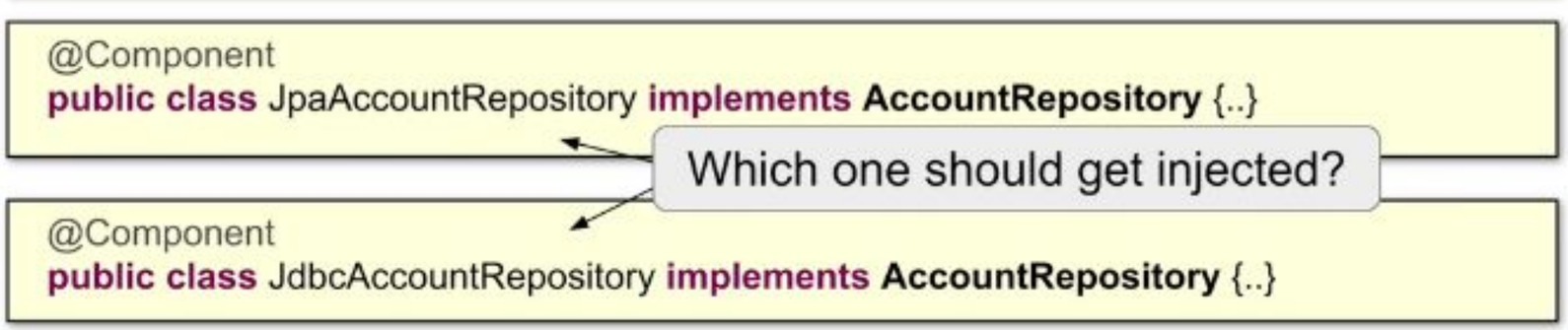
trendyol learning

Constructor vs Setter (Method) Dependency Injection

Constructor	Setter (Method)
Required dependencies	Inherited automatically
Dependencies can be immutable	Dependencies are mutable
Passing several params at once	Could be verbose for several params

🔔 Constructor injection is generally preferred

Autowiring and Disambiguation

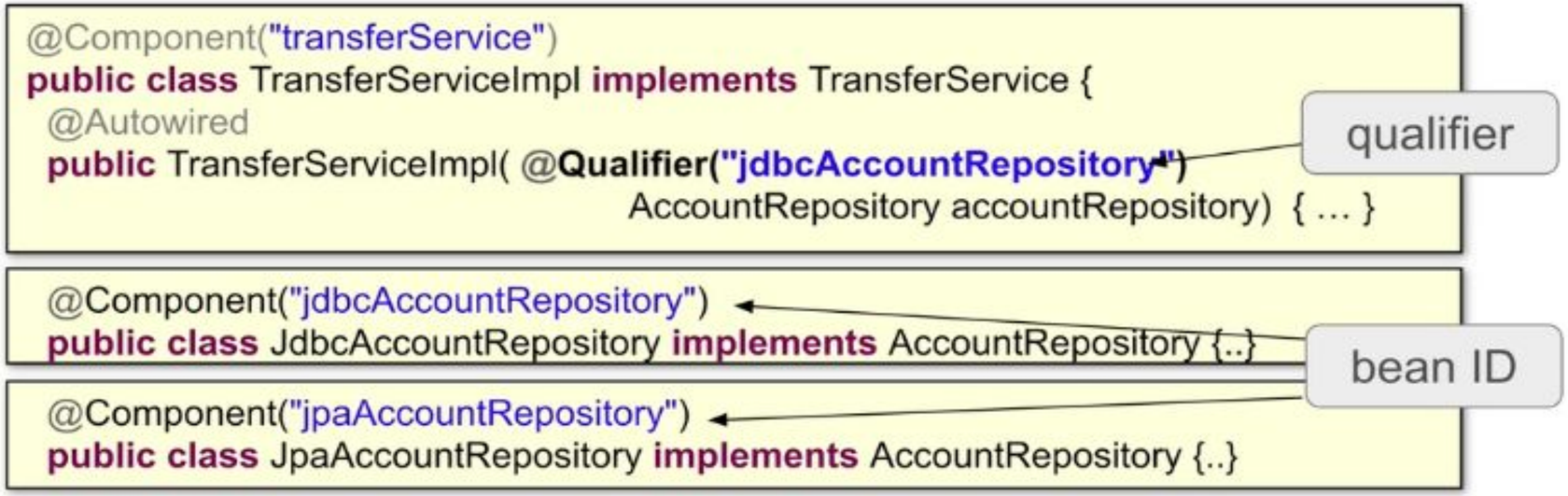



```
@Component
public class TransferServiceImpl implements TransferService {
  @Autowired // optional
  public TransferServiceImpl(AccountRepository accountRepository)  { … }
}
```

At startup: *NoSuchBeanDefinitionException*, no unique bean of type [AccountRepository] is defined: expected single bean but found 2...

🔔 **@Autowired** does autowiring by type.

Using Qualifier



What if we didn't specify Bean's name using @Component("jdbcAccountRepository") or @Component("jpaAccountRepository")?

- Names are auto-generated
- Take class name and put first letter in lower case. e.g : **JdbcAccountRepository** → **jdbcAccountRepository**
- *Recommendation*: never rely on generated names
- Common strategy: avoid using qualifiers and don't use 2 beans of same type in ApplicationContext

Delayed Initialization with @Lazy

Beans normally created on startup when application context created.

- ❏ As we learn, Spring loads all Singleton beans *eagerly* in default.
- ❏ Prototype beans are loaded only when they are asked for from the context.
- ❏ Bootstrap of the application takes time for singleton beans eagerly load.

@Lazy is used to specify that singleton beans should be loaded lazily.
Useful if bean's dependencies *not* available at startup

Lazy beans created first time used

- ❏ When dependency injected
- ❏ By ApplicationContext.getBean methods invoked

```
@Lazy @Component
public class MailService {
    public MailService(@Value("smtp:...") String url) {
        // connect to mail-server
    }
    ...
}
```

SMTP server may not be running when this process starts up

Autowiring Constructors

If a class *only* has a default constructor
- ❏ Nothing to annotate

If a class only has one non-default constructor
- ❏ It is the only constructor available, Spring will call it
- ❏ **@Autowired** is optional

If a class has *more than one* constructor
- ❏ Spring invokes zero-argument constructor by default
- ❏ Or you must annotate with **@Autowired** the one you want Spring to use

4. Annotation Based Configuration

1. Annotation Based Configuration Concept
2. Usage of Autowired
3. Component Scan
4. Lifecycle Annotations
5. Stereotype Annotations
6. Lab Section: *Annotation Based Configuration*

About Component Scanning

Components are scanned at startup

- ❏ JAR dependencies also scanned

@ComponentScan: org.springframework.context.annotation.ComponentScan

- ❏ This annotation provides component scanning directive to use with all @Component classes.

! Could result in slower startup time if too many files scanned

What are the best practices?

Component Scanning: Best Practices

@ComponentScan({“org”, “com”})

@ComponentScan({“com”})

@ComponentScan({“com.trendyol.bootcamp”})

@ComponentScan({“com.trendyol.bootcamp.repository”, “com.trendyol.bootcamp.service”, “com.trendyol.bootcamp.controller”})

4. Annotation Based Configuration

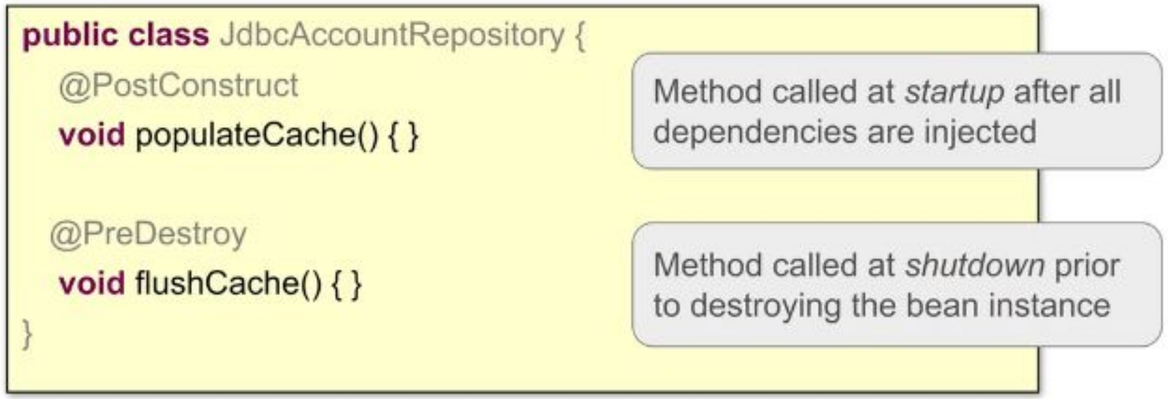
1. Annotation Based Configuration Concept
2. Usage of Autowired
3. Component Scan
4. Lifecycle Annotations
5. Stereotype Annotations
6. Lab Section: *Annotation Based Configuration*

Lifecycle Annotations

2 Annotations:

- ❏ **@PostConstruct** → *package* javax.annotation, called in *startup*
- ❏ **@PreDestroy** → *package* javax.annotation, called in *shutdown*

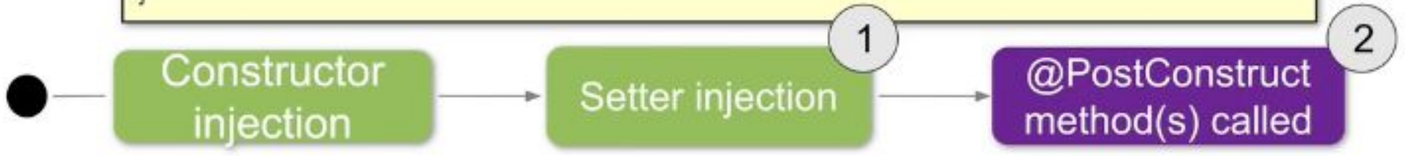
! ❏ Annotated methods can have any visibility but must take no parameters and only return void.



