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

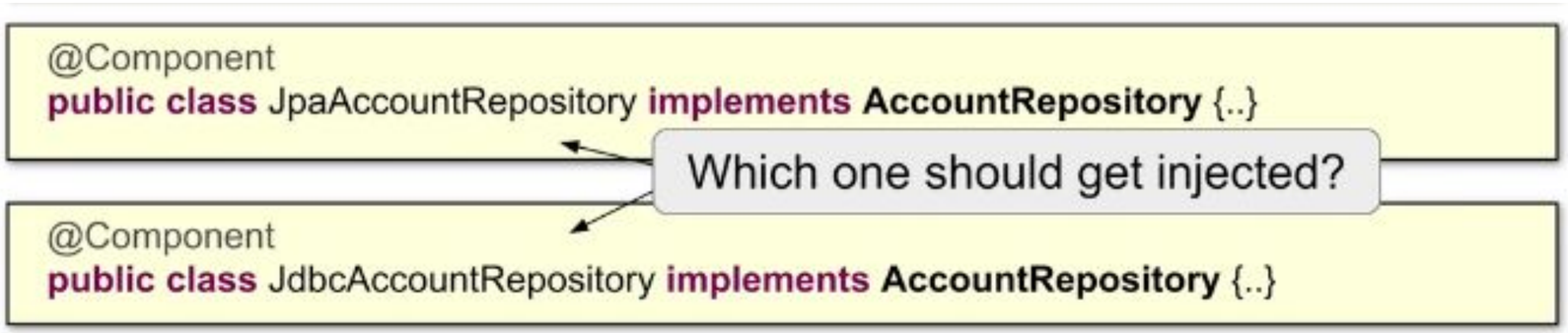
trendyol learning

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

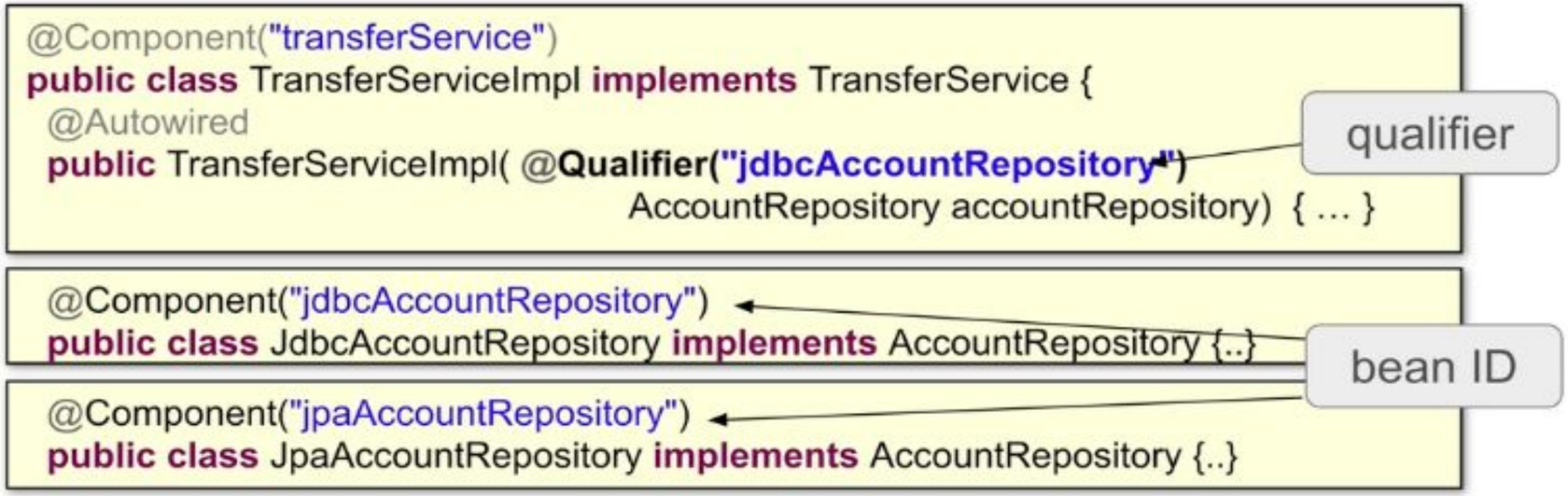



```
@Component
public class TransferServiceImpl implements TransferService {
  @Autowired // optional
  public TransferServiceImpl(AccountRepository accountRepository)  { … }
}
```

