



# Spring

Programmierung leichtgewichtiger Java-Anwendungen

# Administratives

- Kennenlernen
  - Wer bin ich?
  - Meine Tätigkeitsfelder
- Agenda
- Pausen
- Per “Sie“ oder per „du“?
- Maven oder Gradle?
- IntelliJ, VSCode oder Eclipse?

# Was ist Spring?

- Spring ist eine Familie von Frameworks, die eine Entwicklung komplexer Java Anwendungen ermöglichen.
- Spring erfindet dabei das Rad nicht immer wieder neu, sondern nutzt selbst etablierte Frameworks (z.B. Hibernate)
- Spring Boot stellt eine Konfiguration vieler Frameworks bereit
- Einzelne Spring Projekte ermöglichen über neue Annotation und Inversion of Control eine einfachere Nutzung von Frameworks.

# Projektfamilie

- <https://spring.io/projects>

The screenshot shows the Spring Projects page with the following content:

## Projects

From configuration to security, web apps to big data—whatever the infrastructure needs of your application may be, there is a Spring Project to help you build it. Start small and use just what you need—Spring is modular by design.

- Spring Boot**  
Takes an opinionated view of building Spring applications and gets you up and running as quickly as possible.
- Spring Framework**  
Provides core support for dependency injection, transaction management, web apps, data access, messaging, and more.
- Spring Data**  
Provides a consistent approach to data access – relational, non-relational, map-reduce, and beyond.
- Spring Cloud**  
Provides a set of tools for common patterns in distributed systems. Useful for building and deploying microservices.
- Spring Cloud Data Flow**  
Provides an orchestration service for composable data microservice applications on modern runtimes.
- Spring Security**  
Protects your application with comprehensive and extensible authentication and authorization support.
- Spring for GraphQL**  
Spring for GraphQL provides support for Spring applications built on GraphQL Java.
- Spring Session**  
Provides an API and implementations for managing a user's session information.

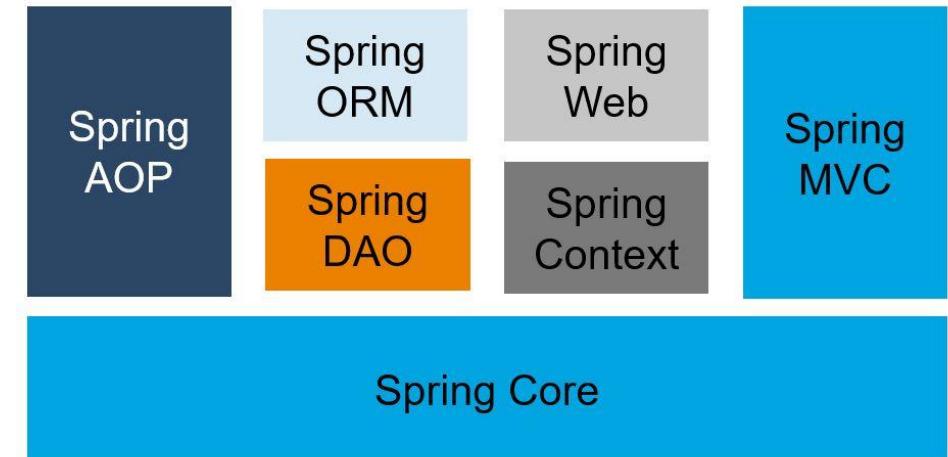
# Was ist Spring?

- Spring erstellt...
  - Anwendungen, die auf der JDK eines Servers laufen.
  - einzelne jar-Dateien, mit eingebettetem Server oder war-Dateien.
  - Standalone Anwendungen.
- Spring unterstützt auch Groovy und Kotlin.
- Aktuelle Spring Version: 6.1.4, benötigt Java 17 oder höher
- Aktuelle Spring Boot Version: 3.3.0
- Spring ist zunächst ein 2003 gestartetes Projekt, auf dem viele weiter Projekte basieren.

# Wichtige Module

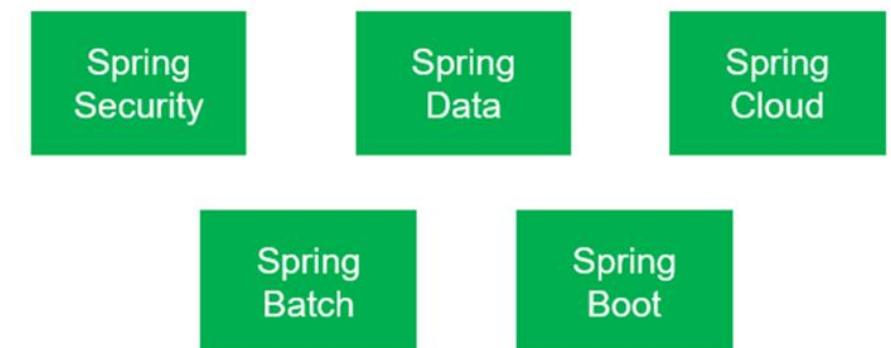
Modul	ermöglicht
Core	Dependency Injection
AOP	Aspektorientierte Programmierung
ORM	Objektrelationales Mapping (Hibernate)
DAO	Datenbankzugriff (JDBC) + Transaktionsmanagement
Web	Webanwendungen
Context	Erweiterung der BeanFactory um Validierung, EJB-Integration, Scheduling
MVC	Model-View-Controller Struktur + REST
Security	Diverse Sicherheitsaspekte
Data	Zugriff auf relationale und noSQL Datenbanken
Cloud	Microservices
Batch	Batch Jobs
Boot	Vereinfacht erstellen von Springapplikationen

Basispakete



Spring Core

wichtigste weitere Pakete



# Versionshistorie

- Versionen von Spring:

Version	Date	Notes
0.9	2003	
1.0	March 24, 2004	First production release.
2.0	2006	
3.0	2009	
4.0	2013	
5.0	2017	
6.0	November 16, 2022	

- 6.2.x is the upcoming feature branch (November 2024).
- 6.1.x is the main production line as of November 2023.
- 6.0.x is the previous production line, available since November 2022. This new framework generation comes with a JDK 17 and Jakarta EE 9 baseline.

# Versionshistorie – JDK Support

- Spring Framework 6.2.x: JDK 17-25 (expected)
- Spring Framework 6.1.x: JDK 17-23
- Spring Framework 6.0.x: JDK 17-21
- Spring Framework 5.3.x: JDK 8-21 (as of 5.3.26)
  
- 5.3.x is the final feature branch of the 5th generation, with long-term support provided on JDK 8, JDK 11, JDK 17 and the Java EE 8 level.

# Versionshistorie – Jakarta EE Versions

- Spring Framework 6.2.x: Jakarta EE 9-11 (jakarta namespace)
- Spring Framework 6.1.x: Jakarta EE 9-10 (jakarta namespace)
- Spring Framework 6.0.x: Jakarta EE 9-10 (jakarta namespace)
- Spring Framework 5.3.x: Java EE 7-8 (javax namespace)

# Philosophie

- Ermögliche eine Wahl auf jedem Level.
- Unterschiedliche Perspektiven berücksichtigen.
- Erhalte Rückwärtskompatibilität.
- Designe die API sorgfältig.
- Hohe Standards bei der Codequalität.

# Die Drei Säulen von Spring

- Spring Application Context (Context)
- Dependency Injection (DI)
- Aspekt-orientierte Programmierung (AOP)

# Spring Tools Suite (STS)

- Spring Tools Suite ist eine IDE (basierend auf Eclipse) bzw. Plugin für IDEs (VSCode und Theia)
- Die Werkzeuge der STS sind für IntelliJ Nutzer nur in der Ultimate Edition enthalten.
- Ermöglicht die Nutzung des Spring Initializer von der IDE aus.
- Verbessert Codecompletion und -navigation für Springanwendungen.
- Anzeige von Laufzeitinformationen.
- <https://spring.io/tools>

# Spring Initializr

- Erstelle auf <https://start.spring.io/> die Konfiguration für dein Projekt.
- Füge alle benötigten Dependencies hinzu.
- Projekt kann heruntergeladen, entpackt und mit einer IDE geöffnet werden (IntelliJ lädt dann automatisch).

The screenshot shows the Spring Initializr web application. At the top right is a toggle switch between sun and moon icons. The main header is "spring initializr". On the left is a menu icon (three horizontal lines). The interface is divided into several sections:

- Project**: Options for Gradle - Groovy (selected), Gradle - Kotlin, and Maven.
- Language**: Options for Java (selected), Kotlin, and Groovy.
- Spring Boot**: Options for 3.0.1 (SNAPSHOT), 3.0.0 (selected), 2.7.7 (SNAPSHOT), and 2.7.6.
- Project Metadata**: Fields for Group (com.example), Artifact (demo), Name (demo), Description (Demo project for Spring Boot), Package name (com.example.demo), and Packaging (Jar selected, War available). Below these are Java version options: 19, 17 (selected), 11, and 8.
- Dependencies**: A section showing "Lombok DEVELOPER TOOLS" as an annotation library for reducing boilerplate code. An "ADD ..." button is at the top right of this section.

At the bottom are social sharing icons for GitHub and Twitter, and three buttons: "GENERATE", "EXPLORE", and "SHARE...".

**spring initializr**

**Project**  
 Gradle - Groovy    Gradle - Kotlin    Maven

**Language**  
 Java    Kotlin    Groovy

**Spring Boot**  
 3.0.1 (SNAPSHOT)    3.0.0    2.7.7 (SNAPSHOT)    2.7.6

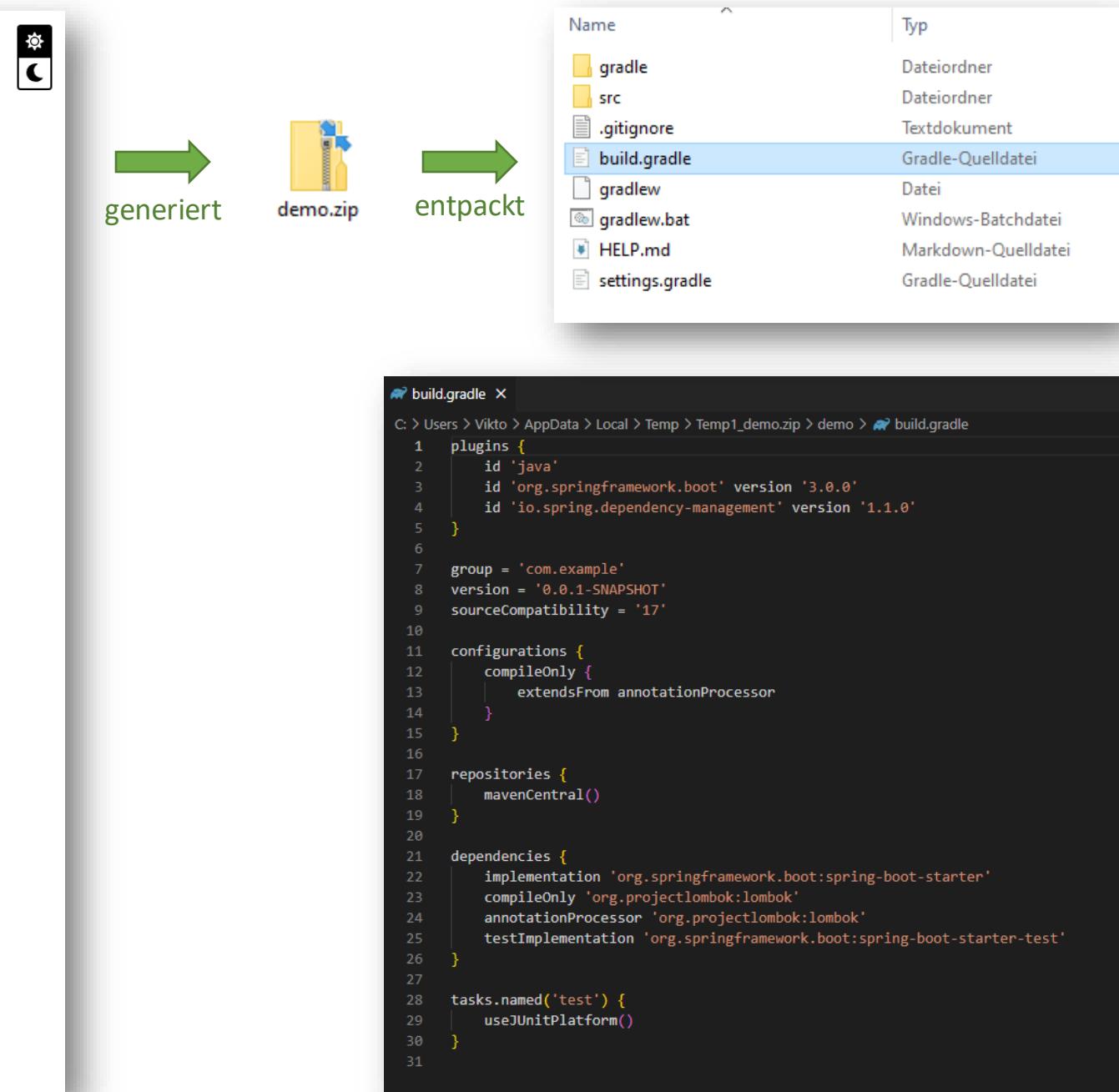
**Project Metadata**

Group	com.example
Artifact	demo
Name	demo
Description	Demo project for Spring Boot
Package name	com.example.demo
Packaging	<input checked="" type="radio"/> Jar <input type="radio"/> War
Java	<input type="radio"/> 19 <input checked="" type="radio"/> 17 <input type="radio"/> 11 <input type="radio"/> 8

**Dependencies** ADD ...

**Lombok** DEVELOPER TOOLS  
 Java annotation library which helps to reduce boilerplate code.

GENERATE EXPLORE SHARE...



# Übungsaufgabe

- Erstelle mit dem Spring Initializr ein Projekt mit den Dependencies:
  - Spring Web
  - h2
- Starte die Application.
- Gehe auf localhost:8080

# Inhalt eines neues Projektes

Dateien	Bedeutung
mvnw und mvnw.cmd	Maven Wrapper Skripte, mit denen das Projekt auch ohne installiertes Maven gebaut werden kann.
pom.xml	Maven Build Specifications
<projektname>Application.java	Main-Klasse von Spring Boot.
application.properties	Zunächst leer. Ermöglicht Einstellungen vorzunehmen.
static	Ordner in dem statische Inhalte gespeichert werden (Javascript/Bilder/css...)
templates	Ordner für template Dateien
<projektname>AplicationTests.java	Testklasse

# Dependency Injection (DI)

- DI ist ein Design Pattern, bei dem die Abhängigkeiten von Klassen und Instanzen nicht hart gecodet werden, sondern erst zur Compile- oder Laufzeit bestimmt werden.