@PostConstruct

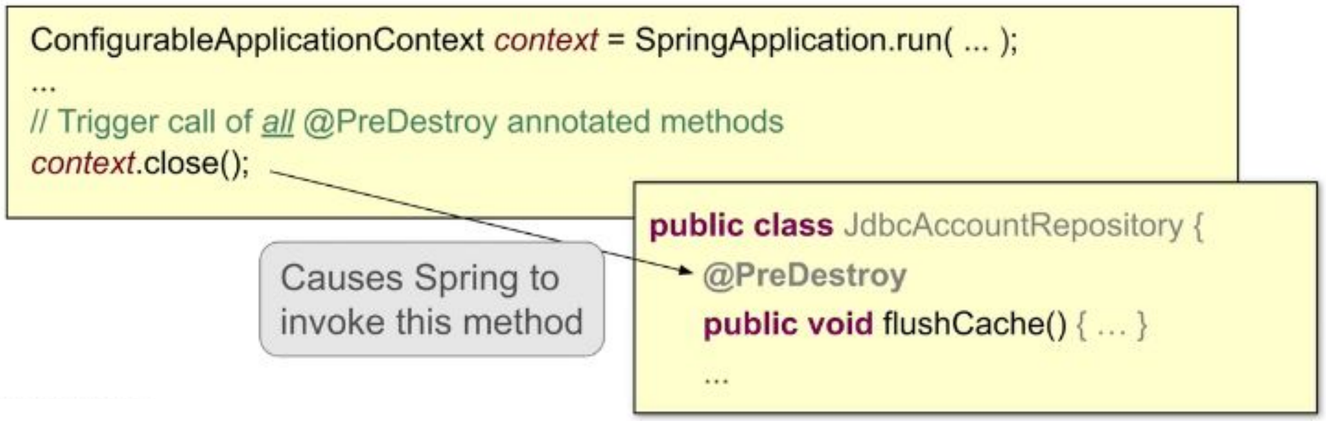
- Only one method in a given class can be annotated with *@PostConstruct*
- PostConstruct method can be public, protected, private

```
public class JdbcAccountRepository {
    private DataSource dataSource;
    @Autowired                                          1
    public void setDataSource(DataSource dataSource)
      {  this.dataSource = dataSource;   }

    @PostConstruct
    public void populateCache()                         2
      {   Connection conn = dataSource.getConnection(); //...   }
}
```



@PreDestroy

- ❏ Useful for releasing resources & cleaning up purpose
- ❏ PreDestroy method can be public, protected, private
- ❏ Not called for prototype beans
- ❏ PreDestroy methods called if application shuts down normally. (*not process dies or is killed*) !

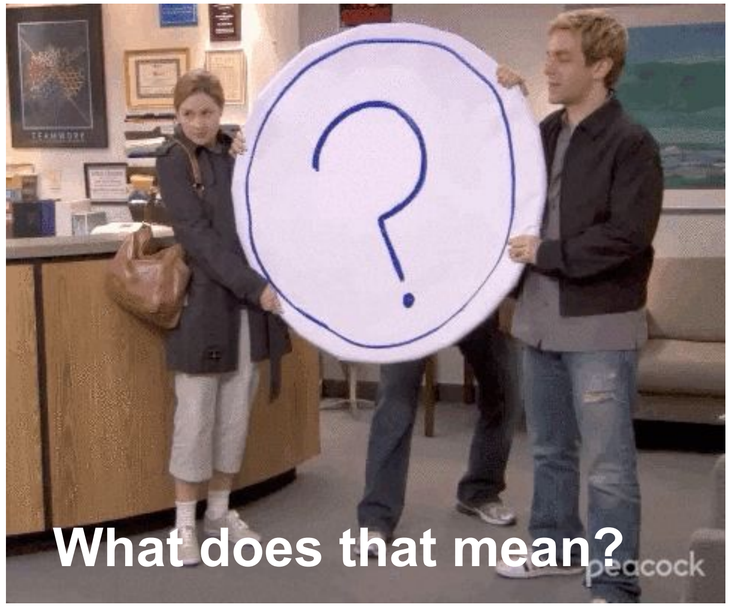


4. Annotation Based Configuration

1. Annotation Based Configuration Concept
2. Usage of Autowired
3. Component Scan
4. Lifecycle Annotations
5. Stereotype Annotations
6. Lab Section: *Annotation Based Configuration*

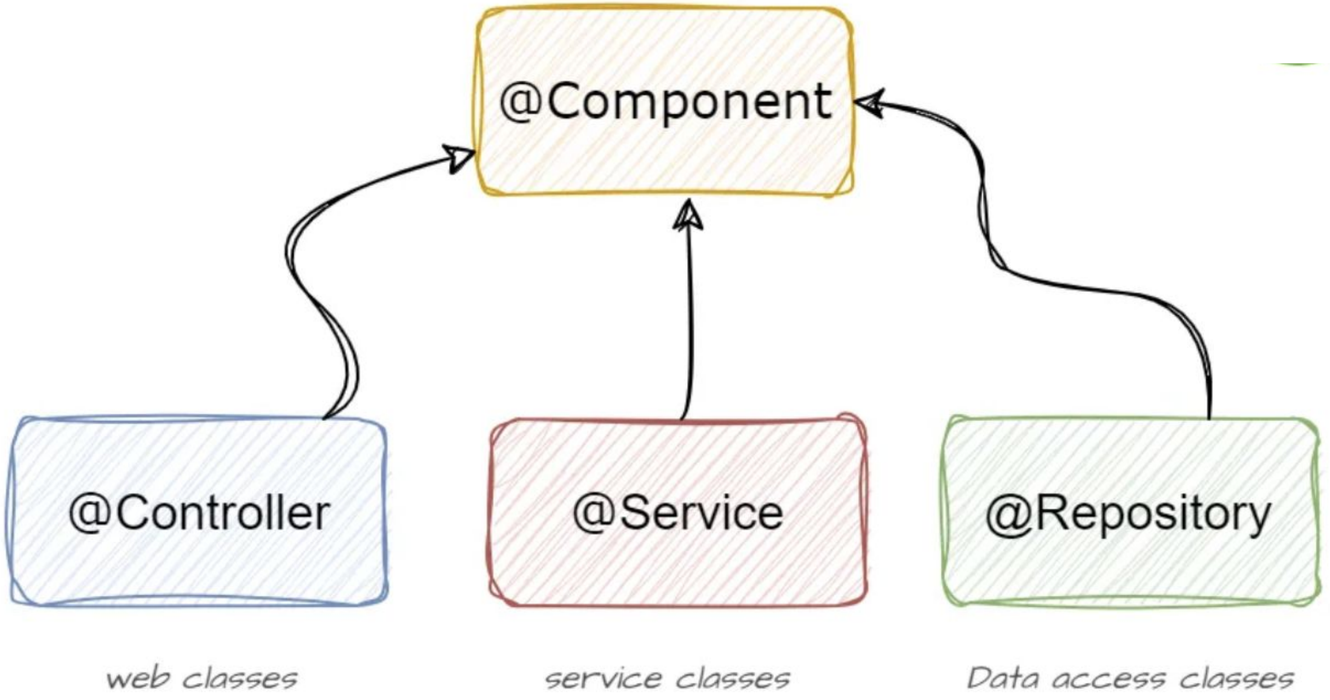
Stereotype Annotations

Recall: ComponentScan scans annotations @Component
- It also scans annotations which are annotated by @Component.



With Spring 2.5, published a new package which name is stereotype it holds annotations which are annotated by *@Component*.

Stereotype Annotations: @Service, @Controller, @Repository



Let's look at these Annotations together

Summary 

Spring beans can be defined:

- ❏ Explicitly using **@Bean** methods inside **@Configuration** class (*Java-based configuration*)
- ❏ Implicitly using **@Component** and component scanning (*Annotation-based configuration*)

Applications can use both

- ❏ Implicit for your classes - annotation based
- ❏ Explicit for the rest - *for large applications* - java based

Can perform initialization and clean-up

- ❏ Use **@PostConstruct** and **@PreDestroy**

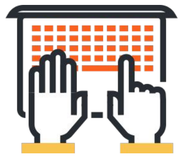
With using Spring's stereotypes annotation, you can create custom annotations.

