# Core Spring 4.2

Inhalt

[Core Spring 4.2 1](#_Toc481176430)

[Wichtige Spring dependencies: 1](#_Toc481176431)

[Abschnitt 2: Dependency, Container and IOC (Inversion of Control) 2](#_Toc481176432)

[Dependency injection using XML Configuration 2](#_Toc481176433)

[Dependency Injection using Java configuration (@Configuration annotation) 4](#_Toc481176434)

[Dependency Injection (@Component, @Autowired, @Qualifier) and component scan 5](#_Toc481176435)

[The concept of Spring´s Container (ApplicationContext) and its lifecycle 6](#_Toc481176436)

[Java Configuration advantages and @Configuration usage 7](#_Toc481176437)

[Understand the usage of Spring framework beans, inner beans and @Bean Annotation 7](#_Toc481176438)

[Final methods with @Bean annotation and abstract beans 9](#_Toc481176439)

[How to configure scopes for Spring Beans 9](#_Toc481176440)

[Understand the usage of init and destroy methods 10](#_Toc481176441)

[How to use and configure BeanFactoryPostProcessor and BeanPostProcessor 10](#_Toc481176442)

[How to use @Profile annotation and profiles in XML configuration 12](#_Toc481176443)

[Understand the usage of SpEL (Spring Expression Language) 14](#_Toc481176444)

[How to configure Property Placeholder Configurer 15](#_Toc481176445)

[How to use ApplicationContext in JUnit Tests 16](#_Toc481176446)

[Spring Containers lifecycle in details and creating ApplicationContext 17](#_Toc481176447)

[Abschnitt 3: Aspect Oriented Programming (AOP) 18](#_Toc481176448)

[Understanding the concept of AOP programming (Pointcut, Jointpoint and Advice) 18](#_Toc481176449)

[Benefits of AOP and what are cross cutting concerns 18](#_Toc481176450)

[How to configure @Aspect annotation in XML and JavaConfig 19](#_Toc481176451)

[Learn the usage of Advices (After, Before, AfterReturning, AfterThrowing, Around) 20](#_Toc481176452)

[Learn the usage of @After and @Before advices with code examples 21](#_Toc481176453)

[Learn the usage of @AfterReturning and @AfterThrowing with code examples 21](#_Toc481176454)

# Wichtige Spring dependencies:

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>4.3.7.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>4.3.7.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

<version>4.3.7.RELEASE</version>

</dependency>

## Abschnitt 2: Dependency, Container and IOC (Inversion of Control)

### Dependency injection using XML Configuration

Benötigte Maven dependencies:

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>4.3.7.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>4.3.7.RELEASE</version>

</dependency>

</dependencies>

In der Main-Klasse laden wir die Konfig-XML Datei und daraus eine Bean:

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("application-context.xml");

MessageService messageService = context.getBean(MessageService.class);

messageService.sendMessage();

}

}

Das hier ist eine Bean, welche nun per Konstruktor- & Setter-Injection die zwei Felder gefüllt bekommt:

public class MessageService {

private Log logger;

private String operatorName;

public MessageService(Log logger) {

this.logger = logger;

}

public void sendMessage() {

System.out.println("Operator " + getOperatorName() + " sending message...");

logger.log("sent successful");

}

//getter setter

}

Und hier die Config-Datei:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:aop="http://www.springframework.org/schema/aop"

xmlns:tx="http://www.springframework.org/schema/tx"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.0.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">

<bean id="MessageService" class="de.hays.springtestapplication.MessageService">

<constructor-arg ref="logger" />

<property name="operatorName" value="andrew" />

</bean>

<bean id="logger" class="de.hays.springtestapplication.LogImpl" />

</beans>

### Dependency Injection using Java configuration (@Configuration annotation)

Man kann auch eine Java-Klasse zur Konfiguration verwenden und das ganze programmatisch abbilden- ähnlich wie im vorherigen Beispiel mit der XML-Datei.

**Dazu erstellt man eine Klasse, welche die Annotation @Configuration bekommt:**

@Configuration

public class JavaConfig {

@Bean

public MessageService messageService() {

MessageService messageService = new MessageService(logger());

messageService.setOperatorName("Java-configured-Op");

return messageService;

}

@Bean

public LogImpl logger() {

LogImpl logger = new LogImpl();

return logger;

}

}

* Hier sieht man, dass die Bean MessageService per Constructor-Injection sich eine LogImpl zieht, ähnlich wie oben per XML-Config

In der MainApp muss man sich dann den ApplicationContext aus der passenden Java-Klasse generieren lassen:

public class MainApp {

public static void main(String[] args) {

ApplicationContext applicationContext = new AnnotationConfigApplicationContext(JavaConfig.class);

MessageService messageService = applicationContext.getBean(MessageService.class);

messageService.sendMessage();

}

}

### Dependency Injection (@Component, @Autowired, @Qualifier) and component scan

**Component scanning:** Definiert welche packages nach Spring beans durchsucht werden sollen. Folgende Stereotypen werden von diesem scan berücksichtigt:

* **@Component** Generischer Stereotyp für eine beliebige Spring-Komponente
* **@Repository** Stereotyp für persistence layer
* **@Service** Stereotyp für service layer
* **@Controller** Stereotyp für presentation-layer, spring-mvc