# Tightly Coupled Code

- Konstruktor der Klasse Doctor erstellt automatisch eine Instanz der Klasse Qualification.

```
3 ► public class Main {  
4 ►     public static void main(String[] args) {  
5         Doctor doctor = new Doctor();  
6         doctor.assist();  
7     }  
8 }
```

```
3     public class Doctor {  
4         Qualification qualification;  
5  
6         public Doctor() {  
7             this.qualification = new Qualification();  
8         }  
9  
10        public void assist() {  
11            System.out.println("Doctor is assisting");  
12        }  
13    }
```

# Loosly Coupled Code

- Feld wird über Parameter des Konstruktors von außen bereitgestellt.
- Gut für Unitests
  - Klassen einzeln testen (mit Mocks)
  - Klassen gemeinsam testen
- Kleine Komponenten
- Austauschbarkeit

```
3 ► public class Main {  
4 ►     public static void main(String[] args) {  
5 |         Qualification qualification = new Qualification();  
6 |         Doctor doctor = new Doctor(qualification);  
7 |         doctor.assist();  
8 }  
9 }
```

```
3 | public class Doctor {  
4 |     Qualification qualification;  
5 |  
6 |     public Doctor(Qualification qualification) {  
7 |         this.qualification = qualification;  
8 |     }  
9 |  
10 |    public void assist() {  
11 |        System.out.println("Doctor is assisting");  
12 |    }  
13 |}
```

# Inversion of Control

- Spring wird das Zusammenstellen und Instanziieren der Klasseninstanzen übernehmen. (Inversion of Control)
- Wir Konfigurieren Spring mit
  - Annotation oder
  - xml-File
- Lade folgende Dependency in die pom.xml:

```
17   <dependencies>
18     <dependency>
19       <groupId>org.springframework</groupId>
20       <artifactId>spring-context</artifactId>
21       <version>6.0.2</version>
22     </dependency>
23   </dependencies>
```

# Dependency Injection mit xml

```
6 ► public class Main {  
7 ►   public static void main(String[] args) {  
8     ApplicationContext applicationContext  
9       = new ClassPathXmlApplicationContext(configLocation: "spring.xml");  
10  
11     Doctor doctor = applicationContext.getBean(Doctor.class);  
12     doctor.assist();  
13   }  
14 }
```

resources/spring.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>  
2 <beans xmlns="http://www.springframework.org/schema/beans"  
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
4   xsi:schemaLocation="http://www.springframework.org/schema/beans  
5     http://www.springframework.org/schema/beans/spring-beans.xsd">  
6   <bean id="doctor" class="de.reichert.Doctor"></bean>  
7 </beans>
```

# Übung

- Erstelle ein Maven (oder Gradle) Projekt.
- Erstelle die Doctor Klasse mit einer einfachen sout-Methode (ohne Qualification Klasse).
- Erstelle eine spring.xml und definiere Doctor als xml.
- Erstelle eine weiter Klasse Nurse, füge sie als Bean hinzu.
- Wie kann die Id einer Bean, statt der Klasse verwendet werden, um eine entsprechende Instanz zu erzeugen? Was muss dann beachtet werden?
- Wer ist für die Instanziierung der Objekte verantwortlich?

# Felder konfigurieren

- Felder können über die xml gesetzt werden.
- Nutze „value“ um eine statische Referenz anzuzeigen.
- Was passiert, wenn die getter und setter fehlen?

resources/spring.xml

```
<bean id="doctor" class="de.reichert.Doctor">
    <property name="qualification" value="PHD"/>
</bean>
```

```
3  public class Doctor {
4      private String qualification;
5
6      public String getQualification() {
7          return qualification;
8      }
9
10     public void setQualification(String qualification) {
11         this.qualification = qualification;
12     }
13 }
```

# Übung

- Füge ein Feld für eine Nurse-Instanz beim Doctor hinzu.
- Lasse das Feld von Spring füllen!
- Nutzt du „value“ oder „ref“?

# DI mit Konstruktorargumenten

- Die DI kann auch über Konstruktorargumente stattfinden.

resources/spring.xml

```
<bean id="doctor" class="de.reichert.Doctor">
    <constructor-arg name="nurse" ref="nurse"/>
    <constructor-arg name="qualification" value="PHD"/>
</bean>
```

```
3     public class Doctor implements Staff{
4         private String qualification;
5         private Nurse nurse;
6
7     public Doctor(String qualification, Nurse nurse) {
8         this.qualification = qualification;
9         this.nurse = nurse;
10    }
```

# DI mit @Component Annotation

- Über @Component kann angezeigt werden, dass die Klasse eine Bean beschreibt.
- Jedoch muss dann ein Scan hinzugefügt werden, damit diese gefunden wird.

resources/spring.xml

```
5  @Component
6  public class Doctor implements Staff{
7
8      @Override
9      ↑
10     public void assist() {
11         System.out.println("Doctor is assisting.");
12     }
13 }
```



```
<context:component-scan base-package="de.reichert"/>
```

# @Configuration statt xml

```
6  <@Configuration  
7  @ComponentScan(basePackages = "de.reichert")  
8  public class BeanConfig {  
9  }
```

```
7 ► public class Main {  
8 ►     public static void main(String[] args) {  
9  
10    ApplicationContext applicationContext  
11    = new AnnotationConfigApplicationContext(BeanConfig.class);  
12    //          = new ClassPathXmlApplicationContext("spring.xml");  
13  
14    Doctor doctor = applicationContext.getBean(Doctor.class);  
15    doctor.assist();  
16  }  
17 }
```

# @Bean

- In mit @Configuration annotierten Klassen können Beans mit @Beans an Methoden definiert werden.
- WICHTIG: @Component von Doctor entfernen!

```
7  @Configuration
8  public class BeanConfig {
9
10 @Bean
11  public Doctor doctor() {
12      return new Doctor();
13  }
14 }
```

# Übung

- Erstelle ein Feld Nurse im Doctor.
- Erstelle eine Funktion nurse(), die mit @Bean annotiert ist und eine Nurse zurückgibt.
- Passe die doctor() Methode an, sodass ein Doctor mit einer Nurse erstellt wird.

# @Autowired am Feld

- Die @Autowire-Annotation wird verwendet, um den IoC-Container dazu anzuhalten, eine Bean-Referenz automatisch zu injizieren.
- Kann gesetzt werden am (bestimmt, wann die Injektion stattfindet):
  - Feld
  - Konstruktor
  - Setter
- Sollte nicht an statischen Feldern gesetzt werden (kann zu Portierungsproblemen führen)

```
6  @Component
7  public class Doctor implements Staff{
8
9      @Autowired
10     private Nurse nurse;
11
12     @Override
13     public void assist() {
14         System.out.println("Doctor is assisting.");
15         nurse.assist();
16     }
17 }
```

# @Autowired am Setter

```
6  @Component
7  public class Doctor implements Staff{
8
9      private Nurse nurse;
10
11     @Autowired
12     public void setNurse(Nurse nurse) {
13         this.nurse = nurse;
14     }
15
16     @Override
17     public void assist() {
18         System.out.println("Doctor is assisting.");
19         nurse.assist();
20     }
21 }
```

# @Autowired am Konstruktor

- Annotation am Konstruktor ist optional. Wird nämlich als default angenommen.

```
6  @Component
7  public class Doctor implements Staff{
8
9      private Nurse nurse;
10
11     @Autowired
12     public Doctor(Nurse nurse) {
13         this.nurse = nurse;
14     }
15
16     @Override
17     public void assist() {
18         System.out.println("Doctor is assisting.");
19         nurse.assist();
20     }
21 }
```

```
6  @Component
7  public class Doctor implements Staff{
8
9      private Nurse nurse;
10
11     public Doctor(Nurse nurse) {
12         this.nurse = nurse;
13     }
14
15     @Override
16     public void assist() {
17         System.out.println("Doctor is assisting.");
18         nurse.assist();
19     }
20 }
```

# Übung

- Erstelle das Interface Staff, dass die Methode void assist() bereitstellt.
- Erstelle eine Klasse Handyman, die das Interface Staff implementiert.
- Lasse Doctor und Nurse das Interface implementieren.
- Erstelle in der Klasse Doctor ein Feld: private Staff staff. Annotiere es mit @Autowired.
- Was passiert?
- Was passiert, wenn du an eine der beiden Klassen die Annotation @Primary setzt?

# Wie wählt @Autowired aus?

1. Beans passenden Typs.
2. Beans mit passendem Qualifier. (@Qualifier)
3. Beans die als Primary markiert sind. (@Primary)
4. Name des Feldes
5. Wenn required=true ist, werfe Exception  
NoSuchBeanDefinitionException oder  
NoUniqueBeanDefinitionException, wenn mehrere Qualifier  
gefunden wurden.

# @Injekt oder @Autowired?

- `@Inject` stammt aus Java EE 6
- `@Autowired` stammt aus dem Spring-Framework
- Weitere Alternative ist `@Resource`, wird jedoch kaum genutzt.

# Optionen von @Autowired

- required (default true): Zeigt an, ob es eine entsprechende Bean geben muss.

# Welche Art von DI nutzen?

- Empfohlen wird die Nutzung von DI per Konstruktor.
- Sehr guter Artikel mit ausführlicher Begründung:  
<https://www.vojtechruzicka.com/field-dependency-injection-considered-harmful/>

# @Value

- Der Inhalt eines Feldes kann auch über @Value gesetzt werden.
- Dies kann an Feldern, Settern oder Konstruktorparametern gesetzt werden.

```
11  @Value("Alex")
12  public void setName(String name) {
13      this.name = name;
14 }
```

# Spring Expression Language

- <https://www.baeldung.com/spring-expression-language>
- Die Datei mit den Werten muss über `@PropertySource` gesetzt werden.
- Mit „,:“ kann ein Defaultvalue gesetzt werden.
- Mit `@PropertySource` können weitere Dateien definiert werden, aus denen Werte ausgelesen werden.

```
welcome.properties ×  
1 | welcome.message=Hallo Welt!  
2 |
```

```
12 @RestController  
13 @PropertySource("classpath:welcome.properties")  
14 public class HelloController {  
15  
16     @Value("${welcome.message}")  
17     private String welcome;  
18  
19     @GetMapping("/")  
20     public String helloWorld() {  
21         return welcome;  
22     }  
23 }
```

```
13     @Value("${name:Jochen}")  
14     private String name;
```

# @Value

- Der Value kann auch aus Umgebungsvariablen ausgelesen werden.
- Auf die selbe weise können Variablen aus der application.properties herausgelesen werden.

```
13     @Value("${system}")
14     private String name;
```

Environment variables:

```
system=Jannik
```



# @Scope

- Der Scope einer Bean bestimmt ihre Lebenszeit.
- Default ist „Singleton“, d.h.: von jeder Bean existiert in der JVM genau eine Instanz.
- <https://refactoring.guru/design-patterns/prototype>
- <https://refactoring.guru/design-patterns/singleton>

```
8  <Component
9  @Scope(scopeName = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
10 public class Doctor implements Staff{
11     private String name;
12
13     public void setName(String name) {
14         this.name = name;
15     }
16
17     @Override
18     public void assist() {
19         System.out.println("Doctor " + name + " is assisting.");
20     }
21 }
```

# @Scope

Typ	Bedeutung
Singleton (default)	Bean existiert im IoC Container genau ein Mal. Genutzt für stateless Beans.
Prototype	Es können mehrere Instanzen dieser Klasse im IoC Container existieren. Genutzt für statefull Beans. Es werden neue Instanzen mit dem Prototype Pattern angelegt.
Request	Wird im Rahmen jedes HTTP-Request erstellt und ist nur während diesem verfügbar.
Session	Wird im Rahmen jeder HTTP-Session erstellt und ist nur während diesem verfügbar.
Global	Wird im Rahmen jeder globalen HTTP-Session erstellt und ist nur während diesem verfügbar.
Custom	Man kann von allen Scopes außer Singleton und Prototype ableiten.

```
@Component
@Scope("singleton")
public class ExampleBean {
    // bean properties and methods
}
```

 Copy code

```
<bean id="exampleBean" class="com.example.ExampleBean"
scope="singleton">
    <!-- bean properties -->
</bean>
```

 Copy code

# @Scope

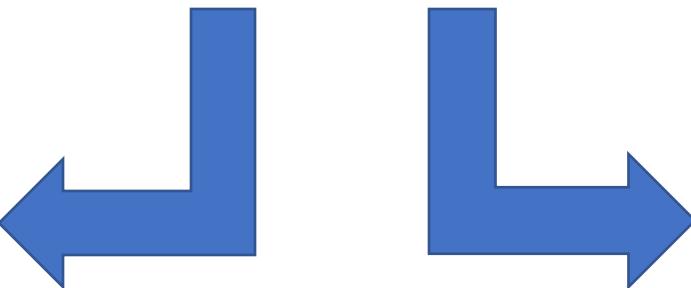
```
Doctor doctor = applicationContext.getBean(Doctor.class);
doctor.assist();
doctor.setName("Karl");
doctor.assist();
Doctor doctor1 = applicationContext.getBean(Doctor.class);
doctor1.assist();
```

Prototype

Doctor null is assisting.  
Doctor Karl is assisting.  
Doctor null is assisting.

Singleton

Doctor null is assisting.  
Doctor Karl is assisting.  
Doctor Karl is assisting.



# Spring Bean - Eigenschaften

- Spring Beans sind die Objekte, die vom Spring IoC Container verwaltet werden.
- id
- name
- Bean Klasse: Voll qualifizierter Klassename
- Scope: Bestimmt den Lifecycle einer Bean (singleton, prototype, request, session, oder global session)
- Konstruktorargumente: Welche Konstruktorargumente enthält eine Bean
- Autowiring: Legt fest, welche Beans zum Autowiring verwendet werden dürfen (no, byName, byType, constructor, autodetect)

# Bean Life Cycle

- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/BeanFactory.html>
- Bei der Erstellung und Löschung einer Bean werden viele verschiedene Schritte durchgegangen.
- Über Aware-Interfaces lassen sie in diese einsteigen und auch eigene init und destroy Methoden definieren.

# Aware Interfaces

- Klassen können verschiedene Aware-Interfaces implementieren, um in den Beanlifecycle einzusteigen.

```
10  @Component("arzt")
11  public class Doctor implements Staff, BeanNameAware {
12
13      @Override
14      public void setBeanName(String name) {
15          System.out.println("Bean name is: " + name);
16      }

```

# @PostConstruct & @PreDestroy

```
<dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>javax.annotation-api</artifactId>
    <version>1.3.2</version>
</dependency>
```

```
29     @PostConstruct
30     public void postConstruct() {
31         System.out.println("Doctor created with name: " + name);
32     }
33
34     @PreDestroy
35     public void preDestroy() {
36         System.out.println("soon, i will be gone.");
37     }
```

# Unitest mit Stateful Beans

- Sollte eine Bean in einem Test dauerhaft geändert werden, kann der Ursprüngliche Zustand über `@DirtiesContext` wiederhergestellt werden.

```
20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
```

```
@SpringBootTest  
@DirtiesContext(classMode = ClassMode.AFTER_EACH_TEST_METHOD)  
public class ShoppingCartTest {  
    @Autowired  
    ShoppingCart cart;  
  
    @Test  
    public void testAddOneItem() {  
        cart.addItem(new Item(name: "test", price: 20));  
        assertEquals(expected: 1L, cart.getItemCount());  
        assertEquals(expected: 20L, cart.getPrice());  
    }  
    @Test  
    public void testAddTwoItems() {  
        cart.addItem(new Item(name: "test1", price: 1));  
        cart.addItem(new Item(name: "test2", price: 2));  
        assertEquals(expected: 2, cart.getItemCount());  
        assertEquals(expected: 3, cart.getPrice());  
    }  
}
```

# ApplicationRunner & CommandLineRunner

- Zur Ausführung bestimmter Codezeilen können die Klassen CommandLineRunner oder ApplicationRunner genutzt werden.
- Annotiere die Implementierungen mit @Component, damit sie von Spring gefunden werden können.
- Der Inhalt wird dann automatisch von Spring durchgeführt.
- Best Practice: Nutze @Profile(..), damit die Runner nicht bei Tests durchgeführt werden.