- ❏ @Service, @Repository, @Controller

4. Annotation Based Configuration

1. Annotation Based Configuration Concept
2. Usage of Autowired
3. Component Scan
4. Lifecycle Annotations
5. Stereotype Annotations
6. Coding Section: *Annotation Based Configuration*

Coding Section



What you will learn:

1. How to use component scanning with annotations
2. Component scanning in spring
3. How to implement your own bean lifecycle behaviors

Todos:

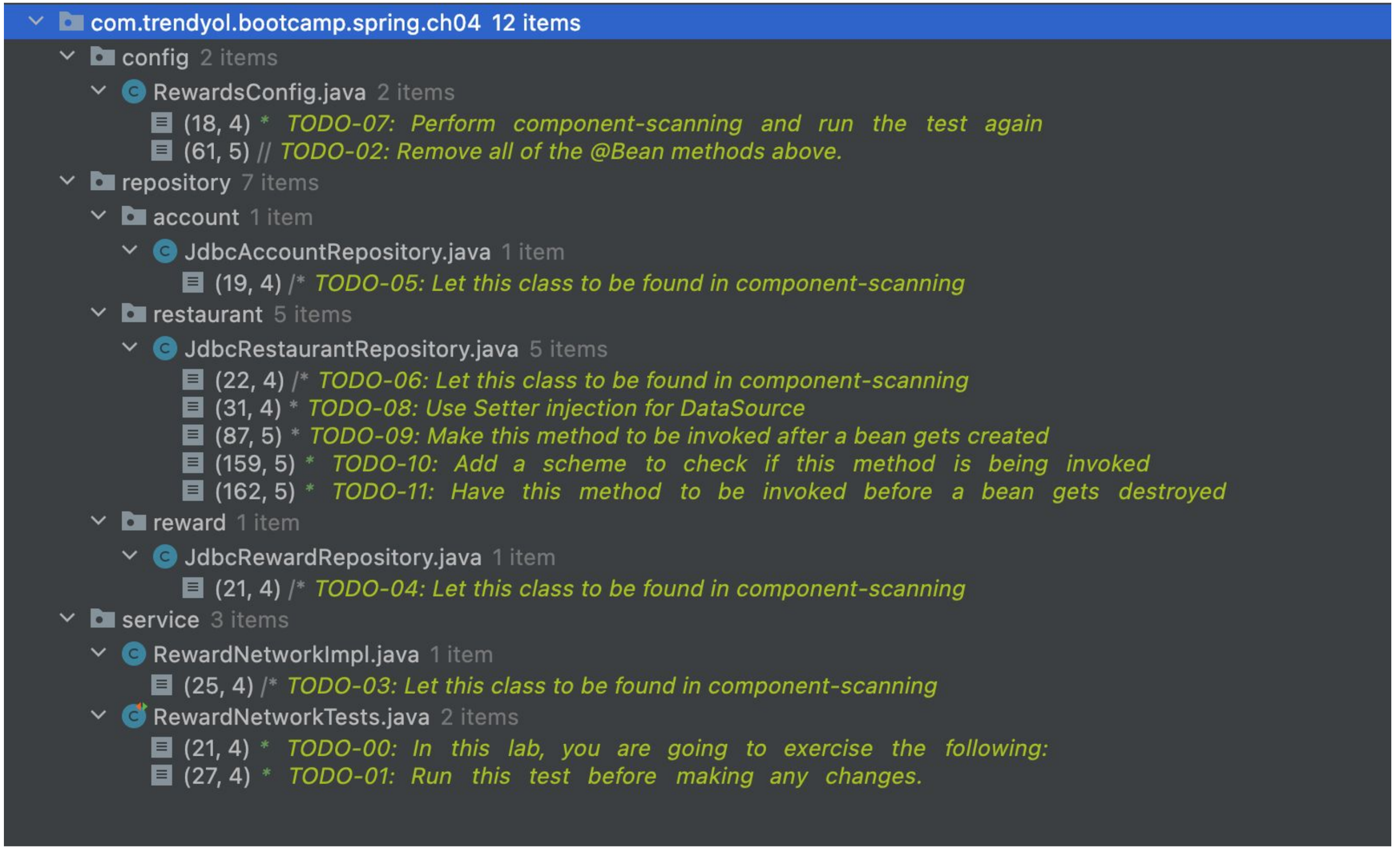
1. Clone project from github, if you did not clone before:
2. Switch branch to feature/annotation-based-configuration
3. Create a new branch from this branch, your new branch name should be feature/annotation-based-configuration-lab
4. There are 12 TODOs in the project files. Look at these TODOs
5. Please try to do each TODO
6. Please make sure tests are success.
7. Please add the changes and push the solution code in your github repository.

Repository: https://github.com/gulumseraslann/spring-training/tree/feature/annotation-based-configuration
clone repository:
git clone https://github.com/gulumseraslann/spring-training.git

switch branches:
git checkout feature/annotation-based-configuration
git checkout -b feature/annotation-based-configuration-lab

push:
git add .
git commit -m “implemented annotation-based-configuration lab section”
git push

Coding Section



5. Aspect Oriented Programming

1. What Is Cross-Cutting Concern & Aspect Oriented Programming?
2. What Problems Does AOP Solve?
3. AOP Concepts
4. Pointcut Expressions
5. Advice Types
6. Lab Section: *Aspect Oriented Programming*

What is Cross-Cutting Concern & Aspect Oriented Programming?

Cross-Cutting Concern

- *Generic functionality that is needed in many places in your application.*
- Logging, tracing, caching, error handling etc.

Aspect Oriented Programming

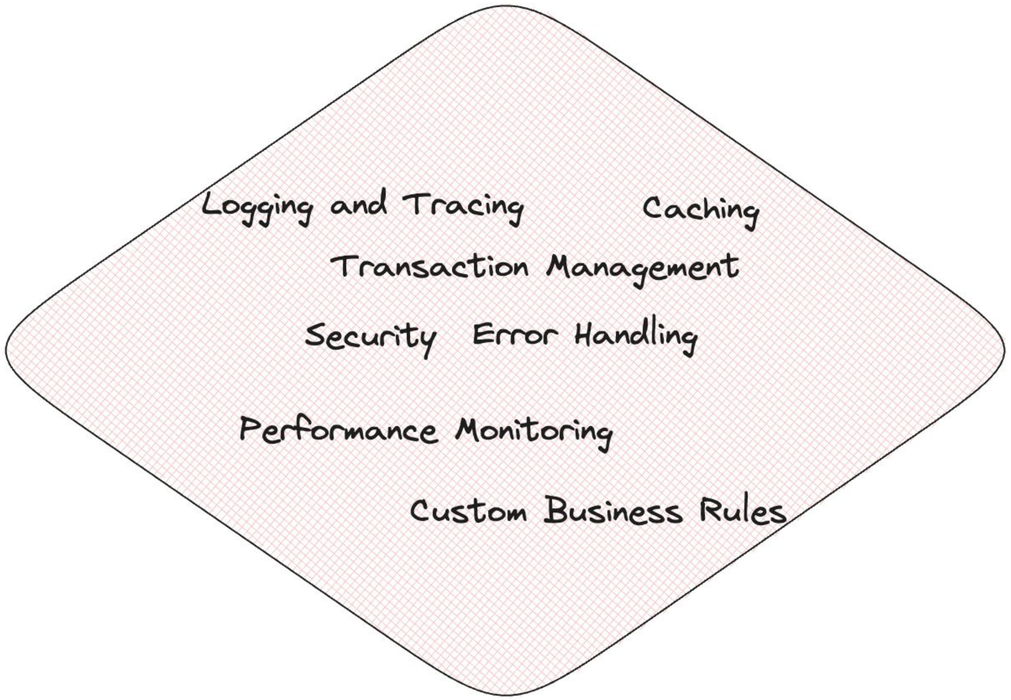
- *A programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns.*
- enables modularization of cross-cutting concerns.
- makes code clean and simpler.
- is shortly AOP

5. Aspect Oriented Programming

1. What Is Aspect Oriented Programming?
2. What Problems Does AOP Solve?
3. AOP Concepts
4. Pointcut Expressions
5. Advice Types
6. Lab Section: *Aspect Oriented Programming*

What Problem Does AOP Solve?

- ❑ AOP separates core business functionality.
- ❑ Allows us to be able to divide a bigger problem into smaller pieces
- ❑ Provide separation of concerns principle.
- ❑ Enables modularization of *cross-cutting concerns*