Code-Beispiel:

@Configuration

**@ComponentScan**(basePackages = "de.hays.springtestapplication.bean")

public class JavaConfig { … }

…die anderen Komponenten (Beans) müssen nun noch mit der passenden Annotation getagged werden:

@Component

@Qualifier("superlog")

public class LogImpl implements Log {

@Override

public void log(String message) {

System.out.println("Logger is logging: " + message);

}

}

*Hierbei wird ein Qualifier angegeben, falls es mehrere Interface-Implementierungen gibt und ansonsten Spring folgende Exception wirft, wenn er keinen eindeutigen Injection-Kandidaten findet:*

**org.springframework.beans.factory.UnsatisfiedDependencyException**

**@Autowired** wird verwendet, um eine Bean in einem Feld zu injecten lassen. Dabei kann man auch einen Qualifier angeben:

@Component

public class MessageService {

@Autowired

@Qualifier("crazylog")

private Log logger;

@Autowired

private OperatorBean operatorBean;

public void sendMessage() {

System.out.println("Operator " + operatorBean.toString() + " sending message...");

logger.log("sent successful");

}

### The concept of Spring´s Container (ApplicationContext) and its lifecycle

Der ApplicationContext ist ein “Container”–Objekt, welches die Beans enthält.

Der Spring Container hat die Aufgabe des Lifecycle Managers, für jegliche Klassen der Applikation.

Die drei Lifecycle-Phasen sind:

1. Initialization: Beans, Application Services, Ressourcen werden erstellt und konfig.
2. Use Phase: Ojekte werden von den Benutzern verwendet, Requests bearbeitet und Anwendungs-flows vom Container gemanaged
3. Destruction Phase: Verantwortlich für die Zerstörung der Beans, freigeben von System Ressourcen und Garbage-Collection

Um sich eine Bean aus dem ApplicationContext zu laden, macht man es wie in den vorherigen Beispielen: ***(Sobald die Klasse mit angegeben wird, ist es typsicher und muss nicht gecasted werden)***

[Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html?is-external=true) getBean([String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html?is-external=true" \o "class or interface in java.lang) name)

[Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html?is-external=true) getBean([String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html?is-external=true" \o "class or interface in java.lang) name, [Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html?is-external=true)... args)

<T> T getBean([String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html?is-external=true" \o "class or interface in java.lang) name, [Class](https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html?is-external=true)<T> requiredType)

<T> T getBean([Class](https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html?is-external=true" \o "class or interface in java.lang)<T> requiredType)

<T> T getBean([Class](https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html?is-external=true" \o "class or interface in java.lang)<T> requiredType, [Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html?is-external=true)... args)

**Wie sollte ein ApplicationContext geschlossen werden?**

*((ConfigurableApplicationContext)appCtx).close();*

### Java Configuration advantages and @Configuration usage

Advantages of Java-Config:

* Typsicher
* Refactoring friendly
* Reduced and eliminated XML-Config

**Java Configuration Klassen dürfen nicht final sein!** Da im Hintergrund das Spring Framework dynamische Proxies erstellt und davon erben muss.

Wenn man mit der Java-Config Variante arbeitet, so nutzt man den **AnnotationConfigApplicationContext** (Wie in den Beispielen vorher).

@Configuration Klassen benötigen einen **(impliziten) default Konstruktor.**

### Understand the usage of Spring framework beans, inner beans and @Bean Annotation

**What does the @Bean annotation do?** –Registriert innerhalb einer Java-Config Klasse über einer Methode die Bean mit dem entsprechendem Namen, z.B.

@Bean

public MessageService **messageService**() {…}

**What is the default bean id if you only use @Bean?** –In der Java-Config ist die ID gleich dem Methodenname, bei der XML-Config gibt es dazu den id=“x“ flag.

**What is an inner bean? Why does it not have bean id? Can it be reused?** –Eine innere Bean gehört nur zu der umliegenden Bean, deshalb kann sie nirgends anders wiederverwendet werden. **Eine ID hat sie auch nicht**.

Beispiel: Wir haben folgendes Objekt:

MessageService ist eine Bean, welche mehrere Felder hat. Unter anderem enthält sie aber auch noch eine andere Bean – MailServiceObject.

In einer XML-Konfiguration würde das dann wie folgt aussehen:

<bean id="MessageService" class="de.hays.springtestapplication.MessageService">

<property name="property1" value="andrew" />

<property name="property1" value="andrew" />

<property name=”MailServiceObject”>

<bean class=”de.hays.MailServiceObject”>

<property name=”server” value=”x1”/>

</bean>

</property>

</bean>

**What is bean definition inheritance?**

Beispiel:

Angenommen man hat zwei Beans- mit fast ähnlichen Feldern.  
So kann man in der Spring-Config einstellen, dass die eine Bean von der anderen die Properties erbt- selbt wenn die beiden Klassen gar keine Vererbung nutzen:

<bean id="testUser" class="de.hays.springtestapplication.bean.User">

<property name="username" value="maxmustermann"/>

<property name="password" value="secret"/>

<property name="active" value="true"/>

</bean>

<bean id="superUser" class="de.hays.springtestapplication.bean.SuperUser" parent="testUser">

<property name="systemPassword" value="verysecret"/>

<property name="contact" value="-hidden-"/>

</bean>

### Final methods with @Bean annotation and abstract beans

What happens if the parent bean is not abstract?