```
1 @Component
2 public class AppStartupRunner implements ApplicationRunner {
3     private static final Logger logger = LoggerFactory.getLogger(AppStartupRunner.class);
4
5     @Override
6     public void run(ApplicationArguments args) throws Exception {
7         logger.info("Your application started with option names : {}", args.getOptionNames());
8     }
9 }
```

# Aspekt orientierte Programmierung AOP

- Implementierung von Crosscutting Concerns (Logging, Security, Transaktionen, Caching, Überwachung, Konfiguration,...) ist mit AOP so möglich, dass diese von der Businesslogik getrennt ist.
- Spring stellt einfache Möglichkeiten bereit, um AOP zu ermöglichen.
- Spring AOP nutzt dieselben Annotationen wie AspektJ.

# Dependencies für AOP

- Füge Dependencies hinzu.
- Füge @EnableAspectJAutoProxy Annotation bei einer Konfigurationsklasse hinzu.

```
30 <dependency>
31   <groupId>org.aspectj</groupId>
32   <artifactId>aspectjrt</artifactId>
33   <version>1.9.19</version>
34 </dependency>
35 <dependency>
36   <groupId>org.aspectj</groupId>
37   <artifactId>aspectjweaver</artifactId>
38   <version>1.9.19</version>
39 </dependency>
40
```

Bei XML Konfiguration:

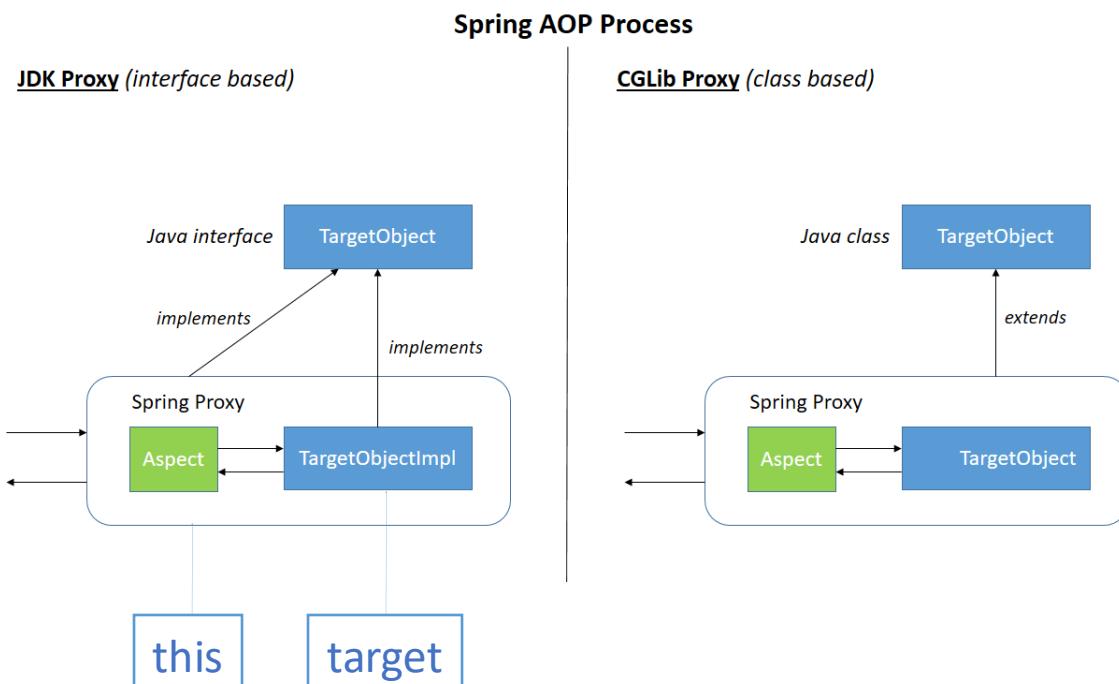
```
<aop:aspectj-autoproxy/>
```

# Grundbegriffe bei AOP

Begriff	Ganz kurz	Bedeutung
Aspect	Klasse, mit Cross Cutting Concern	Ein Codefeature, dass in vielen Stellen auftaucht und sich von der eigentlichen Businesslogik unterscheidet (z.B. Logging, Transactionsmanagement). Jeder Aspekt beschreibt eine „cross-cutting functionality“. Die Klasse, die das cross-cutting functionality implementiert, wird mit @Aspect und @Component gekennzeichnet.
Joinpoint	Wo setzt der Aspekt an?	Spezieller Punkt in der Programmausführung (z.B. Methodenaufruf, Konstruktoraufruf. In SpringAOP beinhaltet das Joinpoint-Objekt wird alle Informationen über den Methodenaufruf beinhalten (this-Referenz, Parameter, Signatur usw.)
Advice	Einzelne neue Funktionalität	Aktionen, die auf einem bestimmten Joinpoint ausgeführt werden. Dies werden die Methoden sein, die die cross-cutting functionality implementieren.
Pointcut	Wer ist vom Aspekt betroffen?	Gibt an, welche JoinPoints welche Advices durchführen sollen. Wird über Pointcut Designator Language gesteuert.
Target Object	Das Betroffene Objekt	Das Objekt, auf dem Advices durchgeführt werden sollen. Spring erstellt ein Proxy basierend auf dem Target Objekt.
AOP Proxy	„Objekt + Aspekt“	Proxy, dass das Target Objekt umfasst und um Advices erweitert.
Weaving	Aspektverkettung	Prozess, der mehrere Advices zusammenstellt.

# Spring AOP Proxy

- Wenn die Klasse auch nur ein Interface implementiert, wird JDK Proxy von Spring AOP per default verwendet. Ansonsten CGLib Proxy.
- <https://refactoring.guru/design-patterns/proxy>



# Joinpoint

- Spring AOP unterstützt nur Joinpoints bei der Methodenausführung.

Joinpoint	Spring AOP Supported	AspectJ Supported
Method Call	No	Yes
Method Execution	Yes	Yes
Constructor Call	No	Yes
Constructor Execution	No	Yes
Static initializer execution	No	Yes
Object initialization	No	Yes
Field reference	No	Yes
Field assignment	No	Yes
Handler execution	No	Yes
Advice execution	No	Yes

# Übersicht

```
@Aspect  
@Component  
public class MyAspect {  
  
    @Pointcut("execution(* com.example.service.*.*(..))")  
    public void serviceMethods() {}  
  
    @Before("serviceMethods()")  
    public void beforeAdvice() {  
        // Code to be executed before service methods  
    }  
  
    @After("serviceMethods()")  
    public void afterAdvice() {  
        // Code to be executed after service methods  
    }  
}
```

**Der Aspekt**

Pointcut:  
Welche Methodenausführungen sind betroffen?

Wann soll der Advice ausgeführt werden?  
Hier kann auch direkt ein Pointcut-Ausdruck stehen.

**Der Advice**

# Advice Typen

Typ	Annotation	Aspekt wird ...
Before Advice	@Before	vor dem Joinpoint ausgeführt. Kann die Methodenausführung nicht unterbrechen, außer durch das werfen einer Exception.
After returning Advice	@AfterReturning	nach der normalen Durchführung der Methode ohne Exception ausgeführt.
After throwing Advice	@AfterThrowing	nach der Durchführung mit Exceptionrückgabe ausgeführt.
After (finally) Advice	@After	nach jeder Durchführung der Methode durchgeführt (Exception oder normal)
Around Advice	@Around	statt der Methode durchgeführt. Joinpoint muss hier manuell durchgeführt werden. Aspekt muss ProceedingJoinPoint als Parameter haben und hat einen Rückgabewert.

```
@Around("serviceMethods()")
public Object aroundAdvice(ProceedingJoinPoint joinPoint) throws Throwable {
    // Code to be executed before the service method
    Object result = joinPoint.proceed();
    // Code to be executed after the service method
    return result;
}
```

# Pointcut Designation Language

## Pointcut Designators

<b>execution</b>	<b>angegebene Methode.</b>
<b>within</b>	<b>Alle Methoden in einer Klasse.</b>
<b>this</b>	Alle Methoden, bei denen die Proxy eine bestimmte Klasse hat.
<b>target</b>	Alle Methoden, bei denen das Target Object die bestimmte Klasse.
<b>args</b>	<b>Argumente haben den angegeben Typ. Die Argumente können dann als Parameter des Aspekts angegeben werden.</b>
<b>@target</b>	Alle Methoden, bei denen die Klasse des Target Objekts die angegebene Annotation aufweist.
<b>@args</b>	Alle Methoden, bei denen die Parameter die angegebene Annotation aufweist.
<b>@within</b>	Alle Methoden einer Klasse, die die angegebene Annotation aufweist.
<b>@annotation</b>	<b>Alle Methoden, die die angegebene Annotation aufweisen.</b>
<b>bean</b>	Alle Methoden der angegebenen Bean.

# args

- Mit args können schneller auf bestimmte Parameter zugegriffen werden.
- Joinpoint kann dabei hinzugefügt werden oder nicht.
- Bei Pointcuts muss der entsprechende Parameter weitergereicht werden.

```
@Before("targetShoppingCart() && args(item)")  
public void shoutItem(JoinPoint joinPoint, Item item) {  
    System.out.println(item.getName() + " was touched!");  
}
```

```
@Before("targetShoppingCart() && args(item)")  
public void shoutItem(Item item) {  
    System.out.println(item.getName() + " was touched!");  
}
```

```
@Pointcut("args(item)")  
public void itemParameter(Item item) {}  
  
@Before("targetShoppingCart() && itemParameter(item)")  
public void shoutItem(Item item) {  
    System.out.println(item.getName() + " was touched!");  
}
```

# Pointcuts verknüpfen

- Pointcuts können mit „**&&**“ (und), „**||**“(oder) und „**!**“(nicht) verknüpft werden.
- [Genaue Semantik](#)
- [Beispiele](#)

```
@Pointcut("execution(public * *(..))")  
private void anyPublicOperation() {} ❶
```

```
@Pointcut("within(com.xyz.myapp.trading..*)")  
private void inTrading() {} ❷
```

```
@Pointcut("anyPublicOperation() && inTrading())")  
private void tradingOperation() {} ❸
```

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?name-pattern(param-pattern)  
throws-pattern?)
```

# AspectJ und Spring AOP

Spring AOP	AspectJ
Implemented in pure Java	Implemented using extensions of Java programming language
No need for separate compilation process	Needs AspectJ compiler (ajc) unless LTW is set up
Only runtime weaving is available	Runtime weaving is not available. Supports compile-time, post-compile, and load-time Weaving
Less Powerful – only supports method level weaving	More Powerful – can weave fields, methods, constructors, static initializers, final class/methods, etc...
Can only be implemented on beans managed by Spring container	Can be implemented on all domain objects
Supports only method execution pointcuts	Support all pointcuts
Proxies are created of targeted objects, and aspects are applied on these proxies	Aspects are weaved directly into code before application is executed (before runtime)
Much slower than AspectJ	Better Performance
Easy to learn and apply	Comparatively more complicated than Spring AOP

# Spring Initializr

- Erstelle auf <https://start.spring.io/> die Konfiguration für dein Projekt.
- Füge alle benötigten Dependencies hinzu.
- Projekt kann heruntergeladen, entpackt und mit einer IDE geöffnet werden (IntelliJ lädt dann automatisch).

The screenshot shows the Spring Initializr web application. At the top right is a toggle switch between sun and moon icons. The main header is "spring initializr". On the left is a menu icon (three horizontal lines). The interface is divided into several sections:

- Project**: Options for Gradle - Groovy (selected), Gradle - Kotlin, and Maven.
- Language**: Options for Java (selected), Kotlin, and Groovy.
- Spring Boot**: Options for 3.0.1 (SNAPSHOT), 3.0.0 (selected), 2.7.7 (SNAPSHOT), and 2.7.6.
- Project Metadata**: Fields for Group (com.example), Artifact (demo), Name (demo), Description (Demo project for Spring Boot), Package name (com.example.demo), and Packaging (Jar selected, War option available). Below these are Java version options: 19, 17 (selected), 11, and 8.
- Dependencies**: A section showing "Lombok DEVELOPER TOOLS" as an annotation library for reducing boilerplate code. An "ADD ..." button is at the top right of this section.
- Bottom Navigation**: Icons for GitHub and Twitter, and buttons for "GENERATE", "EXPLORE", and "SHARE...".

**spring initializr**

**Project**  
 Gradle - Groovy    Gradle - Kotlin    Maven

**Language**  
 Java    Kotlin    Groovy

**Spring Boot**  
 3.0.1 (SNAPSHOT)    3.0.0    2.7.7 (SNAPSHOT)    2.7.6

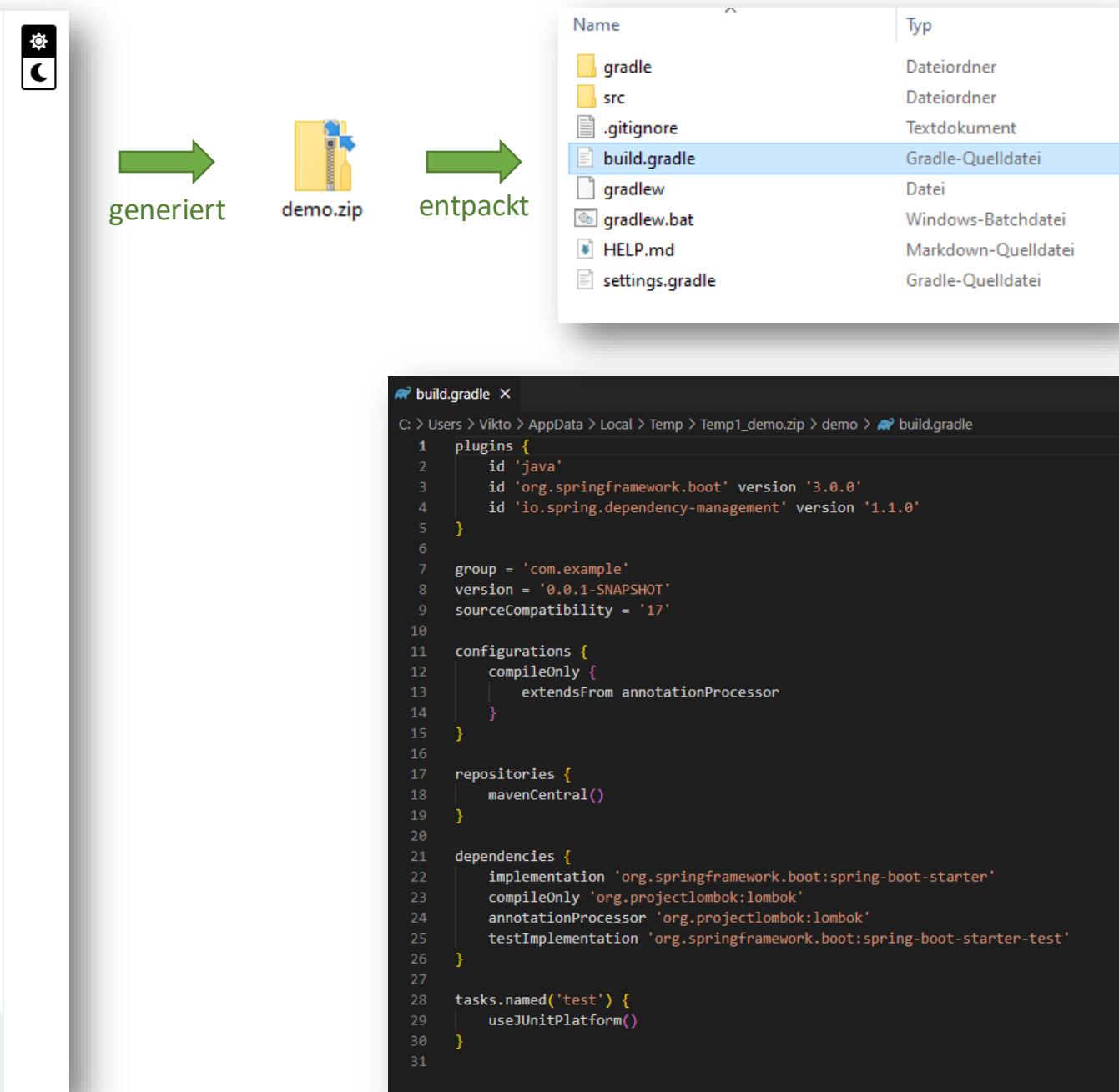
**Project Metadata**

Group	com.example
Artifact	demo
Name	demo
Description	Demo project for Spring Boot
Package name	com.example.demo
Packaging	<input checked="" type="radio"/> Jar <input type="radio"/> War
Java	<input type="radio"/> 19 <input checked="" type="radio"/> 17 <input type="radio"/> 11 <input type="radio"/> 8

**Dependencies** ADD ...

**Lombok** DEVELOPER TOOLS  
 Java annotation library which helps to reduce boilerplate code.

GENERATE EXPLORE SHARE...