An Example of Cross Cutting Concern

Perform a role-based security check before every application method

A sign this requirement is a cross-cutting-concern

2 PROBLEMS:

- Code Complexity
- Code Duplication

First Problem: *Code Complexity*

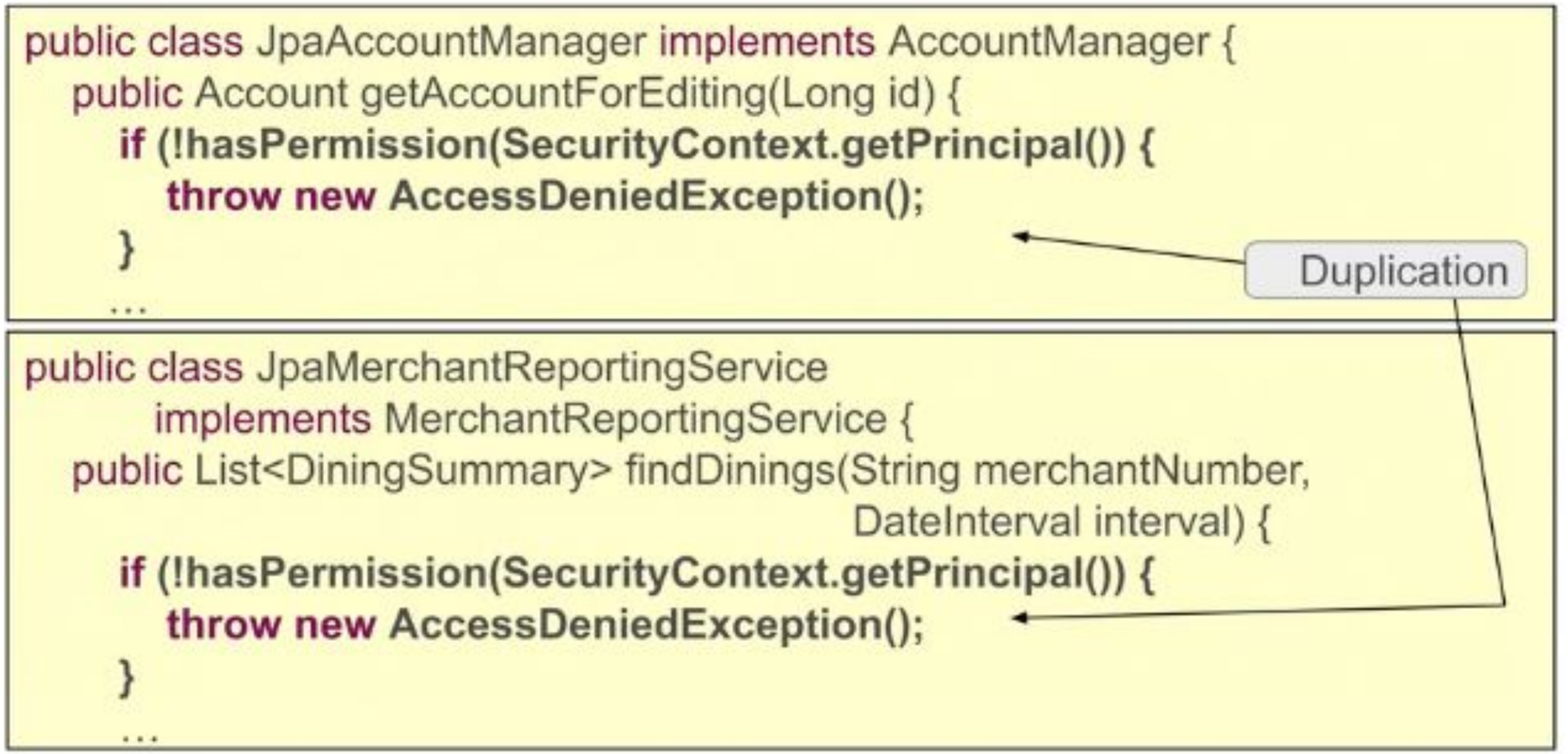
```
public class RewardNetworkImpl implements RewardNetwork {
    public RewardConfirmation rewardAccountFor(Dining dining) {

        // Security-related code
        if (!hasPermission(SecurityContext.getPrincipal()) {
            throw new AccessDeniedException();
        }

        // Application code
        Account a = accountRepository.findByCreditCard(…
        Restaurant r = restaurantRepository.findByMerchantNumber(…
        MonetaryAmount amt = r.calculateBenefitFor(account, dining);
        …
    }
}
```

Mixing of concerns

- Security and Business logic together
- Hard to test

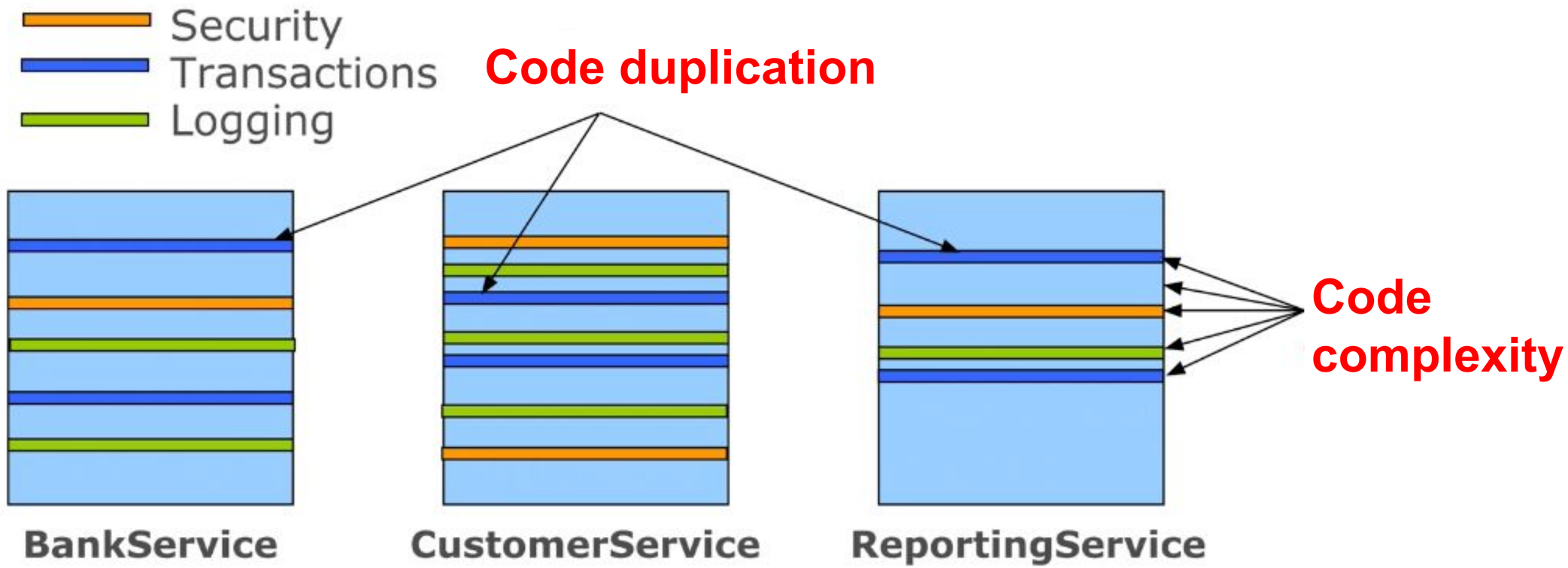
trendyol learning

Second Problem: *Code Duplication*



- Copy and paste the same code
- Must change it every where

System Design Without Modularization



System Design Solution

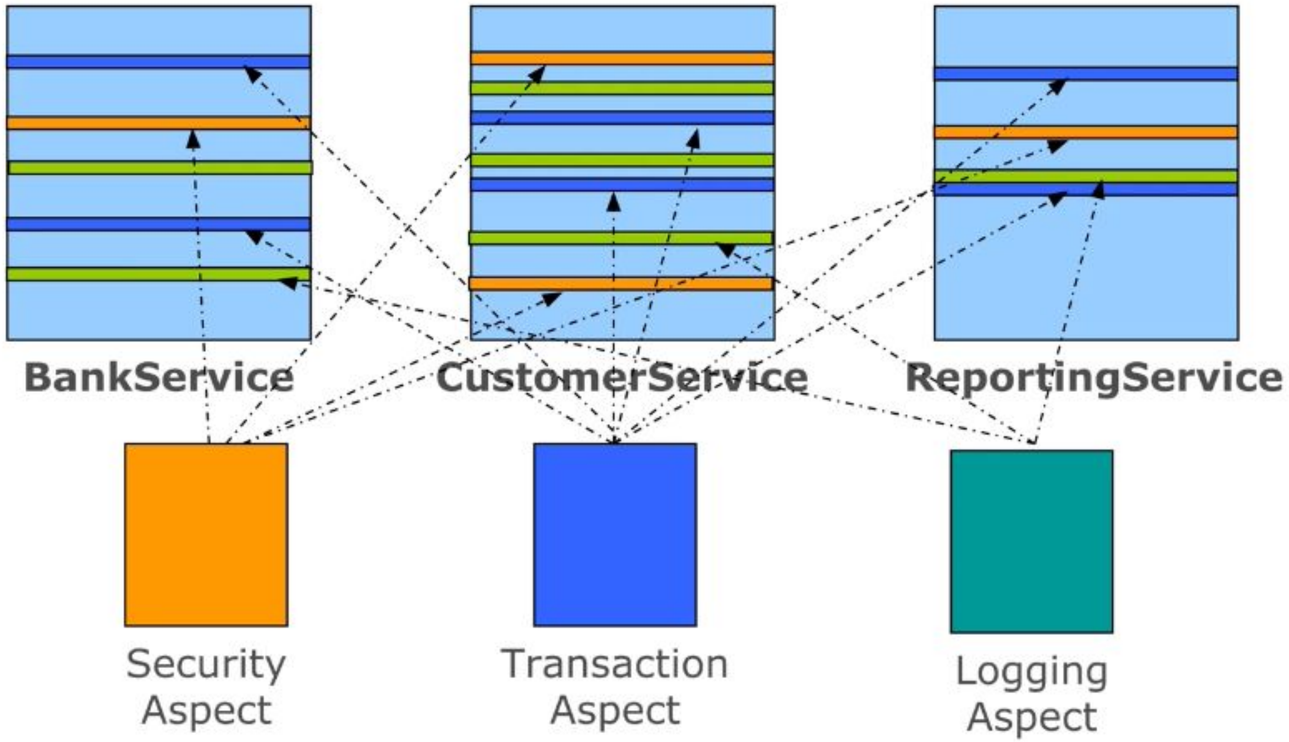
Implement your main application logic

Write Aspects to implement cross cutting concerns

Use aspects in your application

System Design With AOP

🔔 Now, main logic and concerns are *independent*

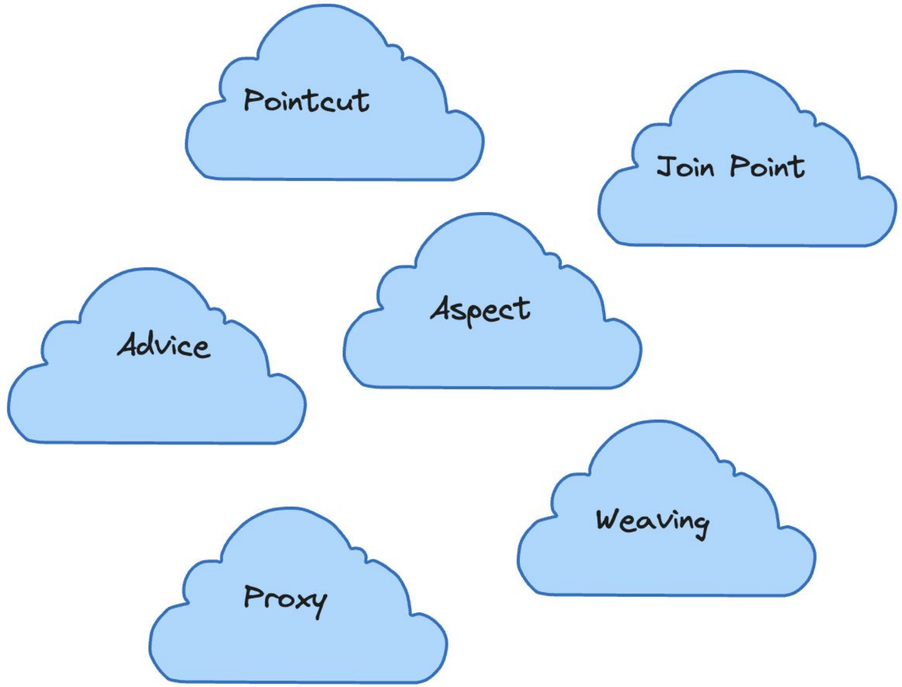


5. Aspect Oriented Programming

1. What Is Aspect Oriented Programming?
2. What Problems Does AOP Solve?
3. AOP Concepts
4. Pointcut Expressions
5. Advice Types
6. Lab Section: *Aspect Oriented Programming*