Abstrakte Beans werden dazu verwendet, gleiche Properties zu definieren und müssen nicht auf eine existierende Java Klasse mappen.

Valides Beispiel: (siehe objekt-auflistung im letzten unterpunkt):

<bean id="testUser" abstract="true">

<property name="username" value="maxmustermann"/>

<property name="password" value="secret"/>

<property name="active" value="true"/>

</bean>

<bean id="superUser" class="de.hays.springtestapplication.bean.SuperUser" parent="testUser">

<property name="systemPassword" value="verysecret"/>

<property name="contact" value="-hidden-"/>

</bean>

Hierbei werden die properties der abstrakten Bean für die SuperUser bean verwendet.

**Why are you not allowed to annotate final methods with @Bean?**

Falls man Methoden mit final markiert, kann CGLIB (Welches hintendran im SpringFramework genutzt wird) nicht mehr von dieser Klasse erben- weshalb es eine Exception gibt.

### How to configure scopes for Spring Beans

Singleton: Eine einzige Instanz des Objekts wird im Framework gehalten

Prototype: Jedesmal wenn das Objekt referenziert wird, wird eine neue Instanz erstellt

Session: Für jede http Session wird eine Instanz erstellt- wie z.B. ein Warenkorb-Objekt

Request: Für jeden Request wird eine neue Instanz erstellt.

Global Session: Das gleiche wie Session, wird genutzt in web applications, Portlets

Application: Wird in web applications genutzt und scoped eine single bean definition im ServletContext

**Der default scope für Objekte, welche in der Spring-Config definiert werden, ist Singleton (One shared instance). Das kann geändert werden, siehe Beispiel:**

<bean id=“DBLogService“ class=“spring.bean.DBLogService“ **scope=“singleton“** />

…oder per Annotation-Config: **@Scope(“session”)**

### Understand the usage of init and destroy methods

**What is the role of the @PostConstruct and @PreDestroy annotations? When will they get called? How do you enable JSR-250 annotations like @PostConstruct?**

@PostConstruct wird nach dem initialisieren der Bean aufgerufen. @PreDestroy bevor das Objekt vom ApplicationContext entfernt wird:

@PostConstruct

public void init() {

System.out.println("@PostConstruct is called!");

}

@PreDestroy

public void destroy() {

System.out.println("@PreDestroy is called!");

}

-Um das mit der XML-Konfig zu machen, fügt man folgenden Parameter hinzu:

<bean id="superUser" class="de.hays.springtestapplication.bean.SuperUser" **init-method="startInit" destroy-method="endInit"**>

### How to use and configure BeanFactoryPostProcessor and BeanPostProcessor

**What is a BeanFactoryPostProcessor and whats used for?**

-Ändert eine Bean Definition bevor sie im ApplicationContext erstellt wird. PropertyPlaceholderConfigurer ist ein Beispiel für einen BeanFactoryPostProcessor

**What is a BeanPostProcessor and how is the difference to a BeanFactoryPostProcessor? What do they do? When are they called?**

BeanFactoryProcessor: Können die Definition einer Bean verändern, noch bevor diese im ApplicationContext sind

BeanPostProcessor (BPP): Können Beans konfigurieren, nachdem diese im ApplicationContext erstellt worden sind

**When do you have to implement the FactoryBean interface?**

Das Interface (**FactoryBean <T>**) enthält folgende Methoden:

T getObject()  
Class getObjectType()  
boolean isSingleton()

*Beispiel-Implementation:*

ComplexBean ist die Bean an sich:

public class ComplexBean {

private String seconds;

public String getSeconds() {

return seconds;

}

public void setSeconds(String seconds) {

this.seconds = seconds;

}

@Override

public String toString() {

return "ComplexBean{" + "seconds=" + seconds + '}';

}

public void initComplexBean() {

for (int i = 0; i < 5; i++) {

try {

Thread.sleep(i \* 1000);

} catch (Exception e) {

} } } }

**ComplexBeanFactoryBeanImplementation implementiert nun das FactoryBean-Interface und ist für die Erstellung der Bean verantwortlich:**

public class ComplexBeanFactoryBeanImplementation implements FactoryBean<ComplexBean> {

@Override

public ComplexBean getObject() throws Exception {

ComplexBean bean = new ComplexBean();

long start = System.currentTimeMillis();

bean.initComplexBean();

long end = System.currentTimeMillis();

bean.setSeconds((end - start) / 1000 + " seconds");

return bean;

}

@Override

public Class<?> getObjectType() {

return ComplexBean.class;

}

@Override

public boolean isSingleton() {

return false;

}

}

Zuletzt muss die Bean nurnoch in der application-context.xml definiert werden und kann anschließend verwendet werden:

*<bean id="complexBean" class="de.hays.springtestapplication.bean.ComplexBeanFactoryBeanImplementation"/>*

### How to use @Profile annotation and profiles in XML configuration

Man kann z.B. ein Development- und Production-Profil „erstellen“ und sich je nach Zweck verschiedene Beans generieren lassen:

#### Als Java-Config

@Configuration

public class JavaConfig {

@Bean

@Profile("production")

public User getUserProduction() {

User user = new User();

user.setUsername("prodUser");

return user;