# Übungsaufgabe

- Erstelle mit dem Spring Initializr ein Projekt mit den Dependencies:
  - Spring Web
  - h2
- Starte die Application.
- Gehe auf localhost:8080

# Inhalt eines neues Projektes

Dateien	Bedeutung
mvnw und mvnw.cmd	Maven Wrapper Skripte, mit denen das Projekt auch ohne installiertes Maven gebaut werden kann.
pom.xml	Maven Build Specifications
<projektname>Application.java	Main-Klasse von Spring Boot.
application.properties	Zunächst leer. Ermöglicht Einstellungen vorzunehmen.
static	Ordner in dem statische Inhalte gespeichert werden (Javascript/Bilder/css...)
templates	Ordner für template Dateien
<projektname>AplicationTests.java	Testklasse

# Spring Boot

- Ziel: Rapid Application Development
- Spring Boot Starter Dependencies (Zusammenstellung gemeinsam funktionsfähiger Libraries in einer xml)
- Ermöglicht automatische Konfiguration (z.B. Hibernate) des Projektes.
- Embedded Server (im jar-File enthalten)
- Spring Boot CLI (Command Line Interface)
- Spring Boot Actuator (Einblick in den Application Context)

Entwicklung mit Spring Boot ist heute Standard.

# @SpringBootApplication

Annotation	Herkunft	Bedeutung
@Target(ElementType.TYPE)	Java	Annotation kann an Klassen, Interfaces, Enums, Annotationen und Records angewandt werden.
@Retention(RetentionPolicy.RUNTIME)	Java	Annotation wird zur Laufzeit beachtet.
@Documented	Java	Die Existenz der Annotation an einem Element wird in die Javadoc des Elements aufgenommen.
@Inherited	Java	Annotation gilt auch für Unterklassen der annotierten Klasse.
@SpringBootConfiguration	Spring	Kennzeichnet eine Klasse als Konfigurationsklasse. (Spezialisierung der @Configuration Annotation)
@EnableAutoConfiguration	Spring	Ermöglicht Spring Boot die automatische Konfiguration (z.B. von Hibernate).
@ComponentScan(...)	Spring	Erlaubt Componenten Scanning. Über Annotation wie @Component, @Controller, @Service können Componenten für Spring sicht- und nutzbar gemacht werden.

```
23  @Target({ElementType.TYPE})
24  @Retention(RetentionPolicy.RUNTIME)
25  @Documented
26  @Inherited
27  @SpringBootConfiguration
28  @EnableAutoConfiguration
29  @ComponentScan(
30      excludeFilters = {@Filter(
31          type = FilterType.CUSTOM,
32          classes = {TypeExcludeFilter.class}
33      ), @Filter(
34          type = FilterType.CUSTOM,
35          classes = {AutoConfigurationExcludeFilter.class}
36      )}
37  )
38  public @interface SpringApplication {
```

# jar und war

- jar (Java Archive) verpackt unser Projekt, sodass es als Bibliothek, Anwendung, Plugin usw. genutzt werden kann.
- war (Web Application Archive oder Web Application Resource) ist nur für die Auslieferung an Servlet/JSP Containern vorgesehen. Es wird ein Server benötigt, um es auszuführen.
- war ist immer Jakarta/Java EE kompatibel.
- Spring Boot stellt uns bei der Kompilierung in jar einen konfigurierten Tomcat Server zur Verfügung. Wir können also auch Webanwendungen als jar bauen.

# ErsterRestController

- `@GetMapping(“/”)` kann auch durch `@RequestMapping(value = “/”, method = RequestMethod.GET)` erreicht werden (altmodisch)
- In der `application.properties` können diverse Einstellungen vorgenommen werden.

```
9  @RestController
10 public class HelloController {
11
12     @GetMapping(“/”)
13     public String helloWorld() {
14         return "Hello World";
15     }
16 }
```

```
application.properties ×
1 server.port=8082
2
```

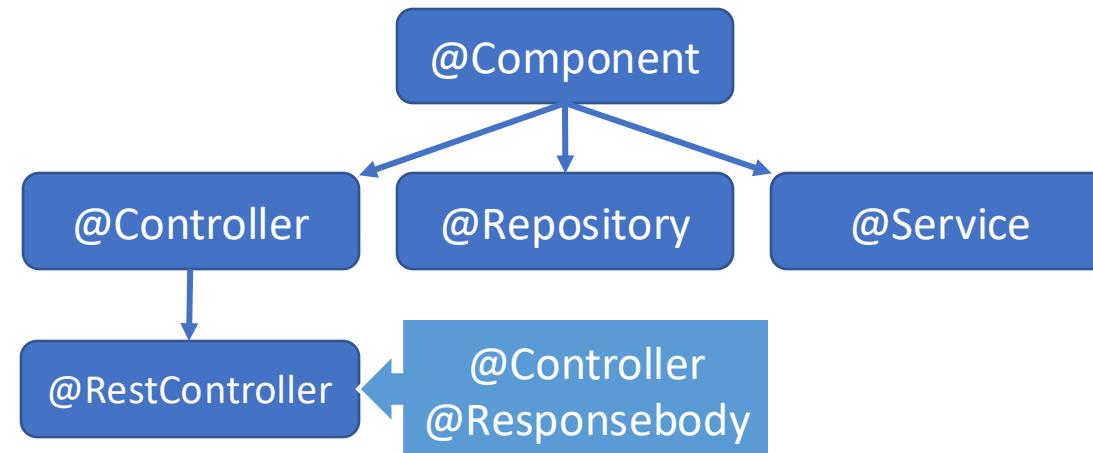
```
C:\Users\Vikto\IdeaProjects\Spring-boot-tutorial> mvn spring-boot:run
```

# Problembehandlung bei belegten Port

- Ports durchsuchen: lsof -i -P | grep LISTEN
- kill -9 <Prozessnummer>
- In Powershell:
  - Get-Process -Id (Get-NetTCPConnection -LocalPort <Portnummer> -State Listen).OwningProcess
  - Stop-Process -Id <Prozessnummer> -Force

# Spring Stereotypes

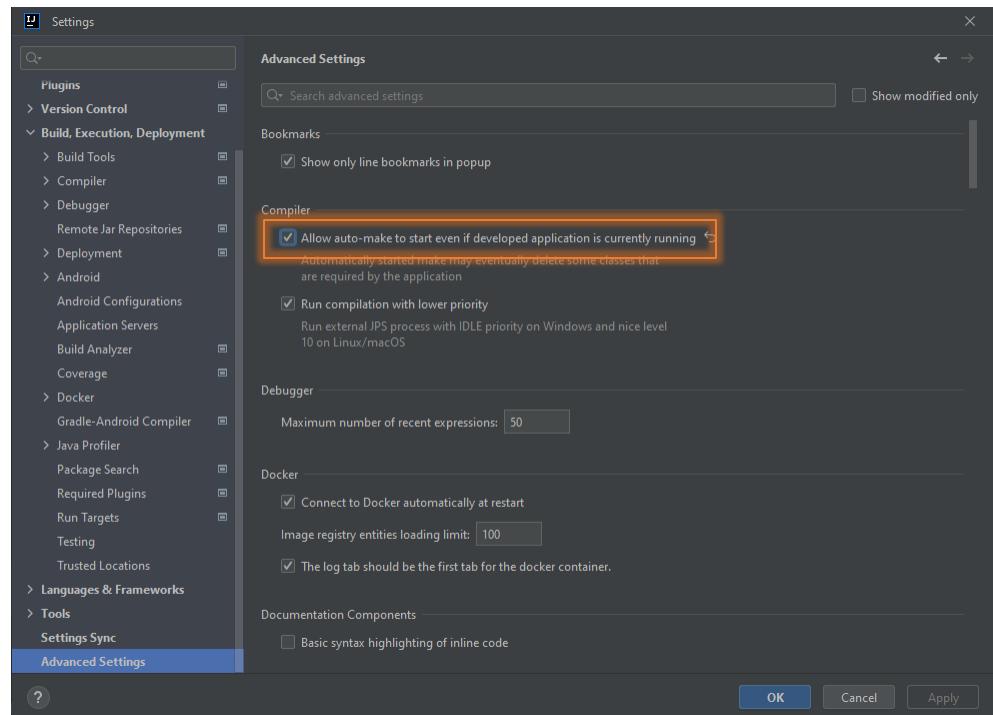
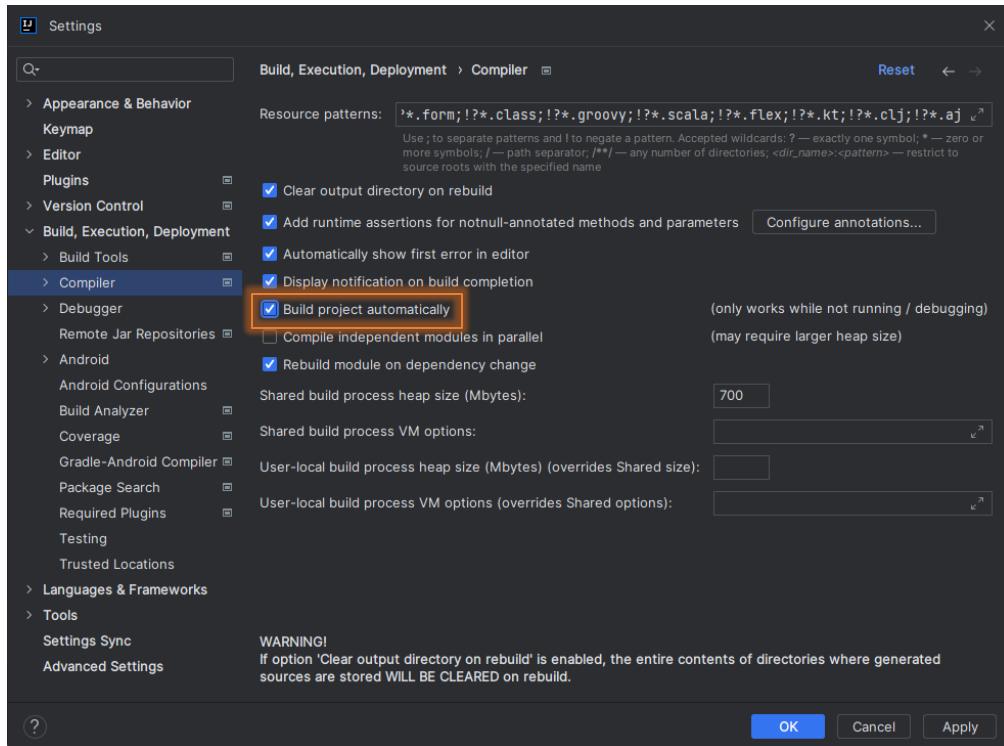
- Spring Stereotypes sind Klassenannotationen, um Beans zu definieren.
  - `@Component`
  - `@Controller`
  - `@RestController`
  - `@Repository`
  - `@Service`
  - ...



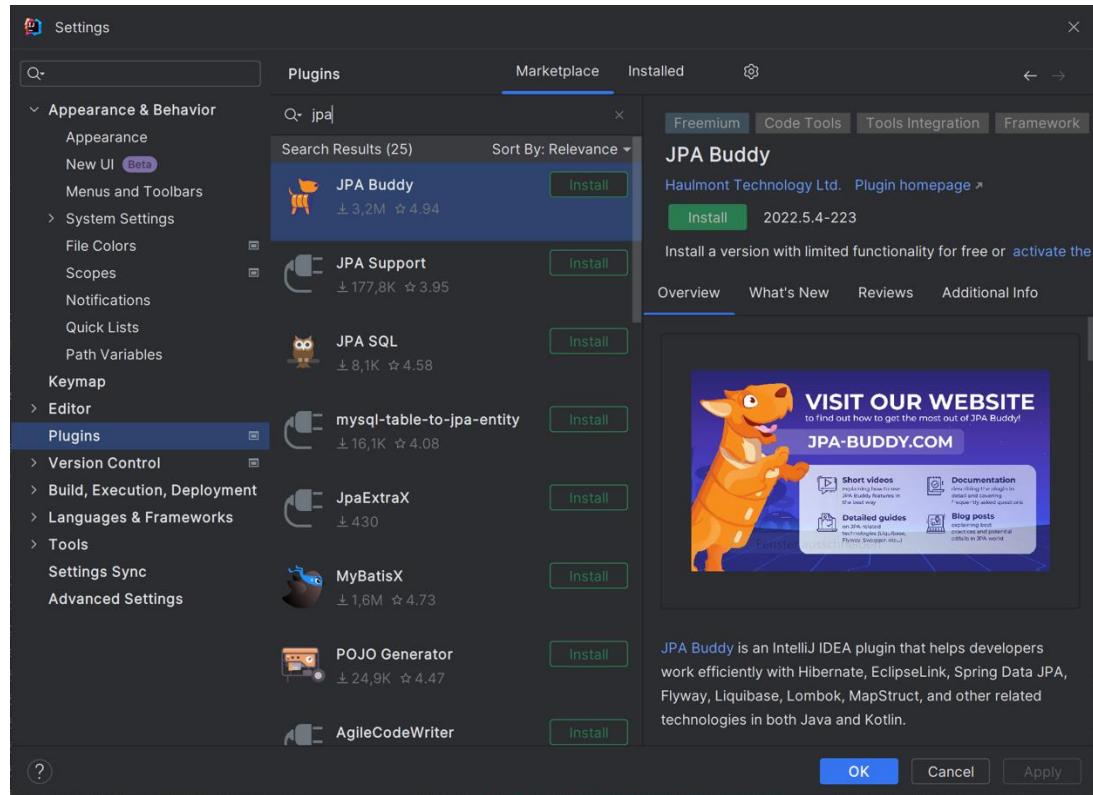
# Autorestart bei IntelliJ aktivieren

pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
</dependency>
```



# JPA Buddy



- Das Plugin JPA Buddy unterstützt bei der schnellen Erstellung von DAOs (Data Access Object).

# h2-Datenbank einrichten

- Eine H2-Datenbank lässt sich leicht mit diesen Konfigurationen einrichten.
- Für `spring.jpa.hibernate.ddl-auto` gibt es die Optionen:
  - none
  - create
  - create-drop
  - validate
  - update

## application.properties