AOP Concepts

- **JoinPoint:** A point in the execution of a program such as a method call or exception thrown.
- **Pointcut**: An expression that selects one or more JoinPoint.
- **Advice**: A specific code executed at a certain join point.
- **Aspect**: A module that encapsulates Pointcuts + Advices
- **Weaving**: Technique by which aspects are combined with main code
- **Proxy:** An “enhanced” class that stands in place of your original, with extra behavior



5. Aspect Oriented Programming

1. What Is Aspect Oriented Programming?
2. What Problems Does AOP Solve?
3. AOP Concepts
4. Pointcut Expressions
5. Advice Types
6. Lab Section: *Aspect Oriented Programming*

Pointcut Expressions

🔔 *Recall*: An expression that selects one or more JoinPoint. (*where to apply Advice*)

execution(<*method pattern*>)

- ❏ Method must match the pattern.

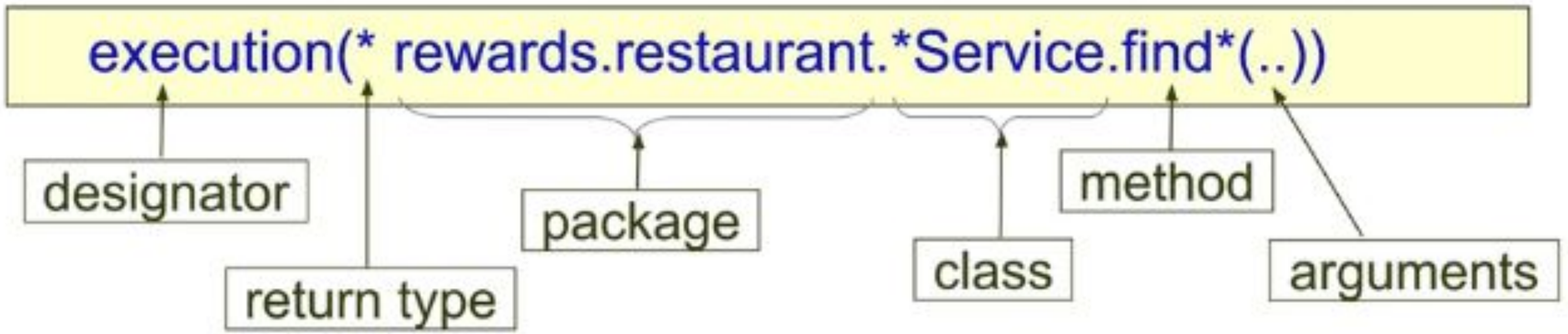
Method Pattern

- ❏ [Modifiers] ReturnType* [Class Type] MethodName(Arguments)* [throw ExceptionType]
- ❏ ReturnType and MethodName(Arguments) are mandatory.

Can chain together to create composite pointcuts

- ❏ and: **&&**
- ❏ or: **||**
- ❏ not: **!**
- ❏ execution(<*method pattern 1*>) || execution(<*method pattern 2*>)

Example Expression



Any method
--> starts with find with zero or more arguments
--> ends with Service
--> in a package with rewards.restaurant
--> any return type

✔ Pattern: [Modifiers] ReturnType [Class Type] MethodName(Arguments) [throw ExceptionType]

✔ * : matches once (ReturnType, Package, class, MethodName, Argument)

✔ .. : matches zero or more (Argument, Package)

trendyol learning

Example Expression: *Any Class or Package*

```
execution(void send*(rewards.Dining))
```

Any method
--> starts with send with argument type rewards.Dining
--> returns void

❗ Use fully-qualifed class name.

✔ Pattern: [Modifiers] ReturnType [Class Type] MethodName(Arguments) [throw ExceptionType]

✔ * : matches once (ReturnType, Package, class, MethodName, Argument)

✔ .. : matches zero or more (Argument, Package)

Example Expression: *Any Class or Package*

```
execution(* send(*))
```

Any method
--> which name is send and takes a single parameter
--> any return type

✔ Pattern: [Modifiers] ReturnType [Class Type] MethodName(Arguments) [throw ExceptionType]

✔ * : matches once (ReturnType, Package, class, MethodName, Argument)

✔ .. : matches zero or more (Argument, Package)

Example Expression: *Any Class or Package*

```
execution(* send(int, ..))
```

Any method
--> which name is send and takes a first argument is an int type and zero or more argument
--> any return type

✔ Pattern: [Modifiers] ReturnType [Class Type] MethodName(Arguments) [throw ExceptionType]

✔ * : matches once (ReturnType, Package, class, MethodName, Argument)

✔ .. : matches zero or more (Argument, Package)

Example Expression: *Implementation vs Interfaces*

```
execution(void example.MessageServiceImpl.*(..))
```

Any method
--> in example package and MessageServiceImpl class with zero or more argument
--> returns void

- -> Including any sub-class with same implementation

```
execution(void example.MessageService.send(*))
```

Any method
--> which name is send and taking one argument, in any object implementing MessageService
--> returns void

More flexible choice - works if implementation changes

✔ Pattern: [Modifiers] ReturnType [Class Type] MethodName(Arguments) [throw ExceptionType]

✔ * : matches once (ReturnType, Package, class, MethodName, Argument)

✔ .. : matches zero or more (Argument, Package)

Example Expression: *Using Annotations*

```
execution(@javax.annotation.security.RolesAllowed void send*(..))
```

--> Ideal technique for your own annotations on your own classes

Any method
--> which name starts with send, takes zero or more argument
--> returns void
--> with annotated with @RolesAllowed annotation

```
public interface Mailer {
    @RolesAllowed("USER")
    public void sendMessage(String text);
}
```

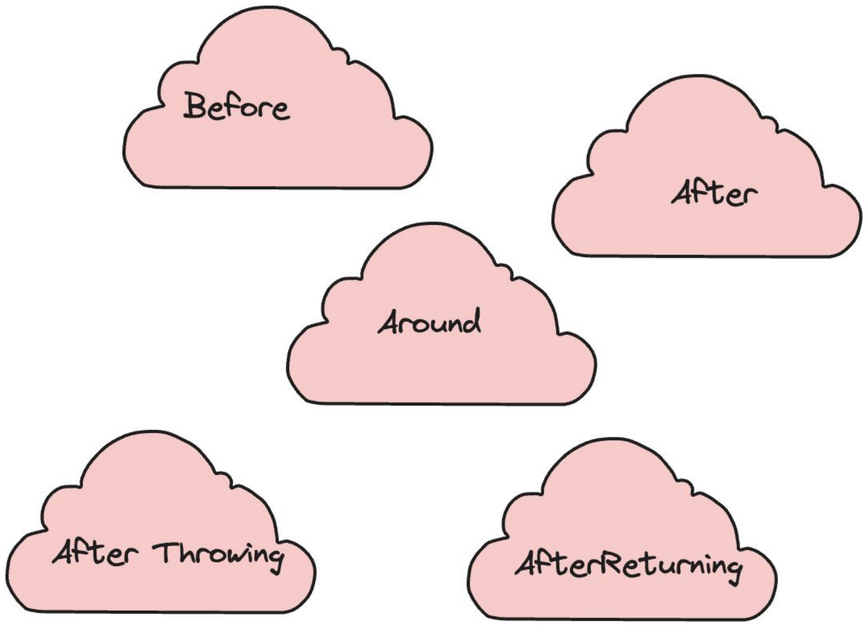
- ✔ Pattern: [Modifiers] ReturnType [Class Type] MethodName(Arguments) [throw ExceptionType]
- ✔ * : matches once (ReturnType, Package, class, MethodName, Argument)
- ✔ .. : matches zero or more (Argument, Package)

5. Aspect Oriented Programming

1. What Is Aspect Oriented Programming?
2. What Problems Does AOP Solve?
3. AOP Concepts
4. Pointcut Expressions
5. Advice Types
6. Lab Section: *Aspect Oriented Programming*

Advice Types

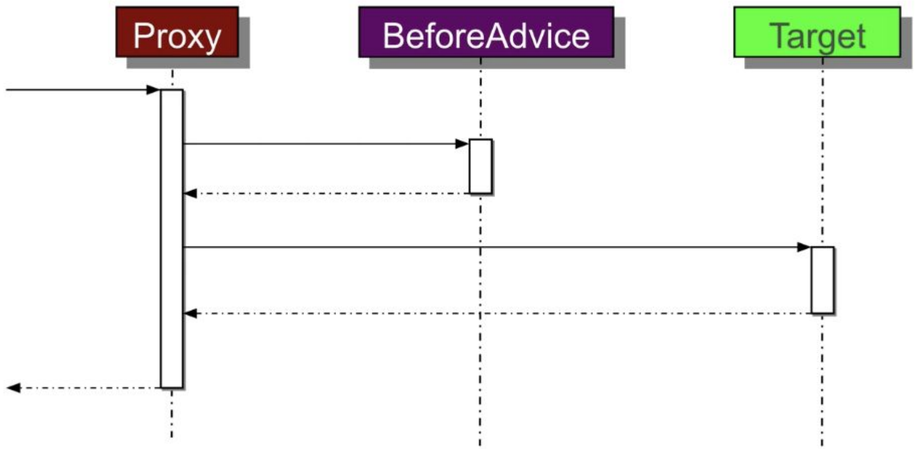
🔔 *Advice*: Advice is a specific code executed at a certain join point.