}

@Bean

@Profile("development")

public User getUserDevelopment() {

User user = new User();

user.setUsername("devUser");

return user;

}

}

Und innerhalb des Unit-Tests kann man sich dann die Bean holen, welche „devUser“ als Username hat:

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration(classes=JavaConfig.class)

@ActiveProfiles(profiles="development")

public class TestJavaConfig {

@Autowired

private ApplicationContext context;

@Test

public void doTest() {

System.out.println(context.getBean(User.class));

}

}

#### Als XML-Config

In der application-config.xml erstellt man mehrere <beans> nodes, welche den parameter *profiles* bekommen- somit sind alle darin verschachtelten bean´s nur für das jeweilige profile gültig:

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.1.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.1.xsd">

<beans profile="**dev**">

<bean id="testUser" class="com.goblingift.bean.UserBean">

<property name="username" value="devUser"/>

</bean>

</beans>

<beans profile="**prod**">

<bean id="testUser" class="com.goblingift.bean.UserBean">

<property name="username" value="prodUser"/>

</bean>

</beans>

</beans>

Anschließend kann man sich im Unit-test mithilfe der Annotation *@ContextConfiguration(locations = {"classpath:application-context.xml"})* die beans aus der XML-Config laden und mit der gleichen Annotation wie oben *@ActiveProfiles(profiles="prod")* sagt man welches Profil aktiviert werden soll:

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration(locations = {"classpath:application-context.xml"})

@ActiveProfiles(profiles="prod")

public class TestJavaConfig {

@Autowired

private ApplicationContext context;

@Test

public void doTest() {

System.out.println(context.getBean(UserBean.class));

}

}

### Understand the usage of SpEL (Spring Expression Language)

Mit der Spring Expression language kann man einfach boolsche Operationen, Methodenaufrufe, Instanzfelder etc. auslesen.

In der XML-Config kann man somit z.B. auf SystemProperties zugreifen:

<bean id="testUser" class="com.goblingift.bean.UserBean">

<property name="username" value**="#{systemProperties['java.version']}"**/>

</bean>

Um auf Objekte zuzugreifen, nutzt man das Dollarzeichen, Bsp:

<property name="username" value=“**${jdbc.username}“/>**

### How to configure Property Placeholder Configurer

Mithilfe eines Property Placeholders, kann man die ${…} aufgerufenen Objekte auflösen.

#### Beispiel XML-Config:

<context:property-placeholder location="classpath:**passwords\_prod.properties**" />

<bean id="testUser" class="com.goblingift.bean.UserBean">

<property name="username" value="**${user\_guest}**"/>

</bean>

Hiermit wird die Properties-Datei „passwords\_prod“ eingelesen (welche unter src/main/resources liegt). Weitergehend kann dann mit der ${} Schreibweise die Werte ausgelesen werden.

#### Beispiel Java-Config:

**Variante A:** Man erstellt in der Java-Config Klasse eine Methode, welche als Bean eine PropertySourcesPlaceholderConfigurer Instanz zurück gibt:

@Configuration

@ComponentScan(basePackages = "com.goblingift.bean")

public class JavaConfig {

@Bean

public static PropertySourcesPlaceholderConfigurer propertySourcesPlaceholderConfigurer() throws IOException {

PropertySourcesPlaceholderConfigurer ps = new PropertySourcesPlaceholderConfigurer();

ps.setLocations(new PathMatchingResourcePatternResolver().getResources("**classpath\*:\*.properties**"));

return ps;

}

}

**Variante B:** Man gibt per Annotation an, wo die Property-Dateien gesucht werden sollen:

@Configuration

@ComponentScan(basePackages = "com.goblingift.bean")

@PropertySource("classpath:passwords\_prod.properties")

public class JavaConfig { }

*Um dann z.B. in einer Bean den Wert aus der Property-Datei zuzuweisen:*

@Component

public class UserBean {

@Value("${user\_guest}")

private String username;

…

### How to use ApplicationContext in JUnit Tests

#### Mit der @RunWith Annotation

Man setzt vor die JUnit Testklasse eine Annotation – danach kann man sich über @Autowired den ApplicationContext ziehen:

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration(classes = JavaConfig.class)

public class TestJavaConfig {

@Autowired

private ApplicationContext context;

…

#### Erben von AbstractJUnit4SpringContextTests

Man erstellt eine JUnit Testklasse, welche von der Abstrakten Spring Test Klasse erbt:

import org.springframework.test.context.junit4.AbstractJUnit4SpringContextTests;

public class TestExtendsJavaConfig extends AbstractJUnit4SpringContextTests { …

**Vorteil: Man hat direkt Zugriff auf ein applicationContext –Objekt aus der Superklasse!**

#### Implementieren von ApplicationContextAware

Man kann auch die JUnit Testklasse das Interface org.springframework.context.ApplicationContextAware implementieren lassen und die Methode public void setApplicationContext(ApplicationContext) implementieren. Man muss aber trotzdem über der Klasse die Annotation @RunWith(SpringJUnit4ClassRunner.class) anhängen- Vorteil ist nur, dass man nochmal eingreifen kann, bevor der ApplicationContext gezogen wird (Man verwendet hier kein @Autowired mehr):