```
3  spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
4  spring.jpa.hibernate.ddl-auto=create-drop
5  spring.jpa.show-sql=true
6  spring.jpa.properties.hibernate.format_sql=true
7
8  spring.datasource.url=jdbc:h2:file:./database/h2db;AUTO_SERVER=true
9  spring.datasource.username=sa
10 spring.datasource.password=
11 spring.datasource.driver-class-name=org.h2.Driver
12
13 spring.h2.console.enabled=true
```

# Einstellungen

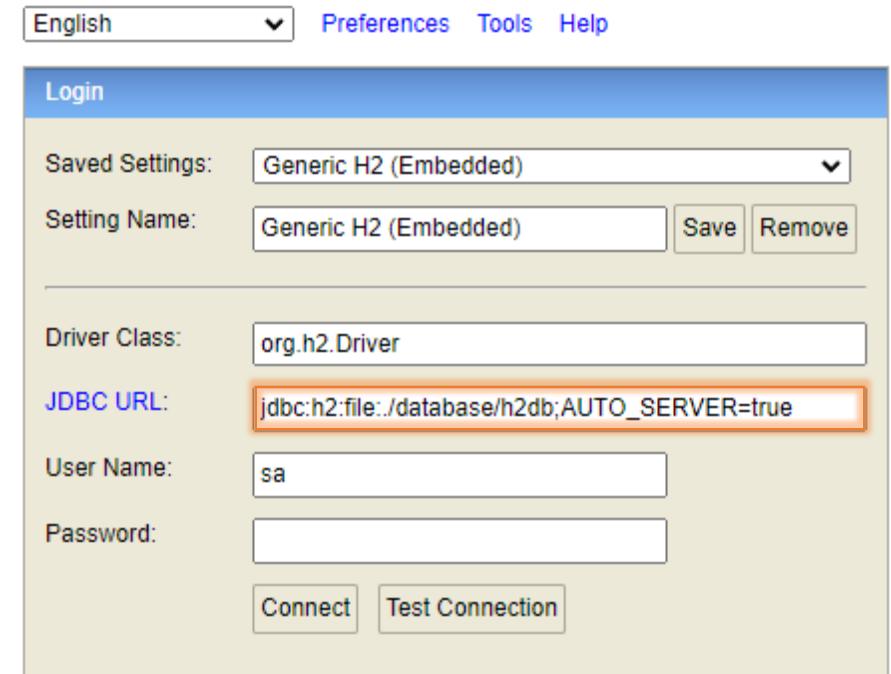
## application.properties

```
3 spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
4 spring.jpa.hibernate.ddl-auto=create-drop
5 spring.jpa.show-sql=true
6 spring.jpa.properties.hibernate.format_sql=true
7
8 spring.datasource.url=jdbc:h2:file:./database/h2db;AUTO_SERVER=true
9 spring.datasource.username=sa
10 spring.datasource.password=
11 spring.datasource.driver-class-name=org.h2.Driver
12
13 spring.h2.console.enabled=true
```

spring.data.url	URL der Datenbank, AUTO_SERVER=true dient dazu, dass mehrere Anwendungen einfach auf die DB zugreifen können.
spring.data.username	Username bei der Datenbank
spring.data.password	Password bei der Datenbank
spring.jpa.hibernate.ddl-auto	<a href="#">Link</a> Nur mit Hibernate nutzbar. Einstellung, wie bei Programmstart mit der Datenbank verfahren werden soll. (none(default), create, create-drop, validate, update)
spring.jpa.show-sql	Zeige die SQL-Statements an, die Hibernate erzeugt
spring.jpa.properties.hibernate.dialect	SQL-Dialekt einstellen.
spring.jpa.properties.hibernate.format_sql	Boolean, ob SQL formatiert werden sollen.
spring.h2.console.enabled	h2 Datenbank über Browser durchsuchbar.

# H2-console

- localhost:8082/h2-console
- Wird von Spring Boot geliefert.
- spring.h2.console.enabled=true



# Entität erstellen

- `@Entity` zeigt an, dass zu dieser Klasse eine Datenbanktabelle gehört.
- `@Id` zeigt Primary Key an und `@GeneratedValue`, dass dieser Wert nicht vom Nutzer gesetzt werden soll, sondern automatisch generiert wird.

```
8  @Entity
9  public class Department {
10
11    @Id
12    @GeneratedValue(strategy = GenerationType.AUTO)
13    private Long id;
14
15    private String name;
16    private String address;
17    private String code;
18
19    public Department() {
20    }
21
22    // Konstruktoren, Getter, Setter,...
```

```
Hibernate:
drop table if exists department cascade
Hibernate:
drop sequence if exists department_seq
Hibernate: create sequence department_seq start with 1 increment by 50
Hibernate:
create table department (
    id bigint not null,
    address varchar(255),
    code varchar(255),
    name varchar(255),
    primary key (id)
)
```

# Eigenen Generator erstellen

- Eigene Generatoren können erstellt werden, indem das Interface IdentifierGenerator implementiert wird.
- Der Generator kann dann mit der @GenericGenerator Annotation benannt werden und dann als Generator angegeben werden.
- Generator muss Threadsafe sein.

```
14  public class CourseIdGenerator implements IdentifierGenerator {  
15      private static final AtomicLong counter = new AtomicLong(initialValue: 0);  
16  
17      @Override  
18  ↗      public String generate(SharedSessionContractImplementor session,  
19                               Object object) throws HibernateException {  
20          return "C" + counter.getAndIncrement();  
21      }  
22  }
```

```
10  ↘ @Entity  
11  @GenericGenerator(name = "custom_id_gen",  
12                      strategy = "com.example.training.model.CourseIdGenerator")  
13  ↗ public class Course {  
14  
15  ↘     @Id  
16  @GeneratedValue(generator = "custom_id_gen")  
17  ↗     private String id;
```

# @Table & @Column & @Transient

- @Table kann genutzt werden, um den Tabellennamen einzustellen, Unique Constraints zu setzen und anderes.
- @Column kann genutzt werden, um die Spalte anzupassen. (Name, Nullable, usw.)
- @Transient schließt ein Feld in der Datenbankerstellung aus.

```
5  <div>@Entity
6  <div>@Table(name = "dep")
7  <div>public class Department {
8
9    <div>@Id
10   <div>@GeneratedValue(strategy = GenerationType.AUTO)
11   <div>private Long id;
12
13  <div>@Column(name = "dep_name", nullable = false)
14  <div>private String name;
```

# Repository erstellen

- Erstelle ein Interface, dass JpaRepository ableitet.
- Die Generischen Typen sind die Entity und der Typ des Keys.
- Annotiere es mit @Repository.

```
7  @Repository
8  public interface DepartmentRepository extends JpaRepository<Department, Long> {
9 }
```

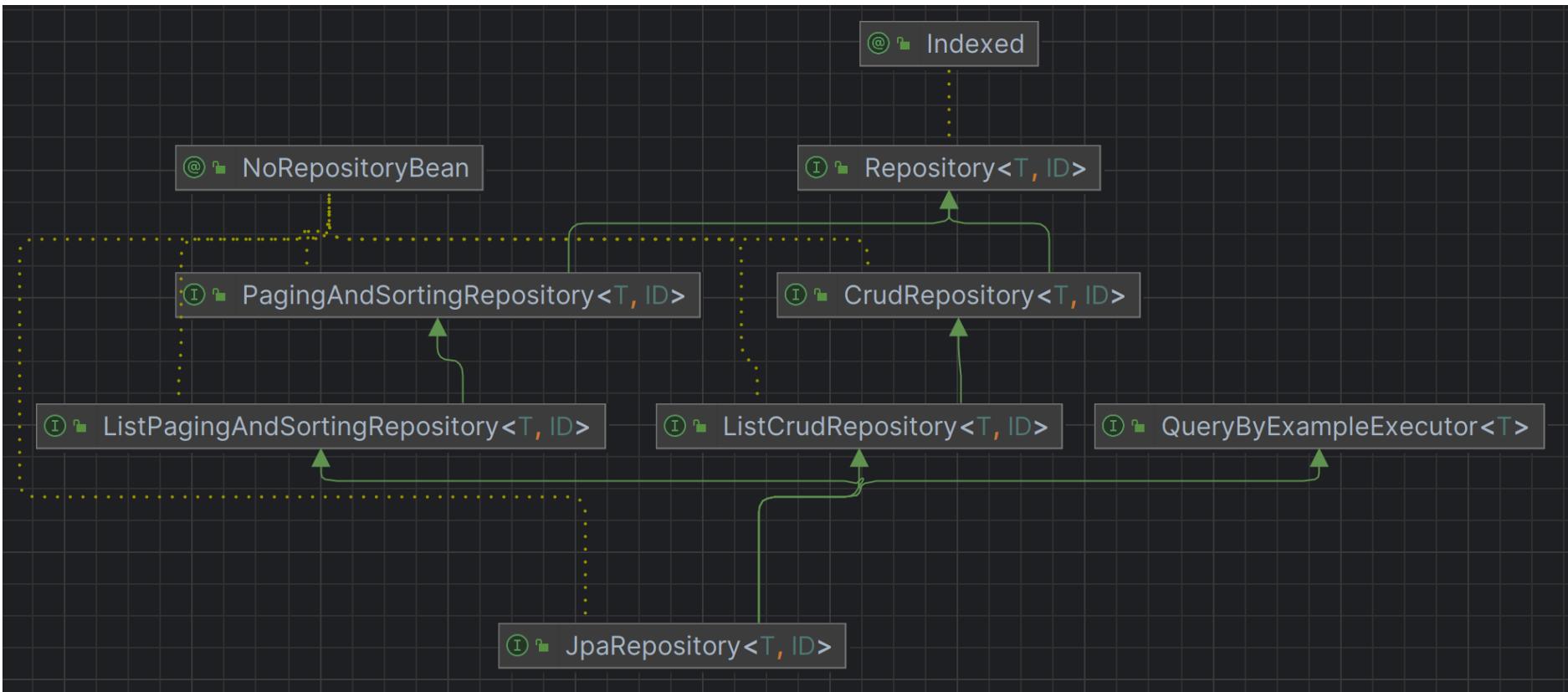
# JpaRepository

- Stellt den Zugriff auf die Datenbank bereit und die Übersetzung der Zeilen in Pojos (Plain Old Java Objects)
- JpaRepository stellt bereits eine große Menge an Funktionen bereit.
- Wir können dieses Interface auch erweitern.

```
▽ ① JpaRepository
  Ⓜ flush(): void
  Ⓜ saveAndFlush(S): S
  Ⓜ saveAllAndFlush(Iterable<S>): List<S>
  Ⓜ deleteInBatch(Iterable<T>): void
  Ⓜ deleteAllInBatch(Iterable<T>): void
  Ⓜ deleteAllByIdInBatch(Iterable<ID>): void
  Ⓜ deleteAllInBatch(): void
  Ⓜ getOne(ID): T
  Ⓜ getById(ID): T
  Ⓜ getReferenceById(ID): T
  Ⓜ findAll(Example<S>): List<S> ↑QueryByExampleExecutor
  Ⓜ findAll(Example<S>, Sort): List<S> ↑QueryByExampleExecutor
  Ⓜ saveAll(Iterable<S>): List<S> →ListCrudRepository
  Ⓜ findAll(): List<T> →ListCrudRepository
  Ⓜ findAllById(Iterable<ID>): List<T> →ListCrudRepository
  Ⓜ save(S): S →CrudRepository
  Ⓜ saveAll(Iterable<S>): Iterable<S> →CrudRepository
  Ⓜ findById(ID): Optional<T> →CrudRepository
  Ⓜ existsById(ID): boolean →CrudRepository
  Ⓜ findAllById(Iterable<ID>): Iterable<T> →CrudRepository
  Ⓜ count(): long →CrudRepository
  Ⓜ deleteById(ID): void →CrudRepository
  Ⓜ delete(T): void →CrudRepository
  Ⓜ deleteAllById(Iterable<? extends ID>): void →CrudRepository
  Ⓜ deleteAll(Iterable<? extends T>): void →CrudRepository
  Ⓜ deleteAll(): void →CrudRepository
  Ⓜ findAll(Sort): List<T> →ListPagingAndSortingRepository
```

# Vererbungshirarchie von JpaRepository

- Ggf. genügt auch schon ein CrudRepository.



# Erste Zeilen hinzufügen

```
Department{id=null, name='Vetrieb', address='Haus 11', code='123'}  
Hibernate:  
    select  
        next value for dep_seq  
Hibernate:  
    insert  
    into  
        dep  
        (address, code, dep_name, id)  
    values  
        (?, ?, ?, ?)  
Department{id=1, name='Vetrieb', address='Haus 11', code='123'}
```

```
9  @SpringBootApplication  
10 public class SpringBootTutorialApplication {  
11  
12     public static void main(String[] args) {  
13         ConfigurableApplicationContext context = SpringApplication.run(SpringBootTutorialApplication.class, args);  
14  
15         DepartmentRepository departmentRepository = context.getBean(DepartmentRepository.class);  
16  
17         Department department0 = new Department( name: "Vetrieb", address: "Haus 11", code: "123");  
18         System.out.println(department0);  
19  
20         department0 = departmentRepository.save(department0);  
21         System.out.println(department0);  
22  
23     }  
24  
25 }  
26 }
```

# Service Layer hinzufügen

- Hier befindet sich die Businesslogik.
- @Service nur eine andere Bezeichnung für @Component.
- Definiere ein Interface und eine konkrete Implementierung.
- Das Repository wird über IoC von Spring zur Verfügung gestellt.