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

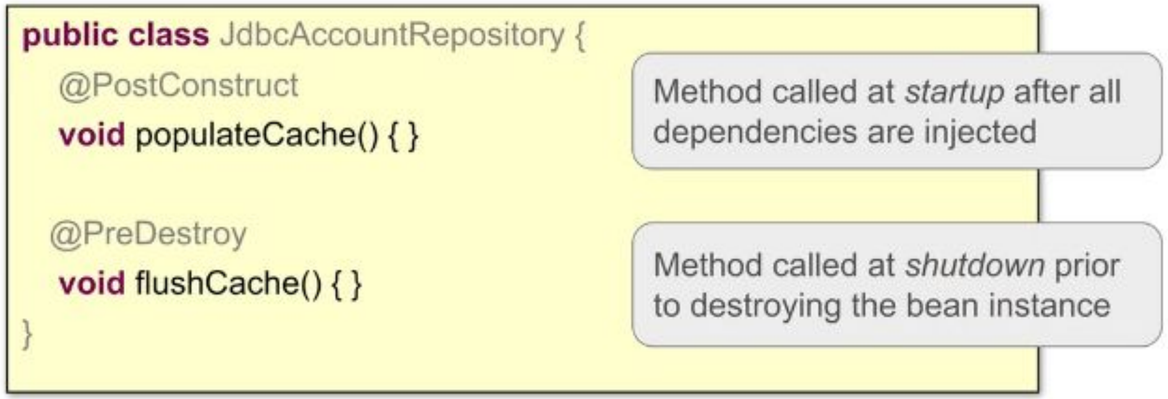
1. Annotation Based Configuration Concept
2. Usage of Autowired
3. Component Scan
4. Lifecycle Annotations
5. Stereotype Annotations
6. Lab Section: *Annotation Based Configuration*

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.