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration(classes = JavaConfig.class)

public class TestExtendsJavaConfig implements ApplicationContextAware{

ApplicationContext applicationContext;

@Override

public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {

this.applicationContext = applicationContext;

}

}

### Spring Containers lifecycle in details and creating ApplicationContext

Life-cycle in details:

1. Beans are loaded (XML parsing of application-context.xml or Javaconfig)
2. Post Process Bean Definitions (Bean definitions load into BeanFactory with **ids**)
3. Each bean is instantiated (Dependency Injection)
4. Setters called (Dependency Injection)
5. Bean Post Processors (before init, can modify a bean)
6. Initializers called (Beans are created)
7. Bean Post Processors (after init)
8. **Beans ready to use**

#### How are you going to create a new instance of an ApplicationContext?

##### ClassPathXmlApplicationContext

ApplicationContext c = new ClassPathXmlApplicationContext(  
new String[] {„application-context.xml“, „dao.xml“});

##### FileSystemXmlApplicationContext

ApplicationContext c = new FileSystemXmlApplicationContext(„configuration/application-context.xml“);

##### XmlWebApplicationContext

XmlWebApplicationContext c = new XmlWebApplicationContext();  
c.setConfigLocation(„/WEB-INF/dispatcher-config.xml“);

##### AnnotationConfigApplicationContext (Java-Config)

ApplicationContext c = new AnnotationConfigApplicationContext(JavaConfig.class);

#### What is the lifecycle on an ApplicationContext?

1. Bean instantiation/wiring
2. BeanPostProcessor registration
3. BeanFactoryPostProcessor registration
4. MessageSource access (for i18n)
5. ApplicationEvent publication

## Abschnitt 3: Aspect Oriented Programming (AOP)

### Understanding the concept of AOP programming (Pointcut, Jointpoint and Advice)

#### What is the concept of AOP? Which problem does it solve?

AOP bietet eine andere Denkweise, Probleme zu lösen, wie z.B.:

* Transaction Management
* Logging
* Security
* Error Handling
* Monitoring and else

#### What is a pointcut, join point, advice, aspect and weaving?

Pointcut: Eine Expression, welche ein oder mehrere Joint Points selektiert und einschränkt

Joint point: Der exakte Punkt in der Ausführung Ausführung eines Programms- sprich die Methode die angesprochen wird. Das ist der Punkt an dem die normale Code-Ausführung und die Aspekt-Ausführung zusammenlaufen.

Advice (interceptors): Damit sagt man, dass z.B. bevor oder nach einer Methode eine andere aufgerufen wird

Aspect: Eine Klasse welche als Aspect angesehen wird, wenn sie mit @Aspect annotiert wurde

Weaving: Synonym für verlinken, sprich wie die Objekte miteinander verbunden werden

### Benefits of AOP and what are cross cutting concerns

#### Benefits of AOP

Reduziert den Code – verhindert z.B. den Aufruf einer securityCheck-Methode vor jeder Methode, indem man diese per AOP einfach immer autom. Aufruft.

Man kann mehr Funktionen zu Klassen hinzufügen, ohne Sourcecode ändern zu müssen.

#### How does Spring solve a cross cutting concern?

* Eine Klasse mit **@Aspect** annotieren (org.aspectj.lang.annotation.Aspect)
* Oder XML Config: **<aop:config>** tag benutzen

#### Name three typical cross cutting concerns

* Logging and Monitoring
* Exception Handling
* Transaction Management

#### Sample concerns in any Java Applications

* Calling a logging method after every method inside a class
* Performing security role check before a method call
* Calling monitoring methods before and after all methods inside a package

#### What two problems arise if you dont solve a cross cutting concern via AOP?

* Boiler plate and unnecessary code will be inside the application
* Verstreuter Code innerhalb der Anwendung, verhindert re-usability

### How to configure @Aspect annotation in XML and JavaConfig

#### What do you have to do to enable the detection of the @Aspect annotation?

##### Java-ConfigMan muss folgende Annotation an die JavaConfig Klasse anfügen, damit die @Aspect Annotationen der Beans berücksichtigt werden:

@EnableAspectJAutoProxy  
public class JavaConfig { … }

##### XML-Config

Man muss folgendes Tag in das XML-File hinzufügen:

<aop:aspectj-autoproxy />

### Learn the usage of Advices (After, Before, AfterReturning, AfterThrowing, Around)

#### How many advice types does Spring support. What are they used for?

##### Before advice (@Before)

@Before an eine Methode, bedeutet dass diese Methode die man definiert, noch vor der referenzierten Methode ausgeführt wird.

**Achtung: Falls hier eine Exception in beforeCalculating() fliegt, wird die andere Methode nicht gerufen!**

@Before("execution(\* com.goblingift.bean.UserBean.calculateRandom(\*))")

public void beforeCalculating() {

System.out.println("X1- beforeCalculating: DOING !");

}

Ausgabe: Zuerst X1 und danach wird die calculateRandom() aufgerufen

##### After returning advice (@AfterReturning)

@AfterReturning an einer Methode bedeutet, dass wenn diese referenzierte Methode fertig mit ihrer Abarbeitung ist, dann erst die hier definierte Methode aufgerufen wird.