```
6  public interface DepartmentService {  
7  }  
8  
9
```

```
8  @Service  
9  public class DepartmentServiceImpl implements DepartmentService{  
10    @Autowired  
11    private DepartmentRepository departmentRepository;  
12  }
```

# Übersicht

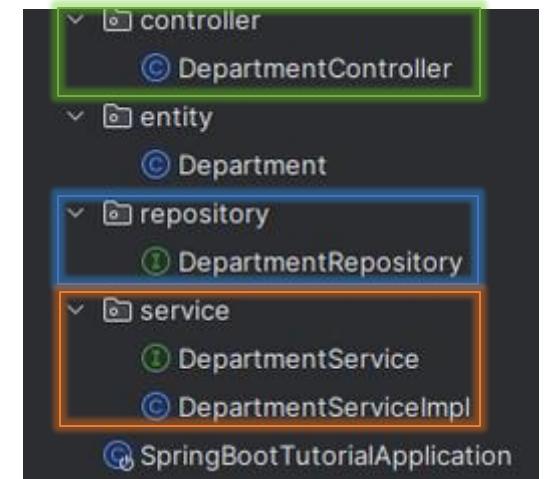


```
10  @RestController
11  public class DepartmentController {
12      @Autowired
13      private DepartmentService departmentService;
14  }
```

```
6  public interface DepartmentService {
7  }
```

```
8  @Service
9  public class DepartmentServiceImpl implements DepartmentService{
10     @Autowired
11     private DepartmentRepository departmentRepository;
12  }
```

```
7  @Repository
8  public interface DepartmentRepository extends JpaRepository<Department, Long> {
9  }
```



H2-Datenbank

# @GetMapping



```
13  @RestController
14  public class DepartmentController {
15      @Autowired
16      private DepartmentService departmentService;
17
18      @GetMapping("/department")
19      public List<Department> getAllDepartments() {
20          return departmentService.getAllDepartments();
21      }
22  }
```

```
public interface DepartmentService {
    List<Department> getAllDepartments();
}
```

```
10  @Service
11  public class DepartmentServiceImpl implements DepartmentService{
12      @Autowired
13      private DepartmentRepository departmentRepository;
14
15      @Override
16      public List<Department> getAllDepartments() {
17          return departmentRepository.findAll();
18      }
19  }
```

```
7  @Repository
8  public interface DepartmentRepository extends JpaRepository<Department, Long> {
9  }
```

H2-Datenbank

# Post Methode aufrufen

IntelliJ IDEA

```
107 ▶ POST http://localhost:8080/students
108   Content-Type: application/json
109
110   {"name": "Khatereh", "matrikelNumber": "789"}
```

PowerShell

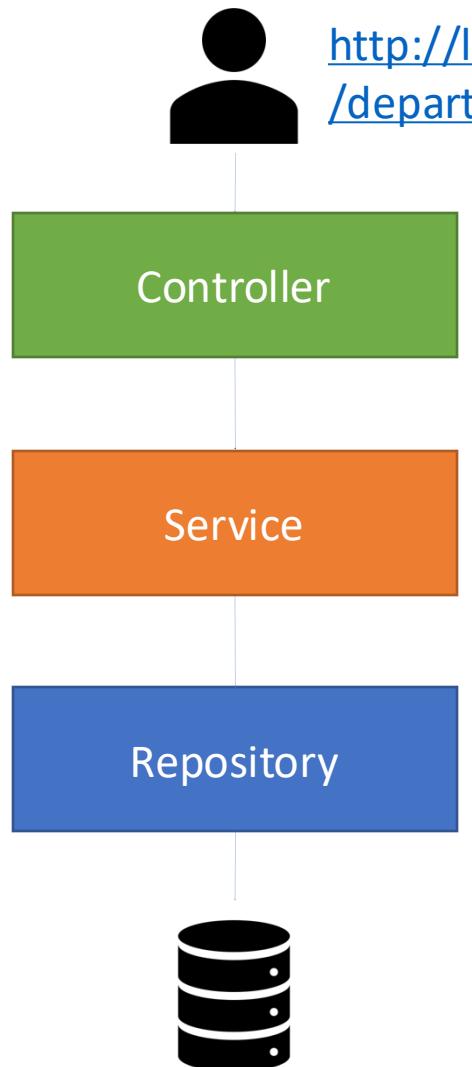
```
PS C:\Users\Vikto> $Body = @{name="housekeeping"
>>   address="House 42"
>>   code="abcd123"}
PS C:\Users\Vikto> $json = $body | ConvertTo-Json
PS C:\Users\Vikto> Invoke-RestMethod http://localhost:8080/department -Method post -Body $json -ContentType "application/json"

id name      address  code
-- --      -----  --
2 housekeeping House 42 abcd123
```

curl

```
curl -d '{"key1":"value1", "key2":"value2"}' -H "Content-Type: application/json" -X POST http://localhost:3000/data
```

# @GetMapping mit Parameter @PathVariable



```
@GetMapping("/{id}")
public Optional<Department> fetchDepartmentById(@PathVariable("id") Long id) {
    return departmentService.fetchDepartmentById(id);
}
```

```
13  ⏵ |     Optional<Department> fetchDepartmentById(Long id);
```

```
26
27 ⏴ |     @Override
28 |     public Optional<Department> fetchDepartmentById(Long id) {
29 |         return departmentRepository.findById(id);
|     }
```

```
7  @Repository
8  public interface DepartmentRepository extends JpaRepository<Department, Long> {
9 }
```

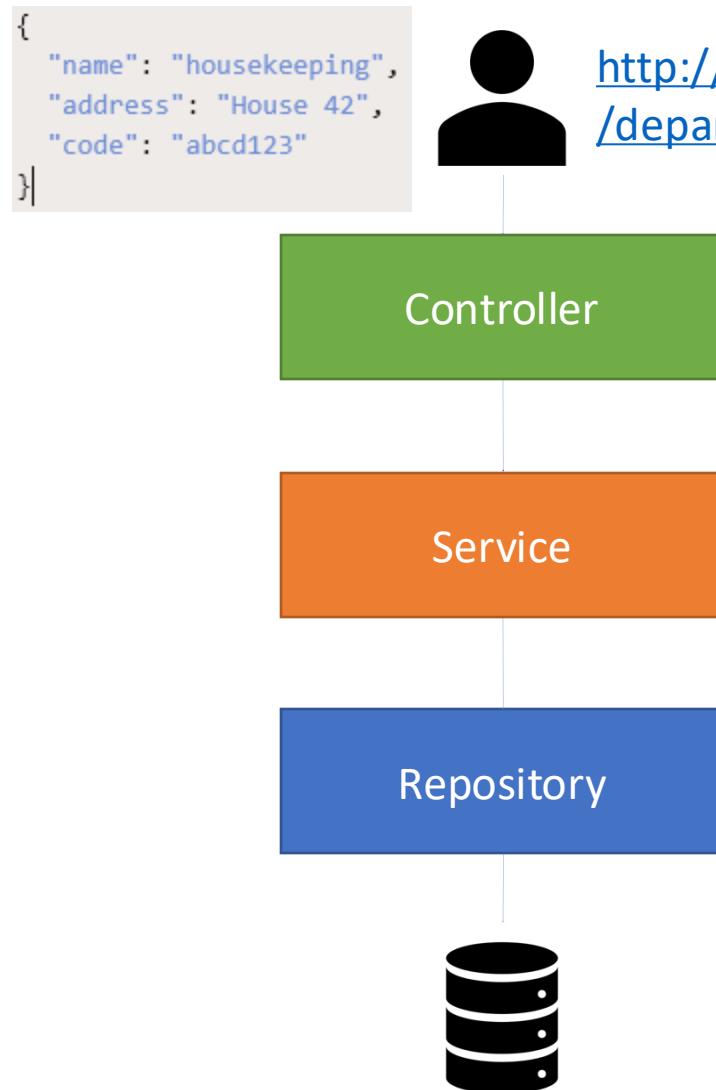
H2-Datenbank

# @GetMapping mit Parameter

## @RequestParam

- Alternativ kann auch @RequestParam genutzt werden.
- Die Anfrage hat dann z.B. die Form  
<http://localhost:8080/department?id=1>
- Eine Liste für einen Parameter kann mit Komma getrennt angegeben werden.
- Mehrere Parameter werden mit „&“ getrennt.
- Optionalität kann über Parameter konfiguriert werden oder über Optional angezeigt werden.
- Es können Defaults definiert werden.

# @PostMapping mit Json



```
16  @GetMapping("/department")  
17  public List<Department> getAllDepartments() {  
18      return departmentService.getAllDepartments();  
19  }
```

```
9  Department saveDepartment(Department department);
```

```
16  @Override  
17  public Department saveDepartment(Department department) {  
18      return departmentRepository.save(department);  
19  }
```

```
7  @Repository  
8  public interface DepartmentRepository extends JpaRepository<Department, Long> {  
9  }
```

H2-Datenbank

# Objekte prüfen

- Über Hibernate Validation gibt es die Möglichkeit Felder zu annotieren und festzulegen, welche Felder auf welche Art und Weise gesetzt werden müssen, damit diese als gültige Objekte gelten.
- Im Controller wird mit @Valid angezeigt, dass das Objekt diesen Anforderungen gehorchen muss.
- Liste der Constraints

```
47 <dependency>
48   <groupId>org.springframework.boot</groupId>
49   <artifactId>spring-boot-starter-validation</artifactId>
50 </dependency>
```

```
@NotBlank(message = "Please add a Department Name.")
private String name;
```

▶ POST <http://localhost:8080/department>  
Content-Type: application/json  
  
{"address": "hier123", "code": "234"}

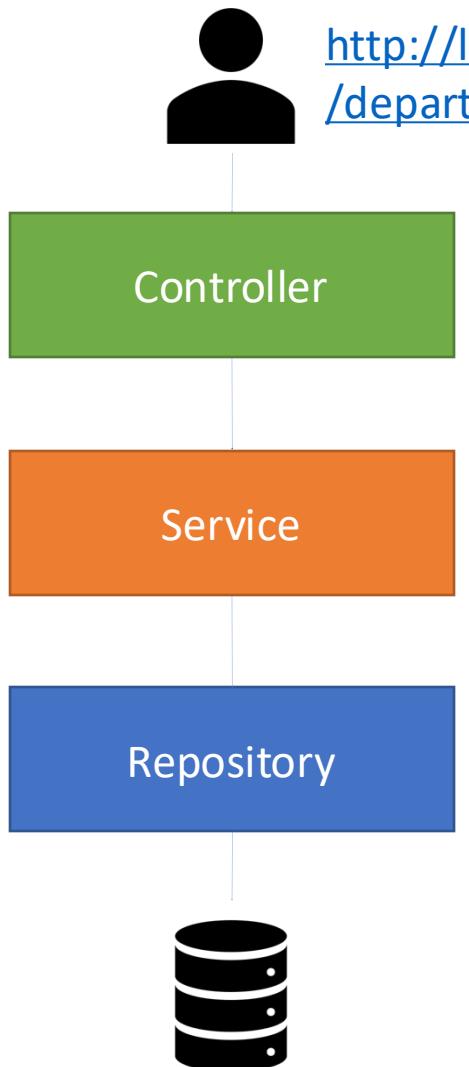
```
34 @PostMapping("/department")
35 public Department saveDepartment(
36   @Valid @RequestBody Department department) {
37     return departmentService.saveDepartment(department);
38 }
```

# Fehler von Hibernate Validation richtig zurückgeben.

- Da @Valid eine MethodArgumentNotValidException wirft und diese die eigentlich relevante Nachricht beinhaltet, fügen wir folgenden Code hinzu:

```
60      @ResponseStatus(HttpStatus.BAD_REQUEST)
61      @ExceptionHandler(MethodArgumentNotValidException.class)
62      @
63      public List<ObjectError> handleValidationExceptions(
64          MethodArgumentNotValidException ex) {
65              return ex.getBindingResult().getAllErrors();
66      }
```

# @DeleteMapping



```
31 @DeleteMapping("/department/{id}")
32 public String deleteDepartmentById(@PathVariable("id") Long id) {
33     boolean foundAndDeleted = departmentService.deleteDepartmentById(id);
34     return foundAndDeleted ? "Deleted Department " + id + " successfully."
35     : "No Department with " + id + " found.";
36 }
```

```
15 boolean deleteDepartmentById(Long id);
```

```
31 @Override
32 public boolean deleteDepartmentById(Long id) {
33     if (!departmentRepository.existsById(id)) return false;
34     departmentRepository.deleteById(id);
35     return true;
36 }
```

```
7 @Repository
8 public interface DepartmentRepository extends JpaRepository<Department, Long> {
9 }
```

H2-Datenbank

# Hibernate Validation

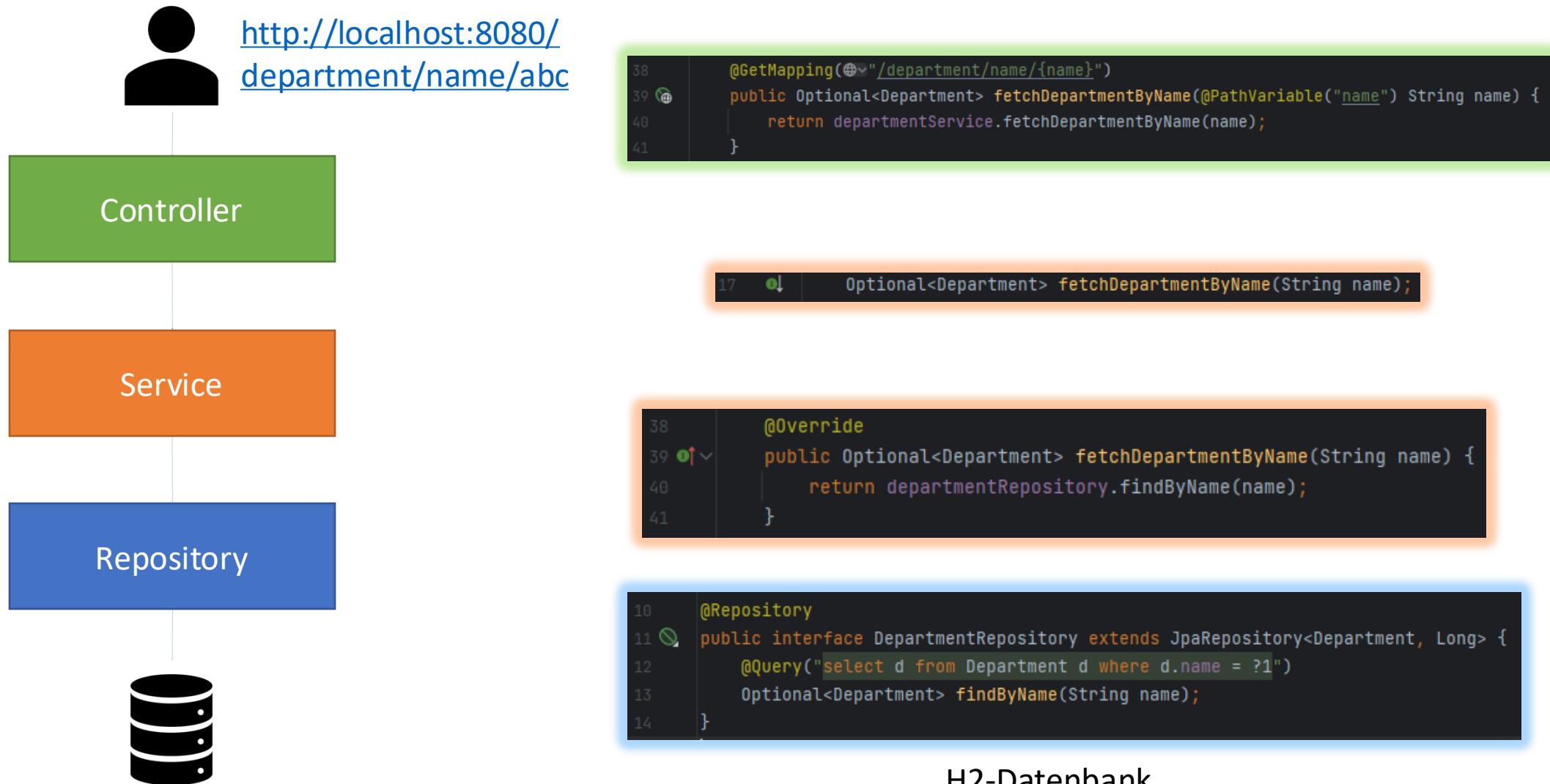
- Um sicherzustellen, dass bestimmte gesetzt sind und Regeln gehorchen, können diese mit Annotationen validiert werden.
- Anfragen, die diese Validierung beachten sollen können ihre Parameter mit @Valid kennzeichnen.
- Es gibt sehr viele Annotationen:
  - NotBlank
  - Length
  - Size
  - Email
  - Positive
  - Negativ
  - ...