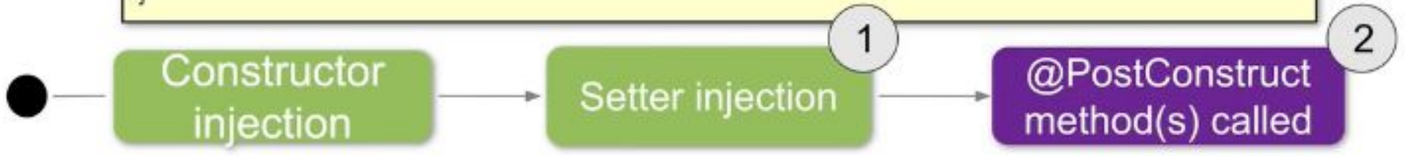


@PostConstruct

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

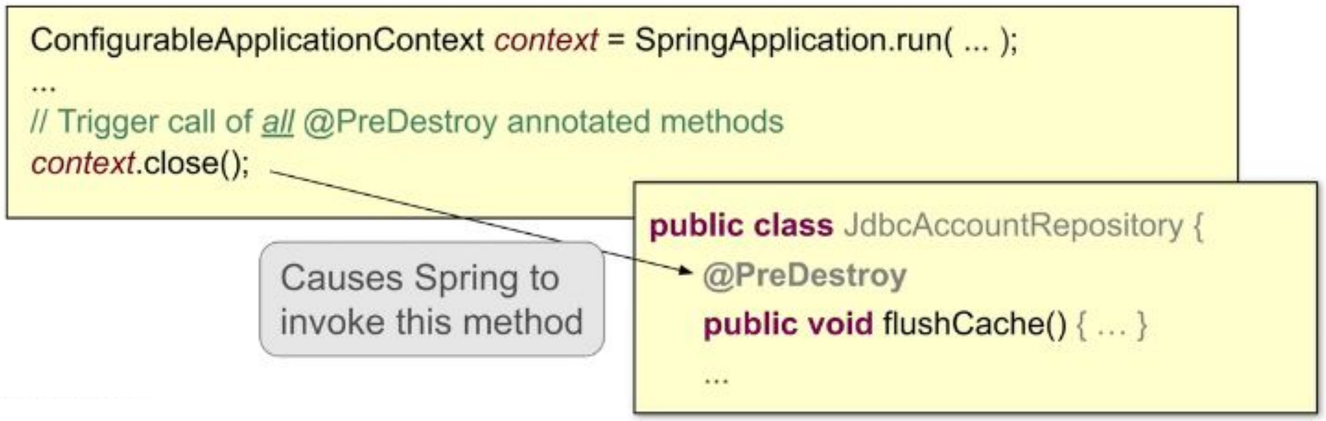
```
public class JdbcAccountRepository {
    private DataSource dataSource;
    @Autowired                                          1
    public void setDataSource(DataSource dataSource)
      {  this.dataSource = dataSource;   }

    @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*) !

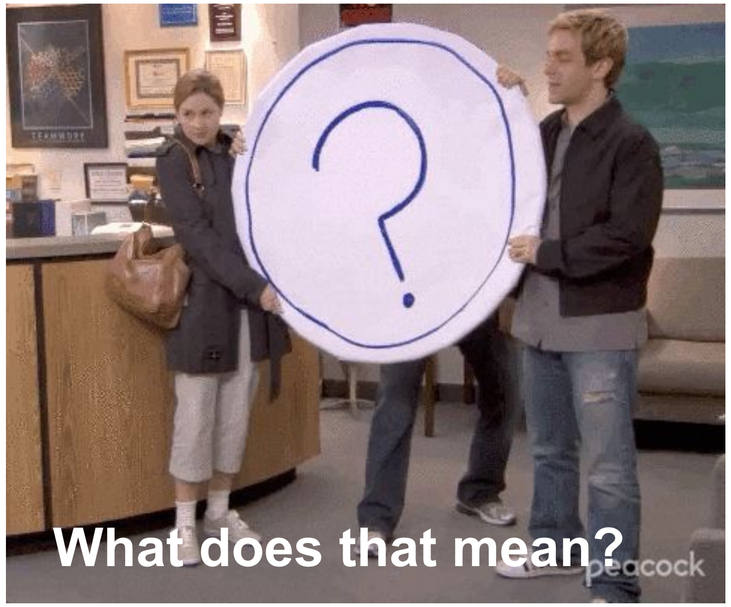


4. Annotation Based Configuration

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

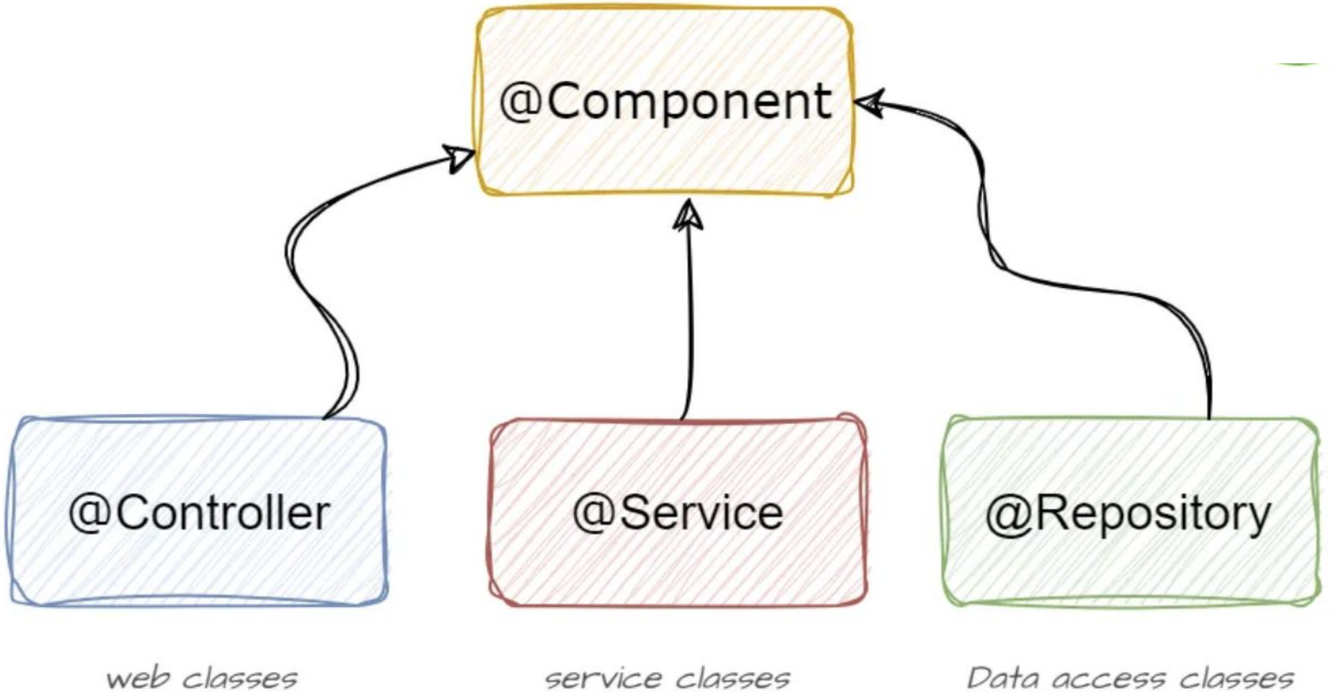
Stereotype Annotations

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**

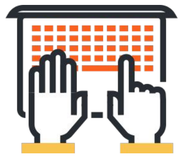
With using Spring's stereotypes annotation, you can create custom annotations.

- ❏ @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/fork 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