**Achtung: Falls in der referenzierten Methode eine Exception fliegt, hindert dies, diese hier definierte Methode zur Ausführung!**

@AfterReturning(("execution(\* com.goblingift.bean.UserBean.calculateRandom(\*))")

public void afterCalculating() {

System.out.println("X2- afterCalculating done!");

}

Ausgabe: Zuerst wird die calculateRandom() ausgeführt, danach X2

##### After Throwing Advice

@AfterThrowing an einer Methode bedeutet, dass wenn die referenzierte Methode eine Exception wirft, dann wird diese hier definierte Methode ausgeführt. Kommt es in der referenzierten Methode zu keiner Exception, wird hier nix ausgeführt:

##### After (finally) advice

@After an einer Methode bedeutet, dass egal wie die referenzierte Methode ausgeführt wurde (Mit/Ohne Exception geworfen), danach wird die hier definierte Methode aufgerufen

##### Around advice

Mithilfe des @Around advices kann man selbst entscheiden ob die Target-Method auch ausgerufen wird oder nicht. Man kann somit eigenes Exception handling schreiben oder die Rückgabewerte selbst ändern:

@Around(USERBEAN\_METH\_REF)

public int aroundEverything(ProceedingJoinPoint joinPoint) throws Throwable {

double random = Math.random();

int randomInt = (int) (random \* 10);

System.out.println("Randomized: " + randomInt);

if (randomInt <5) {

return (int) joinPoint.proceed();

} else {

return 0;

}

}

#### Which advice do you have to use if you would like to try and catch exceptions?

* Around advice (try/catch exceptions)

*Falls wir eine @After und eine @AfterReturn Method definieren, so ist die Reihenfolge der Ausführung wie folgt (falls keine Exception geworfen wird) :*

*@After*

*@AfterReturn*

### Learn the usage of @After and @Before advices with code examples

*Siehe vorheriges Kapitel für Beispiele*

### Learn the usage of @AfterThrowing and @AfterReturning

Der **@AfterThrowing** advice kann mit verschiedenen Parametern angereichert werden, um Zugriff auf das JoinPoint Objekt und die Exception zu bekommen:

@AfterThrowing(value = USERBEAN\_METH\_REF, throwing = "exception")

public void afterThrowingException(JoinPoint jp, Exception exception) {

System.out.println("AfterThrowing advice running now...");

System.out.println("Method which crashed: " + jp.getSignature());

System.out.println("Calling object: " + jp.getTarget());

System.out.println("Params of method call: " + Arrays.asList(jp.getArgs()));

}

Die Ausgabe des Programms wäre wie folgt:

Quizbean is calculating...

AfterThrowing advice running now...

Method which crashed: void com.goblingift.bean.UserBean.calculateRandom(int)

Calling object: UserBean{username=passwGuest123!}

Params of method call: [29]

Der **@AfterReturning** advice kann verwendet werden, um nach dem erfolgreichen Ausführen einer Methode, auf z.B. den Rückgabewert zuzugreifen:

@AfterReturning(value = USERBEAN\_METH\_REF, returning = "ret")

public void afterCalculating(JoinPoint jp, int ret) {

System.out.println("Method returned following value:" + ret);

}

### @Around advice code example

Mithilfe des @Around advices kann man ziemlich hart in den Workflow einer Anwendung eingreifen.

Folgendes Beispiel simuliert eine Bank, welche verhindern will, dass Kunden mehr als 25€ auf einmal abheben – in diesem Fall wird eine Exception geworfen, welche den Programmfluss abbricht:

@Around(USERBEAN\_METH\_REF\_TAKEOFF)

public double aroundTakeOff(ProceedingJoinPoint joinPoint) throws Throwable {

if ( (double)Arrays.asList(joinPoint.getArgs()).get(0) >= 25.00) {

System.out.println("WARNING: User tried to take-off more than 25 bucks: " + joinPoint.getTarget());

throw new IllegalStateException("THATS TOO MUCH!");

}

return (double) joinPoint.proceed();

}

### AOP advice XML configurations

Um per XML-Datei aspects zu definieren, macht man am besten eine zweite XML-Config Datei und liest diese dann mit folgendem Befehl in der application-context.xml ein:

<import resource="classpath:aspect-config.xml"/>

**Die aspect-config.xml:**

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"

xmlns:aop="http://www.springframework.org/schema/aop"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-4.1.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-4.1.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-4.1.xsd">

<bean id="aspectManagerConfigClass" class="com.goblingift.aspect.AspectManager"/>

<aop:config>

<aop:aspect id="aspectManagerXml" ref="aspectManagerConfigClass">

<aop:pointcut id="afterCalculateRandomMethod" expression="execution(\* com.goblingift.bean.UserBean.calculateRandom(\*))" />

<aop:after method="afterMeth" pointcut-ref="afterCalculateRandomMethod"/>

</aop:aspect>

</aop:config>

</beans>

Wichtiges zur Config:

1. Oben müssen die Schema-Definitionen für Spring AOP eingebunden werden (gelb)
2. Die ganze AOP config muss mit dem <aop:config> tag umschlossen sein
3. Danach definiert man die pointcuts (welche wiederverwendbar sind)
4. Diese pointcuts nutzt man dann in den jeweiligen advice aspect tags, wie z.B. <aop:after …>