```
46          <!-- Hibernate Validation, LOOK INSIDE-->
47  ⏷      <dependency>
48
49
50      ⚡  <groupId>org.springframework.boot</groupId>
           <artifactId>spring-boot-starter-validation</artifactId>
           </dependency>
```

```
15      @NotNull(message = "Please add a Department Name.")
16  ⏷    private String name;
```

```
22      @PostMapping("/department")
23  ⏷    public Department saveDepartment(@Valid @RequestBody Department department) {
24
25        return departmentService.saveDepartment(department);
}
```

# Neue Queries definieren



# @Query

- Innerhalb der Query wird die [Java Persistence Query Language \(JPQL\)](#) verwendet.
- Parameter können auch mit „:name“ direkt an die Parameter gebunden werden.
- Man kann auch Queries [ohne @Query Annotation](#) schreiben. ([Keywords](#))
- Mit der Option nativeQuery=true kann auf den Dialekt der Datenbank umgestellt werden. Achtung: die Queries sind dann ggf. nicht mehr zwischen den Datenbanken kompatibel!

```
10  @Repository
11  public interface DepartmentRepository extends JpaRepository<Department, Long> {
12
13      @Query("select d from Department d where d.name = ?1")
14      Optional<Department> findByName(String name);
}
```

```
10  @Repository
11  public interface DepartmentRepository extends JpaRepository<Department, Long> {
12
13      Optional<Department> findByName(String name);
}
```

```
public interface UserRepository extends JpaRepository<User, Long> {

    @Query(value = "SELECT * FROM USERS WHERE LASTNAME = ?1",
           countQuery = "SELECT count(*) FROM USERS WHERE LASTNAME = ?1",
           nativeQuery = true)
    Page<User> findByLastname(String lastname, Pageable pageable);
}
```

# Add Logger

- Wir nutzen den slf4j Logger.
- Andere Logger können konfiguriert werden.
- Alle Einstellungen können in application.properties vorgenommen werden.
- Ausgabearten:
  - trace
  - debug
  - info
  - warn
  - error

```
14  @RestController
15  public class DepartmentController {
16      private final Logger LOGGER = LoggerFactory.getLogger(DepartmentController.class);
17
18      @Autowired
19      private DepartmentService departmentService;
20
21      @GetMapping("/department")
22      public List<Department> getAllDepartments() {
23          LOGGER.info("Getting All Departments");
24          return departmentService.getAllDepartments();
25      }
}
```

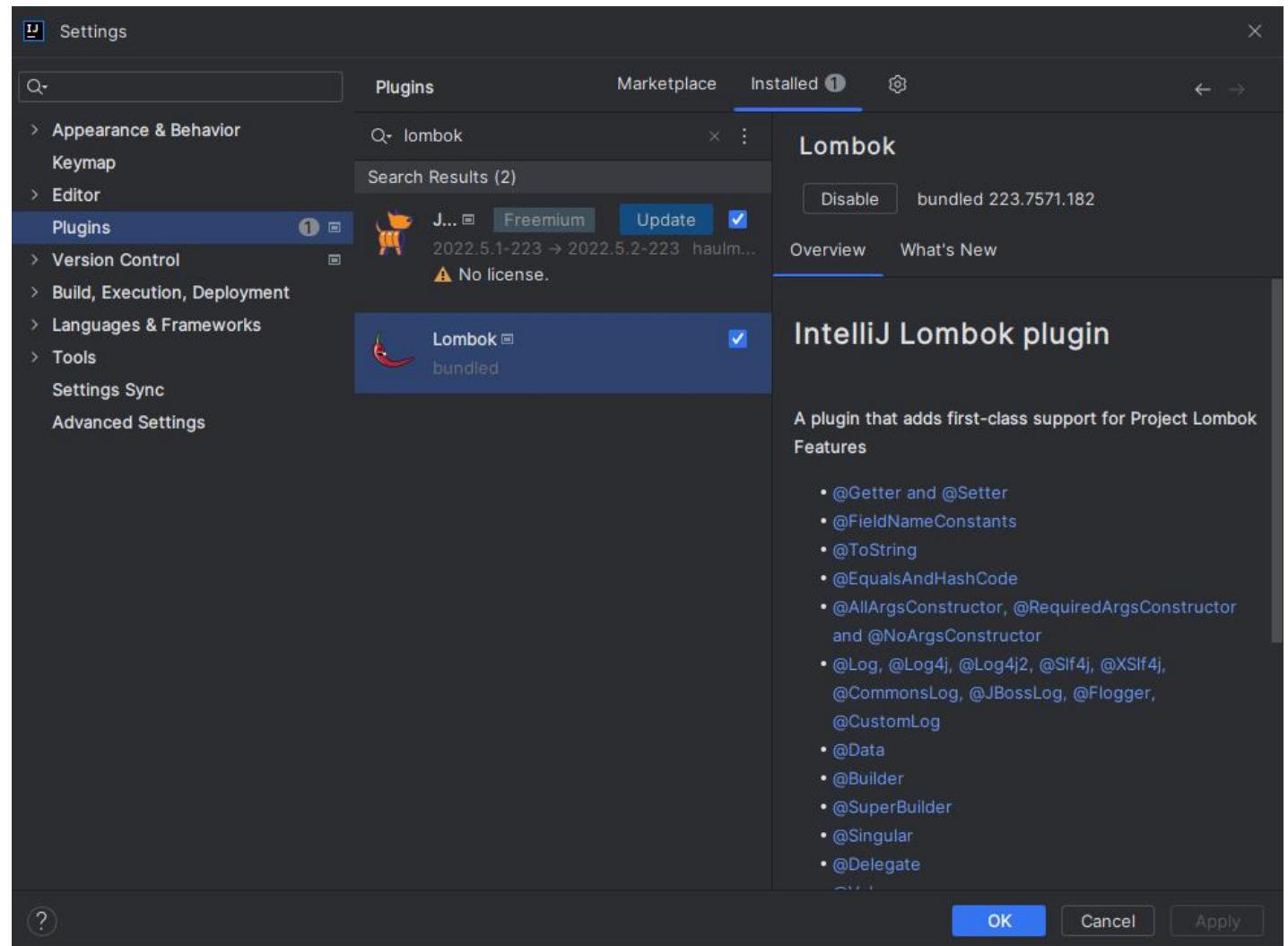
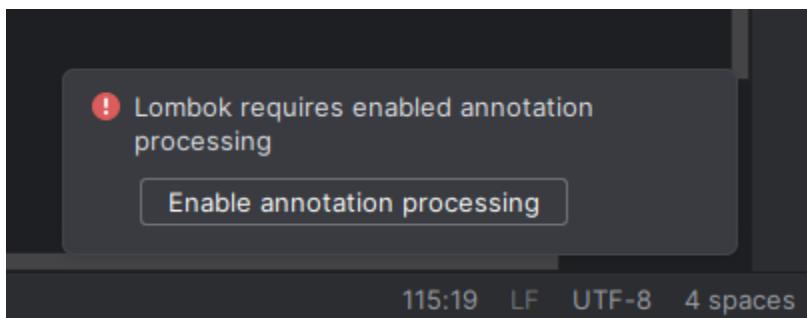
# Projekt Lombok

- Bibliothek, die den Javacode reduziert.
- Entfernt Boilerplate Code und ersetzt diese durch einige Annotationen.
- Der Code wird dann erst zur Compilezeit erstellt

```
59 ①      <dependency>
60          <groupId>org.projectlombok</groupId>
61          <artifactId>lombok</artifactId>
62          <optional>true</optional>
63      </dependency>
64  </dependencies>
65
66  <build>
67      <plugins>
68          <plugin>
69              <groupId>org.springframework.boot</groupId>
70              <artifactId>spring-boot-maven-plugin</artifactId>
71              <configuration>
72                  <excludes>
73                      <exclude>
74                          <groupId>org.projectlombok</groupId>
75                          <artifactId>lombok</artifactId>
76                      </exclude>
77                  </excludes>
78              </configuration>
79          </plugin>
80      </plugins>
81  </build>
```

# Projekt Lombok Plugin

- Plugin muss aktiviert werden, damit die IDE weiß, dass Getter/Setter usw. über Lombok bereitgestellt werden.



# Equals und Hashcode von DTOs

- Wir können Lombok dazu nutzen, um uns die hashCode() und equals() Methoden erzeugen zu lassen.
- Hier ist es wichtig, ausschließlich den Primary Key zu verwenden, da es sonst zu erheblichen Laufzeiteinbußen kommen kann.

```
15 @Entity
16 @EqualsAndHashCode(onlyExplicitlyIncluded = true)
17 public class Card {
18     @Id
19     @EqualsAndHashCode.Include
20     private Long id;
```

# Lombok in Aktion

```
12  @Entity
13  @Data
14  @NoArgsConstructor
15  @AllArgsConstructor
16  @Builder
17  public class Department {
18
19    @Id
20    @GeneratedValue(strategy = GenerationType.AUTO)
21    private Long id;
22
23    @Column(name = "dep_name", nullable = false)
24    @NotBlank(message = "Please add a Department Name.")
25    private String name;
26    private String address;
27    private String code;
28
29 }
```

Lombok Annotationen

Es gibt auch  
inzwischen Records  
in Java

```
Department department0 = Department.builder().name("Vertrieb").address("Haus 11").code("123").build();
```

# @ResponseStatus

- Die Annotation @ResponseStatus erlaubt es einfach den http-Status der Response zu setzen, ohne eine neue ResponseEntity zu instanziieren und zu befüllen.
- HttpStatus:
  - 200: OK
  - 201: OK und erfolgreiches speichern bei Post/Put
  - 400: Fail, da Anfrage nicht für den Server verständlich ist
  - 404: Fail. Obwohl Anfrage korrekt ist, kann der Server sie nicht beantworten.

# Exception Handling mit @RestControllerAdvice

- AOP-basierend.
- Über @RestControllerAdvice kann eingestellt werden, für welche Packages der Advice gilt.
- Gut: Alles ist zentral geregelt.

```
12  @RestControllerAdvice
13  public class ValidationExceptionHandler{
14
15      @ResponseStatus(HttpStatus.BAD_REQUEST)
16      @ExceptionHandler(MethodArgumentNotValidException.class)
17  @
18      public List<ObjectError> handleExceptions(
19          MethodArgumentNotValidException ex
20      ) {
21          return ex.getBindingResult().getAllErrors();
22      }
23 }
```

# ResponseStatusException

- ResponseStatusException ist ein schneller Weg lokal eine Exception Handling zu implementieren.

```
40     @DeleteMapping("/department/{id}")
41     public void deleteDepartmentById(@PathVariable("id") Long id) {
42         try {
43             departmentService.deleteDepartmentById(id);
44         } catch (DepartmentNotFoundException e) {
45             throw new ResponseStatusException(HttpStatus.NOT_FOUND, e.getMessage(), e);
46         }
47     }
```

# H2 → MySQL

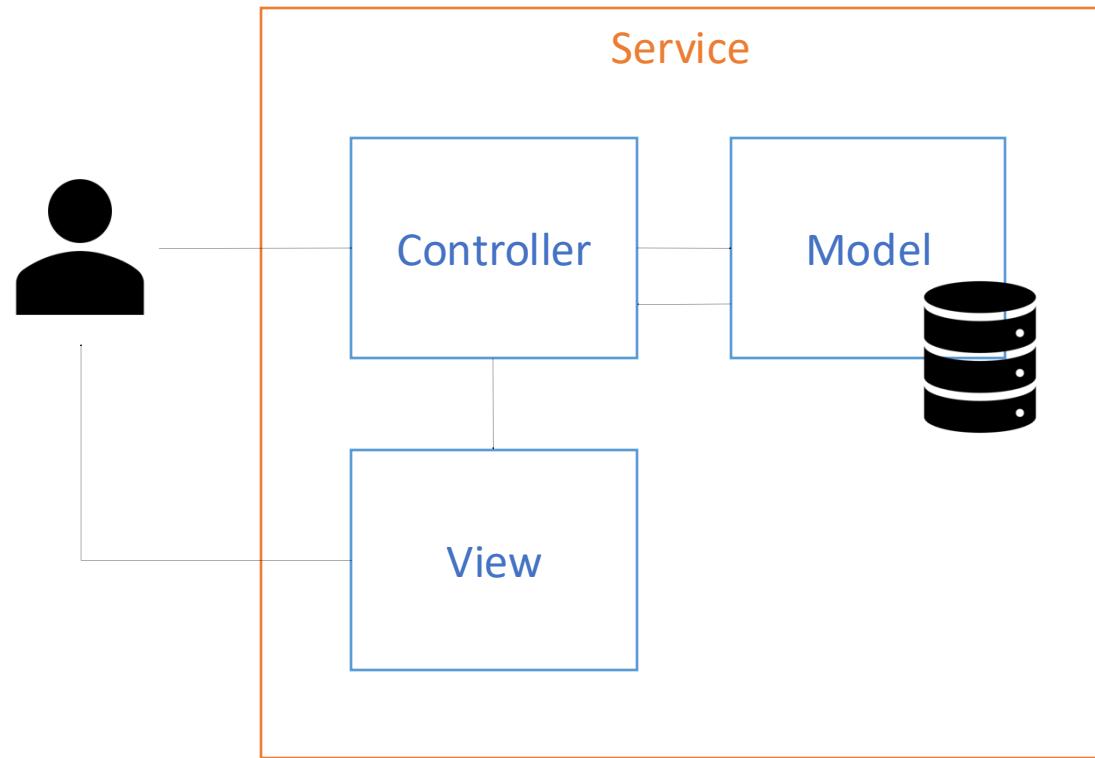
- Download MySQL: <https://www.mysql.com/downloads/>
- Download MySQL Workbench:  
<https://www.mysql.com/products/workbench/>
- DB anlegen.
- Dependency einfügen.
- application.properties anpassen.