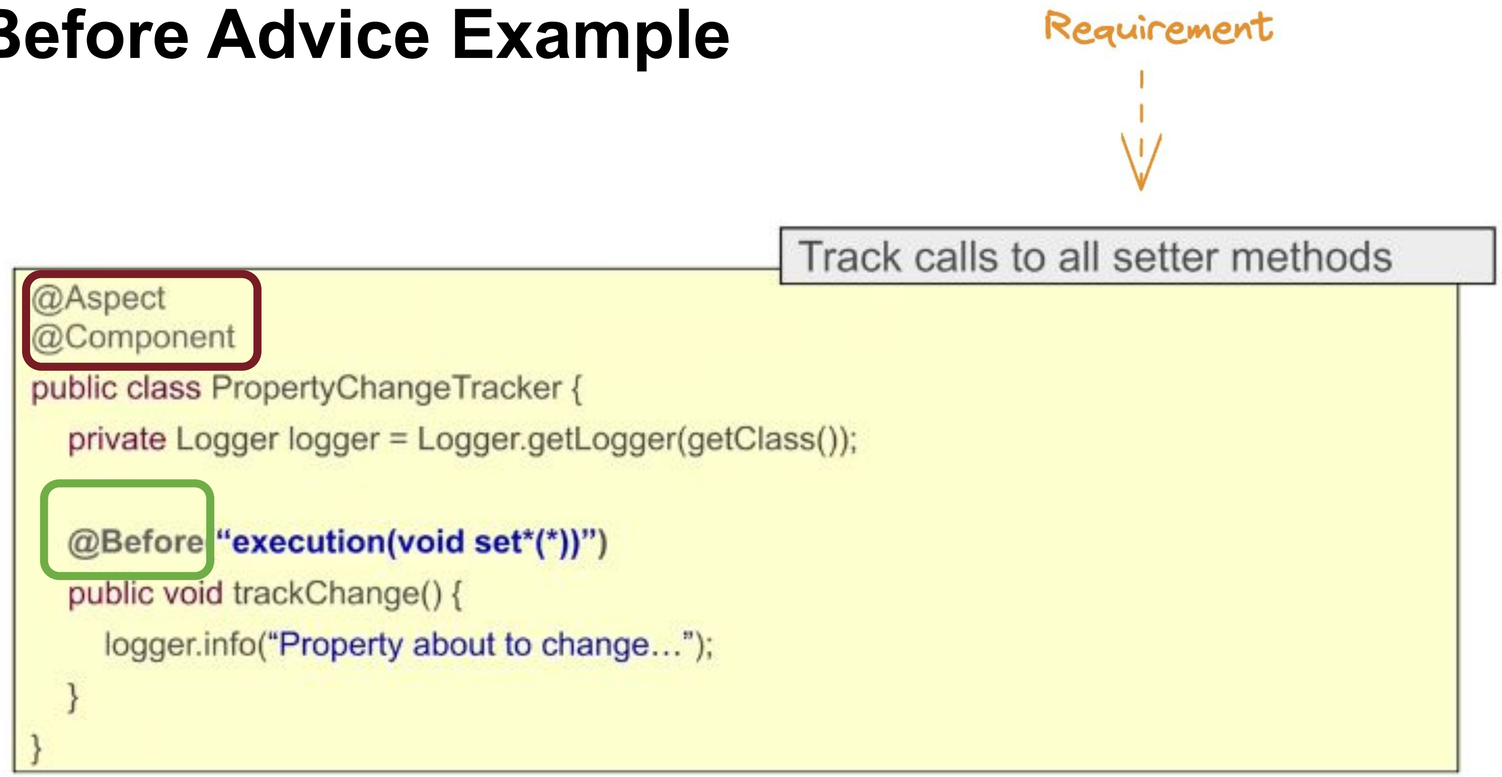
Advice Types: *@Before*

@Before → org.aspectj.lang.annotation package.

- ❏ It runs *before* a join point.
- ❏ Proxy is responsible for advice and target execution.
- ❏ It can’t prevent execution flow proceeding to the join point unless it throws an exception.



Before Advice Example

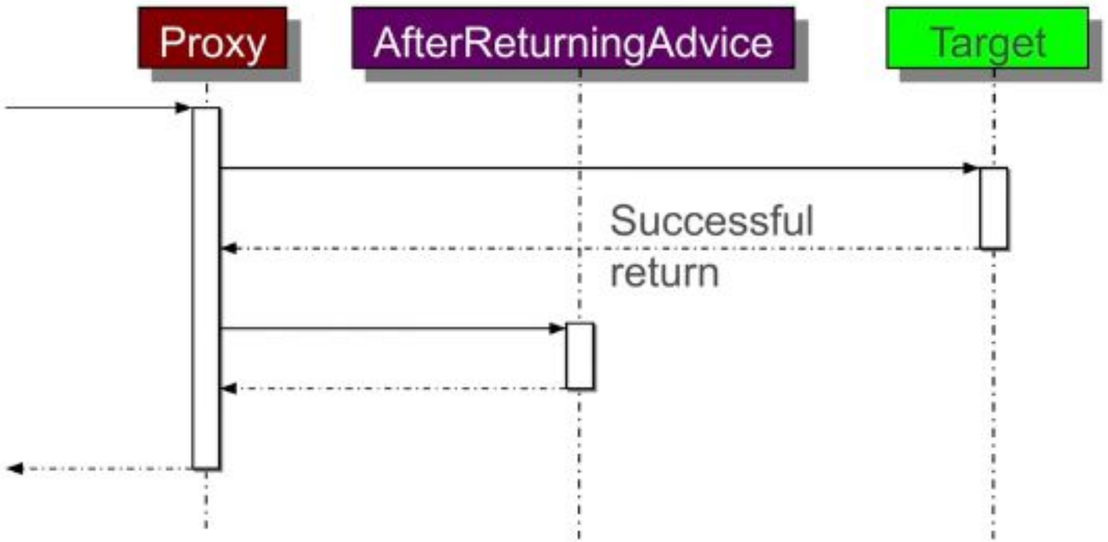


- ❏ Find all set methods with only one argument and void type.
- ❏ Before set method execution, add info log and then call set method.
- ❏ If there is an exception on this trackChange() method, setter methods are not called.

Advice Types: *@AfterReturning*

@AfterReturning → org.aspectj.lang.annotation package.

- ❏ It runs *after* a join point and completes normally
- ❏ Proxy is responsible for advice and target execution.
- ❏ If a method throws an exception this advice does not run. It waits <u>successful</u> return.



trendyol learning

AfterReturning Advice Example

Requirement

Audit all operations in the *service* package that return a *Reward* object

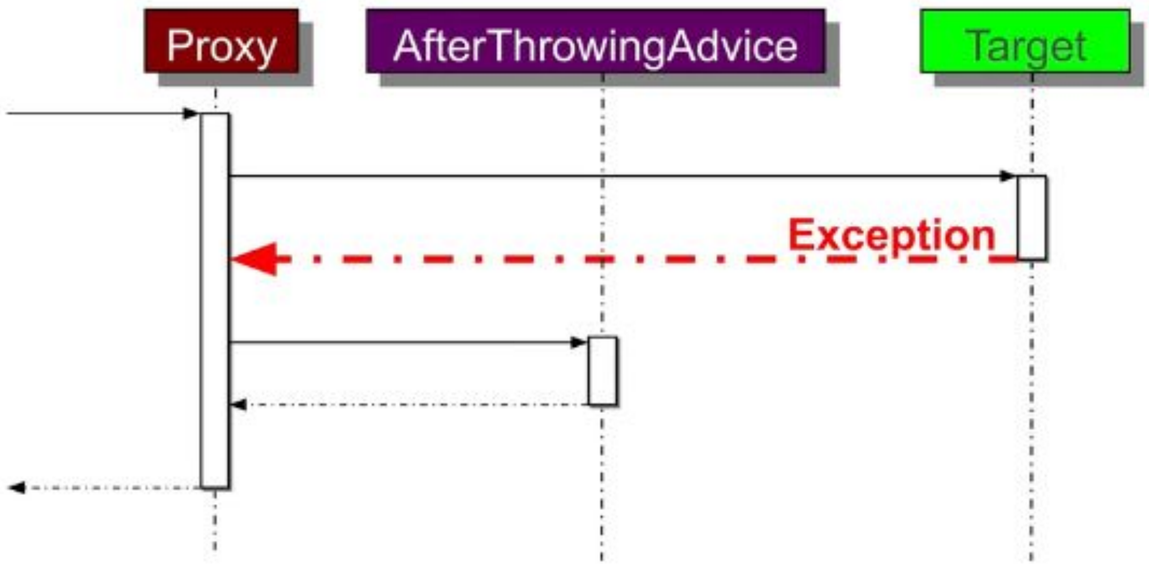
```
@AfterReturning(value="execution(* service..*.*(..))",
                returning="reward")
public void audit(JoinPoint jp, Reward reward) {
   auditService.logEvent(jp.getSignature() +
      " returns the following reward object :"  + reward.toString() );
}
```

- Find all methods with zero or more arguments in any class in any package in service and also returning any type.
- Execute method
- If this method returns Reward object, advice will invoke and logEvent via auditService.
- You can see value and returning parts in the example.