**@After & @AfterThrowing**

Um per XML-Config ein @After und @AfterThrowing advice zu schalten:

<aop:after method="afterMeth" pointcut-ref="calculateRandomMethod"/>

<aop:after-throwing method="afterThrow" throwing="ex" pointcut-ref="calculateRandomMethod"/>

Mithilfe des throwing parameters kann man definieren wie die Exception Variable heißt- siehe:

public void afterThrow(JoinPoint jp, Exception ex) {

System.out.println("AFTER THROW!!! " + ex.getMessage());

}

### How to configure and externalize Pointcuts, named pointcuts in AOP

Um einen Pointcut programmatisch zu definieren, erstellt man eine Methode:

@Pointcut("execution(\* com.goblingift.bean.UserBean.calculateRandom(\*))")

public void namedPointcutMethod() {}

Anschließend kann man für ein beliebigen advice den Methodennamen einsetzen:

@After("namedPointcutMethod()")

public void afterCalculating() {

System.out.println("XYZ: Method after");

}

### Understand the usage of Joinpoint and ProceedingJoinPoint

Mithilfe eines **JoinPoint**-Objekts bekommt man zusätzl. Informationen zu der Aufrufenden Methode raus, wie z.B. Methodensignatur, Parameter, Objekt selbst etc.

Ein **ProceedingJoinPoint** hat noch die zusätzl. Methode: **proceed()**

## Abschnitt 4: Spring JDBC, Transactions and ORM with Hibernate

### Understanding of checked, unchecked and Data Access Exception Hierarchy

#### Why do we (in Spring) prefer unchecked exceptions?

-Checked exceptions müssen gefangen werden oder mit throws hochgereicht werden

#### What is the data access exception hierarchy?

-DataAccessException is a runtime Exception

-Unchecked Exception, muss nicht gefangen werden

-Spring konvertiert alle SQL Exceptions in DataAccessException, um den Umgang zu erleichtern

-Beispiele für DataAccessExceptions: SQLSyntaxErrorException, SQLTransactionRollbackException…

### How to configure data sources with embedded database HSQLDB

#### Pom.xml

Als erstes fügen wir folgende Abhängigkeiten in die pom.xml:

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>4.3.7.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework.data</groupId>

<artifactId>spring-data-jpa</artifactId>

<version>1.11.3.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-orm</artifactId>

<version>4.3.7.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-tx</artifactId>

<version>4.3.7.RELEASE</version>

</dependency>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-entitymanager</artifactId>

<version>5.2.10.Final</version>

</dependency>

<dependency>

<groupId>org.hsqldb</groupId>

<artifactId>hsqldb</artifactId>

<version>2.4.0</version>

</dependency>

<dependency>

<groupId>javax.annotation</groupId>

<artifactId>jsr250-api</artifactId>

<version>1.0</version>

</dependency>

Anschließend erstellt man eine **application-context-jdbc.xml** Datei, worin die Beans definiert werden, um eine Datenbank-Verbindung herzustellen:

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName" value="org.hsqldb.jdbcDriver" />

<property name="url" value="jdbc:hsqldb:mem://localhost" />

<property name="username" value="sa" />

<property name="password" value="" />

</bean>

Dabei verwendet man die Spring Klasse DriverManagerDataSource und vergibt ihr die passenden Properties um sich auf die in-memory hsqldb zu verbinden. *(Multi-threaded-connection)*

#### Testdatenbank erstellen – in memory:

Anschließend fügt man in die application-context-jdbc.xml noch folgende statements ein, welche zwei SQL-Skripte ausführen:

<jdbc:initialize-database data-source="dataSource">

<jdbc:script location="classpath:dbschema.sql" />

<jdbc:script location="classpath:test-data.sql" />

</jdbc:initialize-database>

**Die SQL-Skripte:**

**dbschema.sql**

DROP TABLE IF EXISTS USER;

DROP TABLE IF EXISTS LOG;

CREATE TABLE IF NOT EXISTS USER (

IDUSER INT GENERATED BY DEFAULT AS IDENTITY (START WITH 1, INCREMENT BY 1) NOT NULL,

USERNAME VARCHAR(255) NOT NULL,

PASSWORD VARCHAR(255) NOT NULL,

ACTIVE BOOLEAN NOT NULL,

PRIMARY KEY (IDUSER)

);

CREATE TABLE IF NOT EXISTS LOG (

IDLOG INT GENERATED BY DEFAULT AS IDENTITY (START WITH 1, INCREMENT BY 1) NOT NULL,

LOGSTRING VARCHAR(1000) NULL,

PRIMARY KEY (IDLOG)

);

**test-data.sql**

**INSERT INTO USER (USERNAME, PASSWORD, ACTIVE) VALUES ('tunatore@admin.com', '12345', TRUE);**

**INSERT INTO USER (USERNAME, PASSWORD, ACTIVE) VALUES ('joe.bill@gmail.com', '12345', TRUE);**

**INSERT INTO USER (USERNAME, PASSWORD, ACTIVE) VALUES ('test@outlook.com', '12345', TRUE);**

**INSERT INTO LOG(LOGSTRING) VALUES('SAMPLE LOG');**

#### JDBC Template

JDBCTemplate ist eine Hilfsklasse von Spring, welche Boilercode verhindert und SQLExceptions mit weniger Coding behandelt. Mit dessen Hilfe kann man SQL Queries, Updates, Deletes etc. komfortabel ausführen.

Wir erstellen eine Klasse **DatabaseService**, welche eine javax.sql.DataSource beinhaltet (Z.B. die oben in der XML-Datei konfigurierte hsqldb):

public class **DatabaseService** {

private JdbcTemplate jdbcTemplate;

public DatabaseService(DataSource dataSource) {

this.jdbcTemplate = new JdbcTemplate(dataSource);

}

public JdbcTemplate getJdbcTemplate() {

return jdbcTemplate;

}

}

Anschließend erstellen wir noch eine UserManager Klasse- diese enthält Methoden um auf die Usertabelle zuzugreifen und z.B. Daten zu selektieren oder verändern:

public class UserManager {

private JdbcTemplate jdbcTemplate;

public UserManager(DatabaseService databaseService) {

this.jdbcTemplate = databaseService.getJdbcTemplate();

}

public void readWholeDatabase() {

jdbcTemplate.query("SELECT **\* FROM USER", (ResultSet rs) -> System.out.println(rs.getString(2))**);

}

}

Hierbei haben wir eine Methode readWholeDatabase definiert, welche durch die User in der Datenbank steppt und die 2. Spalte als String ausliest und ausgibt.

Zu letzt, laden wir diese beiden neuen Klassen als Beans in der Spring-Config und füllen jeweils die dependency mithilfe der Constructor-injection:

<bean id="databaseService" class="com.goblingift.orm.DatabaseService">

<constructor-arg ref="dataSource" />

</bean>

<bean id="userManager" class="com.goblingift.orm.UserManager">

<constructor-arg ref="databaseService" />

</bean>

#### Testen der Anwendung

Zum Testen kann man einen einfach Unit-Test schreiben, welcher die zuvor definierte Methode aufruft:

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration(locations = {"classpath:application-context-jdbc.xml"})

public class TestJDBCXml {

@Autowired

private ApplicationContext context;

@Test

public void doTest() {

UserManager userManager = context.getBean(UserManager.class);

userManager.readWholeDatabase();

}

}

Ausgabe:

tunatore@admin.com

joe.bill@gmail.com

[test@outlook.com](mailto:test@outlook.com)

#### Does the JDBC template acquire (and release) a connection for every method called or once per template?

Falls in der DataSource ein connection-pool definiert ist, kann die gleiche Connection in verschiedenen Methodenaufrufen verwendet werden.

Normalerweise erstellt er ein Connection-Objekt für jeden Methodenaufruf.

### How to use JDBCTemplate´s queryForList, queryForObject and queryForMap methods

#### queryForObject

Das RowMapper interface definiert eine Methode, um die Ergebnisse einer SQL-Abfrage, eines ResultSet in ein entsprechendes Objekt zu mappen. Beispiel:

@Component

public class UserMapper implements RowMapper<**User**> {

@Override

public **User** mapRow(ResultSet rs, int rowNum) throws SQLException {

User user = new User();

user.setIdUser(rs.getInt("IDUSER"));

user.setUsername(rs.getString("USERNAME"));

user.setPassword(rs.getString("PASSWORD"));

user.setActive(rs.getBoolean("ACTIVE"));

return user;

}

}

Anschließend kann man die queryForObject Methode auf dem JDBCTemplate Objekt aufrufen:

public User getUserById(int idUser) {

String sqlQuery = "SELECT \* FROM USER WHERE IDUSER=?";

User user = jdbcTemplate.queryForObject(sqlQuery, new Object[]{idUser}, userMapper);

return user;

}

#### queryForMap

Mithilfe der queryForMap Methode kann man sich eine Map<String, Object> der Ergebnismenge liefern lassen. Dabei ist String der Key (Spaltenname) und Object der Wert der Spalte:

public Map<String, Object> getUserByQueryForMap(int idUser) {

String sqlQuery = "SELECT \* FROM USER WHERE IDUSER=?";

return jdbcTemplate.queryForMap(sqlQuery, new Object[]{idUser});

}

Ausgabe:

{IDUSER=1, USERNAME=tunatore@admin.com, PASSWORD=12345, ACTIVE=true}

#### queryForList

Mithilfe dieser Methode kann man sich das Ergebnis eines SELECT statements zurückgeben lassen, welches mehr als ein Ergebnis findet:

public List<Map<String, Object>> getUsersByQueryForList() {

String sqlQuery = "SELECT \* FROM USER";

return jdbcTemplate.queryForList(sqlQuery);

}

Aufruf im Unit-Test:

List<Map<String, Object>> usersByQueryForList = userManager.getUsersByQueryForList();

for (Map<String, Object> actUser : usersByQueryForList) {

System.out.println("Another user found: " + actUser);

}

Ausgabe:

Another user found: {IDUSER=1, USERNAME=tunatore@admin.com, PASSWORD=12345, ACTIVE=true}

Another user found: {IDUSER=2, USERNAME=joe.bill@gmail.com, PASSWORD=12345, ACTIVE=true}

Another user found: {IDUSER=3, USERNAME=test@outlook.com, PASSWORD=12345, ACTIVE=true}

### How to use JDBCTemplate with UPDATE, DELETE and INSERT statements