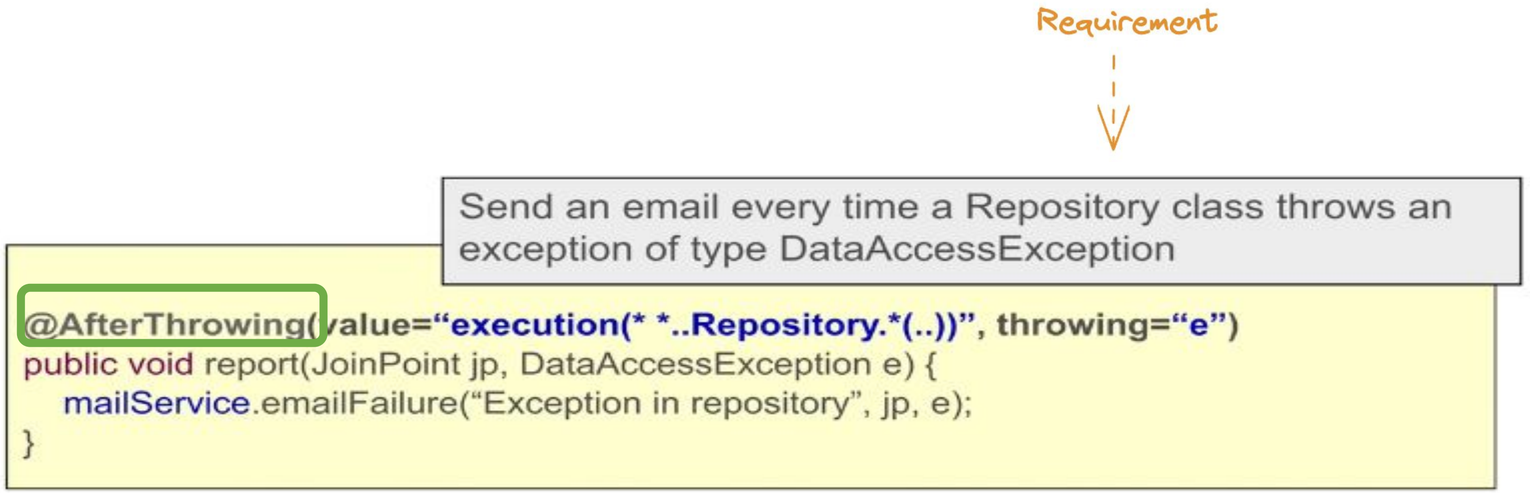
Advice Types: *@AfterThrowing*

@AfterThrowing → org.aspectj.lang.annotation package.

- ❏ It runs if a method throws an exception.
- ❏ Proxy is responsible for advice and target execution.
- ❏ If a method does not throw an exception this advice does not run. It waits exception.
- ❏ 🔔 Very useful for exception handling



AfterThrowing Advice Example

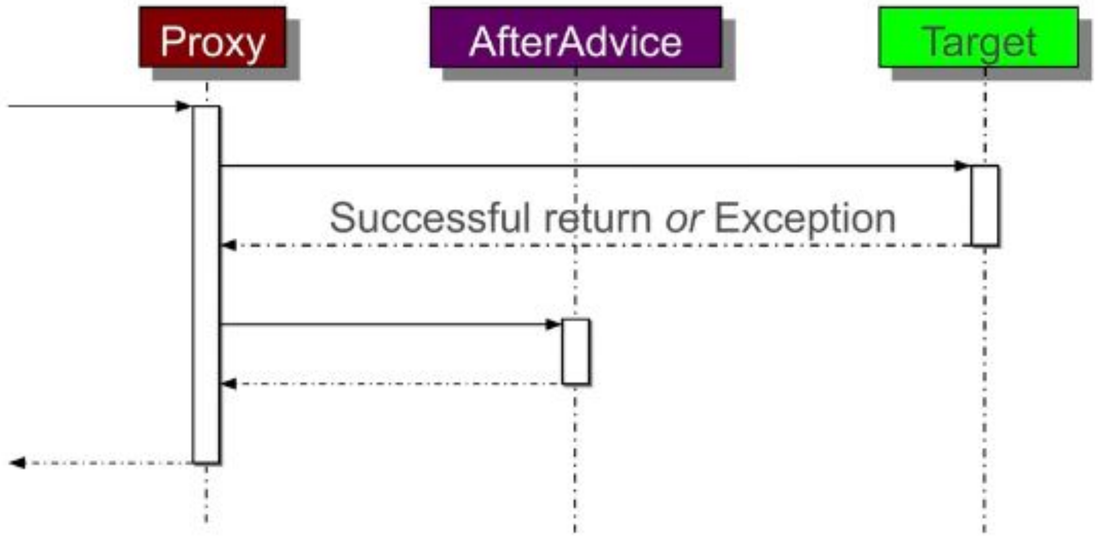


- ❏ Find all methods with zero or more arguments in Repository class in any package and also returning any type.
- ❏ Execute method.
- ❏ If this method throw DataAccessException, advice will invoke and mail will send.
- ❏ You can see value and throwing parts in the example.
- ❏ Any child exception of DataAccessException will match the expression.

Advice Types: *@After*

@After → org.aspectj.lang.annotation package.

- ❏ It runs regardless of how a join point exits whether by normal or exceptional return.
- ❏ Proxy is responsible for advice and target execution.



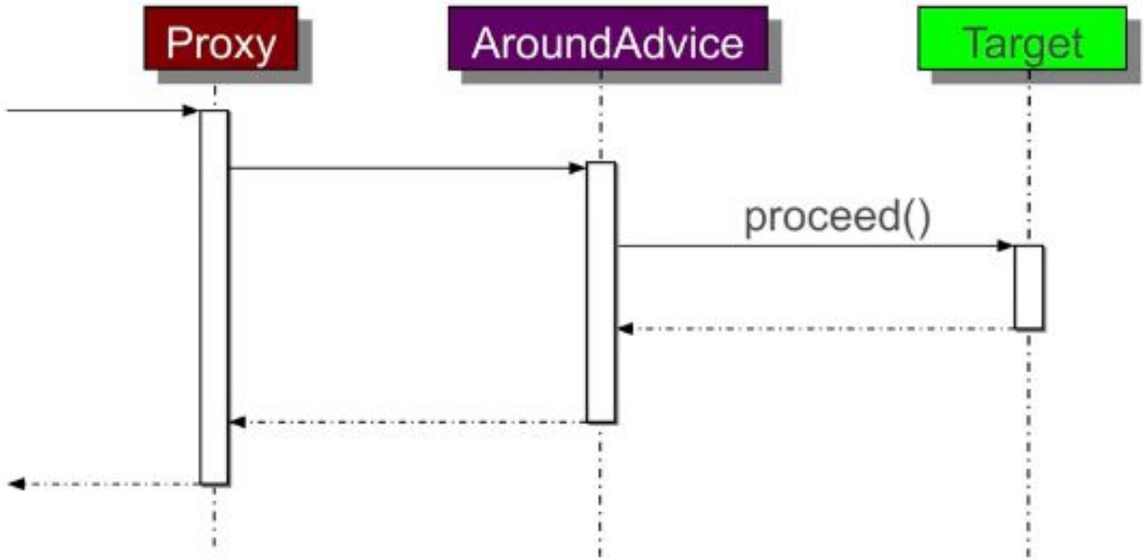
After Advice Example

- ❏ Find any methods with zero or more arguments and started with update and also returning void.
- ❏ Execute method. It does not matter return value exception or success
- ❏ And then add info log.

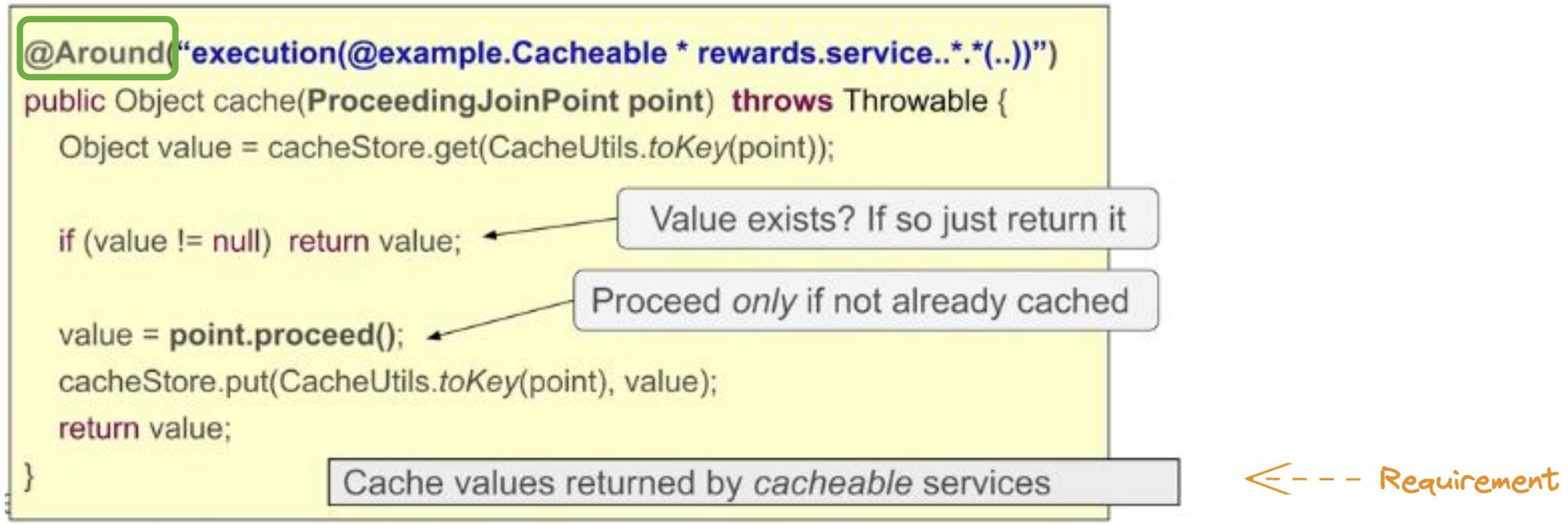
Advice Types: *@Around*

@Around → org.aspectj.lang.annotation package.

- ❏ It runs before and after a join point.
- ❏ It is like a combination of before and after returning aspects.
- ❏ Proxy is responsible for only advice execution. Target will execute by advice.
- ❏ Most *powerful* and also most *dangerous* advice type.



Around Advice Example



- ❏ Find any methods with zero or more arguments in rewards.service package and also returning any type and annotated with Cacheable.
- ❏ put cache new value
- ❏ Execute method.
- ❏ If cache is found, return value otherwise put cache new value

Limitations of Spring AOP

- Can only advise *non-private* methods.
- Can only apply aspects to Spring Beans.
- Assume method a() calls method b() on the same class/interface
  - Advice will *never* be executed for method b()
  - Advice will *never* be executed for inner-classes

Summary 

AOP modularizes *cross-cutting* concerns

An aspect is a module containing cross-cutting behaviour.

- ❏ Annotated with **@Aspect**
- ❏ Behavior is implemented as an “advice” method.
- ❏ Pointcuts select joinpoints(methods)
- ❏ Five advice types
  - ❏ Before, AfterThrowing, AfterReturning, After and Around

AOP is very useful for security, logging, error handling, caching

5. Aspect Oriented Programming

1. What Is Aspect Oriented Programming?
2. What Problems Does AOP Solve?
3. AOP Concepts
4. Pointcut Expressions
5. Advice Types
6. Homework: *Aspect Oriented Programming*

Homework 

What you will learn:

1. How to write an aspect and weave it into your application
2. Spring AOP using annotations
3. Writing pointcut expressions

Requirements:

REQUIREMENT 1: Create a simple logging aspect for repository find methods.

REQUIREMENT 2: Implement an @Around Advice which logs the time spent in each of your repository update methods.

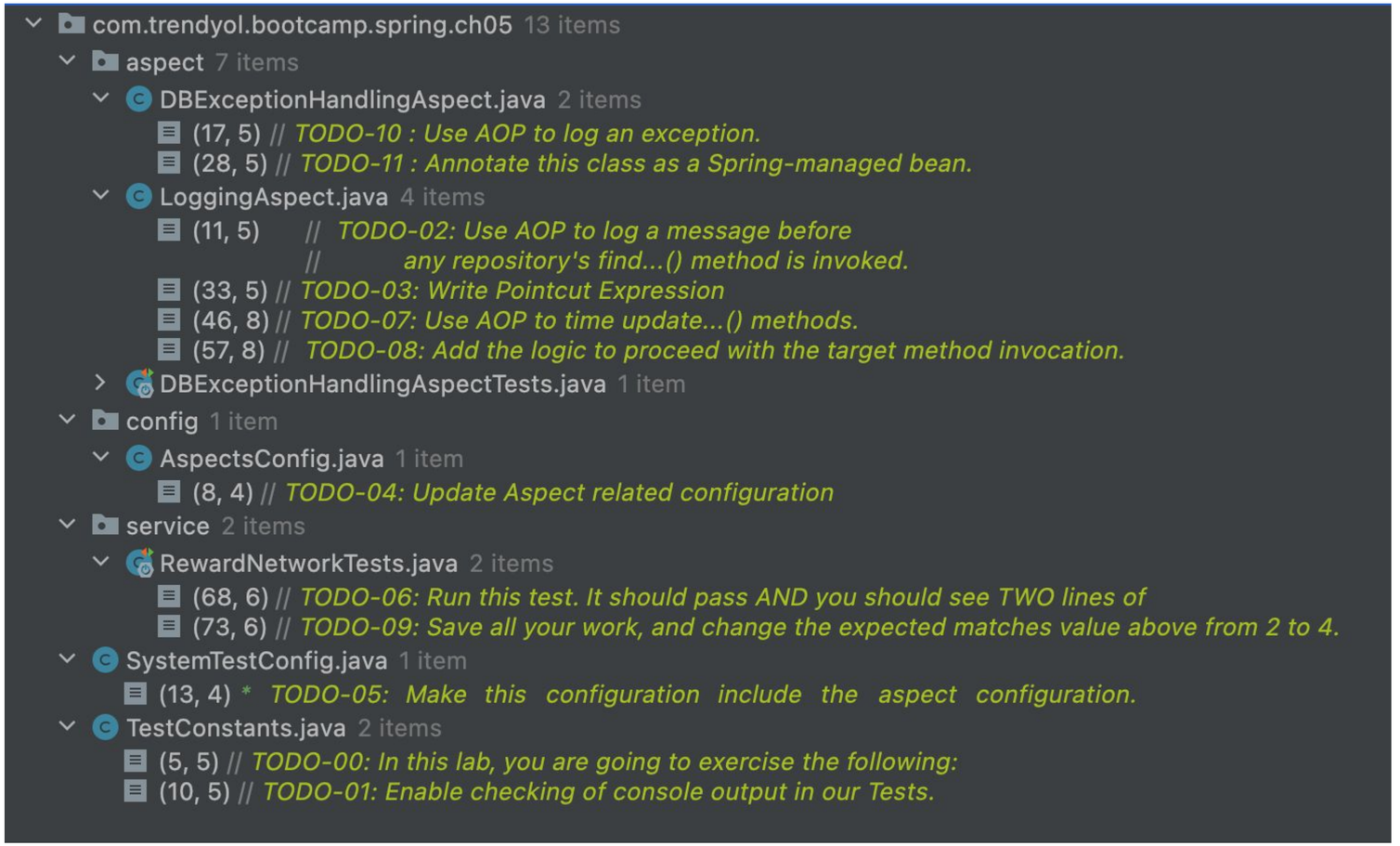
REQUIREMENT 3: Implement an @AfterThrowing Advice which catch exceptions on database operations.

Todos:

1. Fork project from github : https://github.com/gulumseraslann/spring-training
2. Switch branch to feature/aspect-oriented-programming
3. Create a new branch from this branch, your new branch name should be feature/aspect-oriented-programming-homework
4. There are 13 TODOs in the project files. Look at these TODOs
5. Please try to do each TODO
6. Please make sure tests are success.
7. Please add the changes and push the solution code in your github repository.

Homework

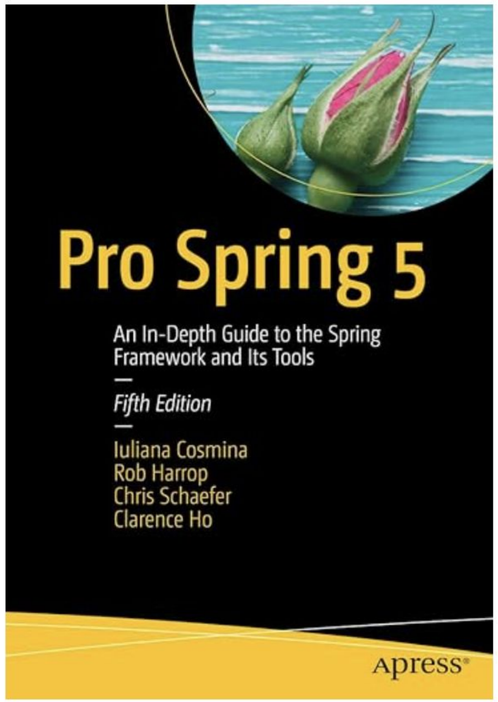
TO-DO:

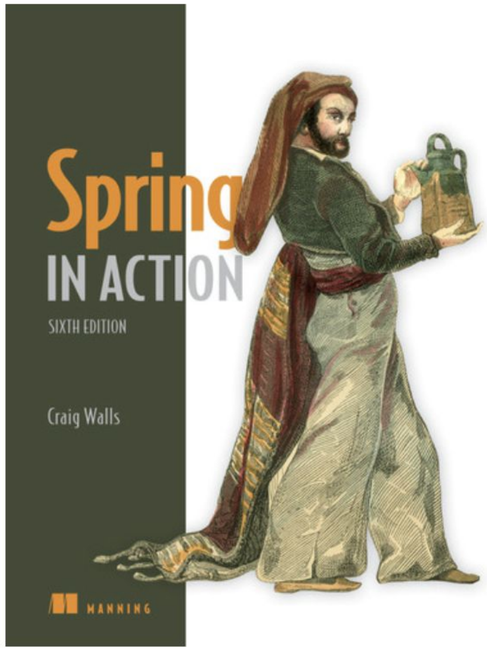


Final


dependency injection
inversion of control
service
advice
bean
aspect
configuration
annotation
pointcut
component
spring boot
dependency

Book Recommendation





Q & A

THANK YOU

trendyol
learning