```
67 ⑩ |     <dependency>
68 |         <groupId>com.mysql</groupId>
69 |         <artifactId>mysql-connector-j</artifactId>
70 |         <scope>runtime</scope>
71 |     </dependency>
```

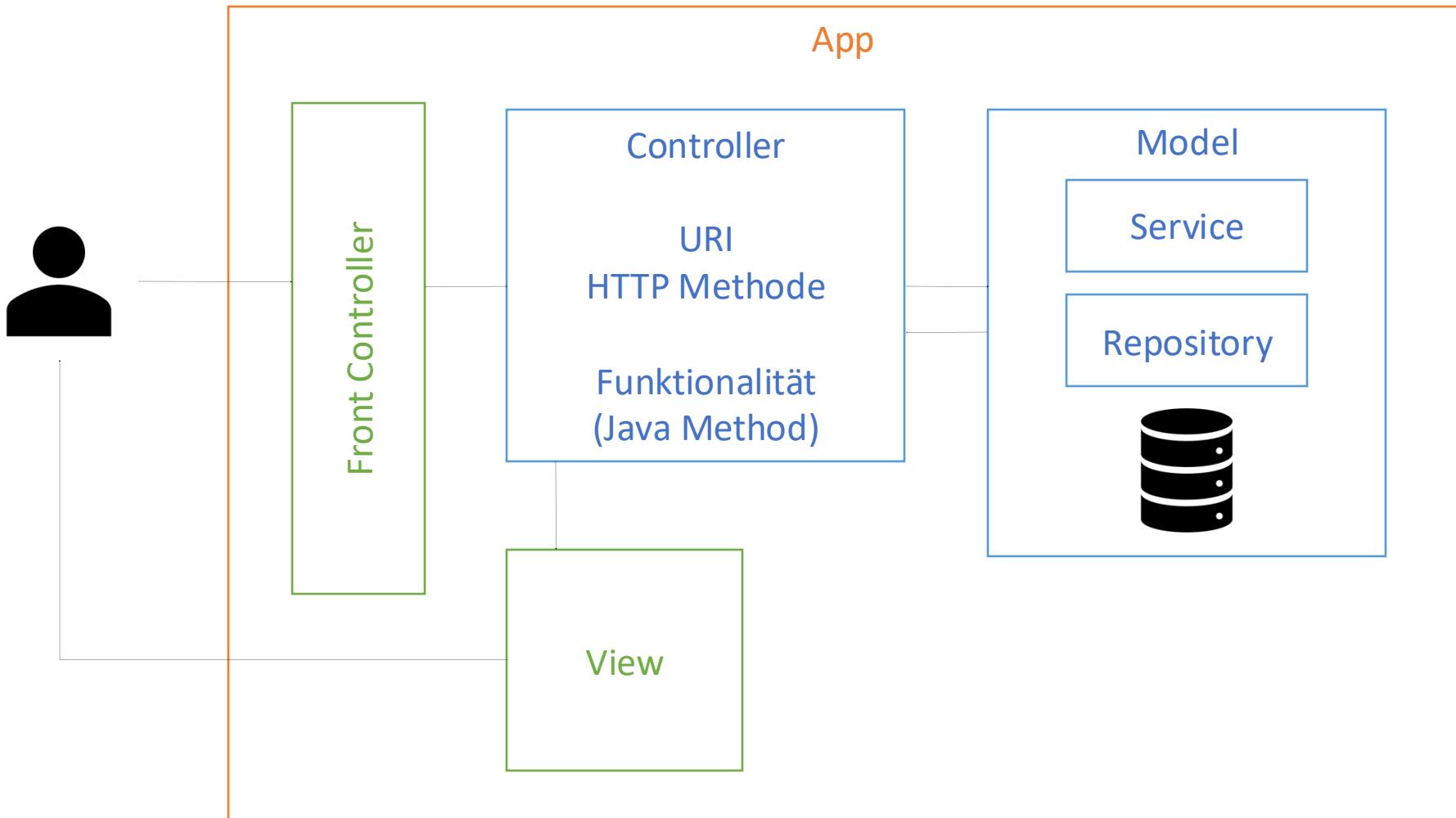
```
14 | spring.datasource.url=jdbc:mysql://localhost:3306/exampledb
15 | spring.datasource.username=root
16 | spring.datasource.password=password
17 | spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

# MVC

- Controller nimmt Anfragen entgegen.
- Controller gibt die Verarbeitung an das Model weiter.
- Model gibt die Ergebnisse an den Controller zurück.
- Der Controller gibt die Ergebnisse an View weiter.
- View übersetzt die Ergebnisse in eine für den Nutzer nützliche Form.



# Spring MVC



# Spring MVC

- Wir müssen nur Teil des Control Layers und des Model Layer implementieren.
- Der Front Controller hat die Funktion, die Anfrage von deren Darstellung unabhängig zu machen (z.B. xml oder json).

# Testing

- Junit und Mockito sind automatisch eingebunden. (spring-boot-starter-test)

# Mocking in Unitests

- In jedem Unitests werden spezifische Komponenten getestet.
- Diese Komponenten hängen von anderen Komponenten ab, deren Funktionalität nicht getestet, sondern vorausgesetzt wird.
- Diese abhängigen Komponenten werden ggf. gar nicht wirklich erstellt, sondern „Mocks“ werden an ihrer Stelle verwendet.
- Dieses Vorgehen nennt sich Mocking.

# @SpringBootTest

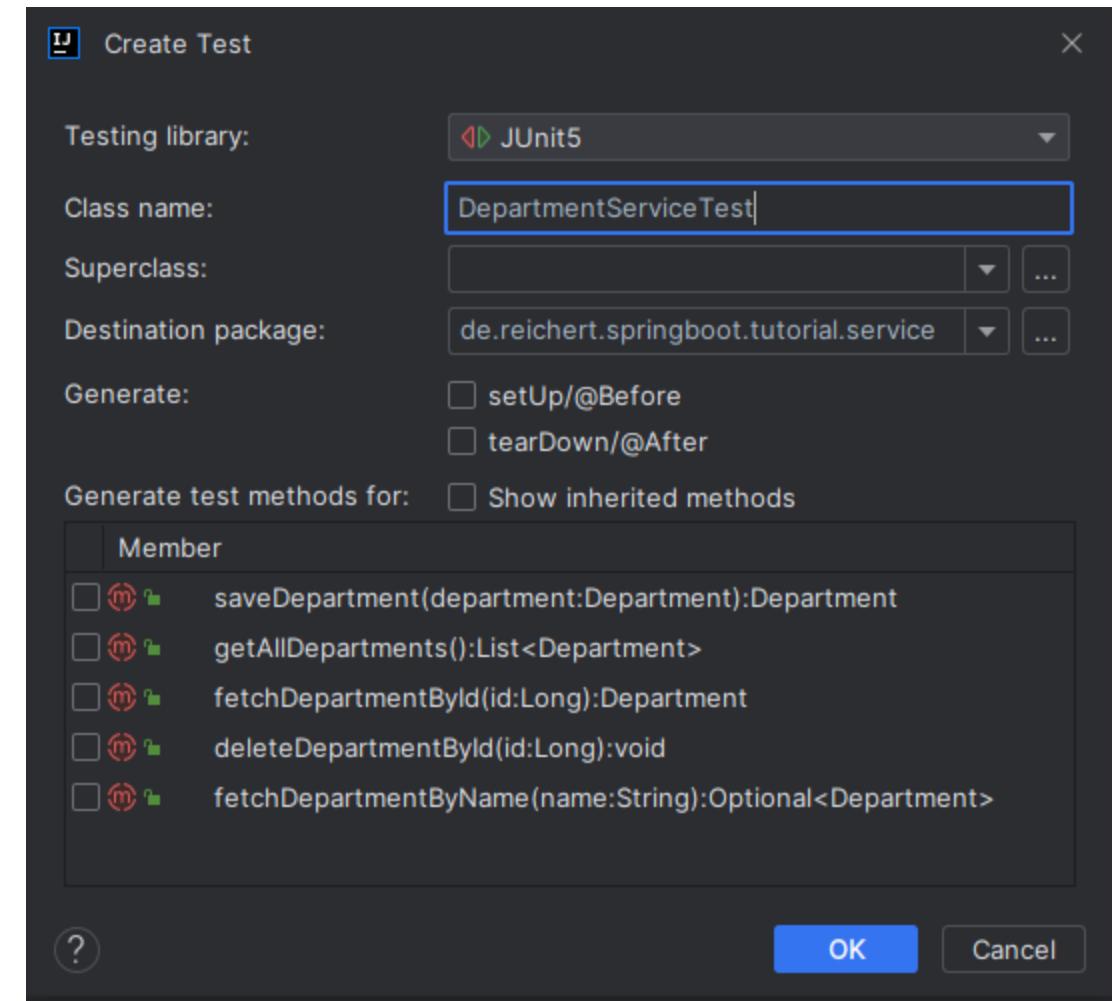
- `@SpringBootTest` zeigt Junit an, dass es sich um Spring Boot Tests handelt.
- Der Test `contextLoads()` ist zwar leer, dennoch wird er nur erfolgreich sein, wenn ein Laden des Spring Application Contexts nicht möglich ist.

```
6  @SpringBootTest
7 ► class SpringWebStarterApplicationTests {
8
9
10 ► void contextLoads() {
11
12
13 }
```

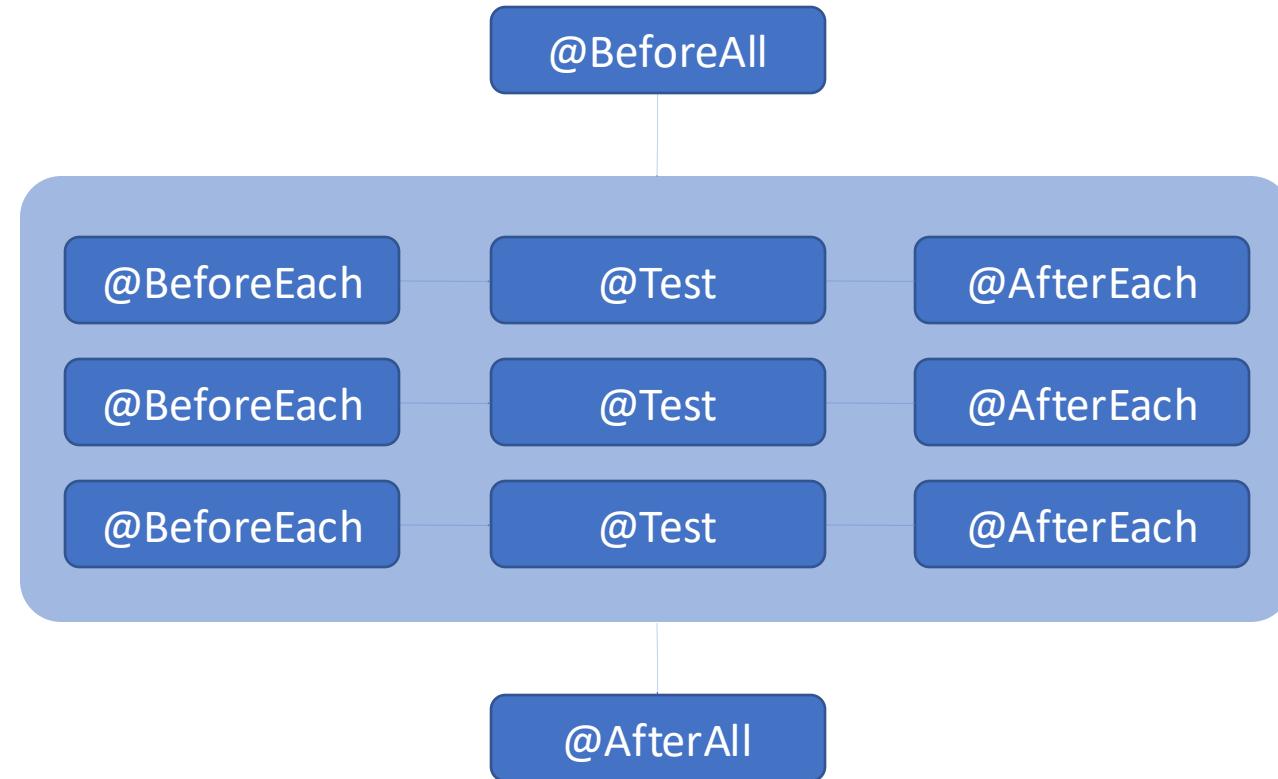
# Testklassen generieren

- IntelliJ erstellt uns auf Wunsch Testklassen.
- Wir müssen jedoch noch die `@SpringBootTest` Annotation setzen.

```
9  public interface DepartmentService { Alt+Einfügen
10     Department saveDepartment(Department department);
11
12     List<Department> getAllDepartments();
13
14     Department fetchDepartmentById(Long id) throws DepartmentNotFoundException;
```



# Testannotationen



Tests, die mit @Disabled gekennzeichnet sind, werden nicht ausgeführt.

# Service Layer testen

```
15 @SpringBootTest
16 class DepartmentServiceTest {
17
18     @Autowired
19     private DepartmentService departmentService;
20
21     @MockBean
22     private DepartmentRepository departmentRepository;
23
24     @BeforeEach
25     void setUp() {
26         Department department = Department.builder().name("IT")
27             .address("Haus 11").id(1L).code("TT-000").build();
28
29         Mockito.when(departmentRepository.findByName("IT"))
30             .thenReturn(Optional.of(department));
31     }
32
33     @Test
34     @DisplayName("Get Data based on valid Department Name")
35     void fetchDepartmentByName_valid() {
36         String departmentName = "IT";
37         Optional<Department> found = departmentService.fetchDepartmentByName(departmentName);
38         assertEquals(departmentName, found.get().getName());
39     }
40 }
```

Mocken vom Repository  
vor jedem Test

# Übung

- Schreibe selber Tests.
- Schreibe dabei auch einen Test, der eine Exception erwartet.

# Repository Layer Testen

```
12  @DataJpaTest
13  class DepartmentRepositoryTest {      Nutze spezielle Konfiguration (Link)
14
15    @Autowired
16    private DepartmentRepository departmentRepository;
17
18    @Autowired
19    private TestEntityManager entityManager;
20
21    @BeforeEach
22    void setUp() {
23        Department department = Department.builder().name("IT")
24            .address("Haus 11").code("TT-000").build();
25
26        entityManager.persist(department);      Objekt ist nur für die Dauer
27    }                                         des Tests persistent.
28
29    @Test
30    public void whenFindById_thenReturnDepartment() {
31        Department department = departmentRepository.findById(1L).get();      Test
32        assertEquals(department.getName(), "IT");
33    }
}
```

# Controller testen

```
19  @WebMvcTest(DepartmentController.class)
20  class DepartmentControllerTest {
21
22      @Autowired
23      private MockMvc mockMvc; Mock MVC
24
25      @MockBean
26      private DepartmentService departmentService;
27
28      private Department department;
29
30      @BeforeEach
31      void setUp() {
32          department = Department.builder().name("IT")
33              .address("Haus 11").id(1L).code("TT-000").build();
34      }
35
36      @Test
37      void saveDepartment() throws Exception {
38          // No id set.
39          Department inputDepartment = Department.builder().name("IT")
40              .address("Haus 11").code("TT-000").build();
41
42          Mockito.when(departmentService.saveDepartment(inputDepartment))
43              .thenReturn(department); Mock Service Layer
Request absenden
44
45          MockHttpServletRequestBuilder post = MockMvcRequestBuilders.post(urlTemplate: "/department")
46              .contentType(MediaType.APPLICATION_JSON)
47              .content("{\"name\":\"IT\", \"address\":\"Haus 11\", \"code\": \"TT-000\"}");
48
49          mockMvc.perform(post).andExpect(MockMvcResultMatchers.status().isOk());
50      }
51  }
```

# Spring Boot Actuator

- Das Actuator Modul von Spring Boot stellt HTTP-Endpunkte bereit, die Auskunft über den Zustand der Anwendung geben.
- <http://localhost:8080/actuator/>
- [Liste aller Endpunkte](#)
  - beans
  - metrics
  - health
  - ...

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```



enthält

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-actuator-autoconfigure</artifactId>
  <version>3.0.1</version>
  <scope>compile</scope>
</dependency>
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-observation</artifactId>
  <version>1.10.2</version>
  <scope>compile</scope>
</dependency>
```

# Java Management Extensions (JMX)

- JMX erlaubt die Administration und das Monitoring von JVMs.
- JMX verwaltet Klassen und deren Methoden als Mbeans (Managed Beans).
- Es stellt Annotation für eine einfache Nutzung von JMX bereit.
- Die Java-Monitoring- und Managementkonsole lässt sich über den Befehl „jconsole“ starten.

# Beispiel

- Über die Annotationen wird angezeigt, dass es sich um Klassen/Methoden handelt, die von JMX überwacht werden
  - @ManagedResource
  - @ManagedAttribute
  - @ManagedOperation
  - @ManagedParameter

```
13 ⑩ @RestController
14 ⑪ @ManagedResource(objectName = "MyMBeans:category=MBean,objectName=testBean")
15 ⑫ public class MyController {
16
17
18 ⑬ @
19
20
21
22
23 ⑭ @ManagedAttribute
24 ⑮ public double getCounter() {
25    return counter.count();
26
27
28
29 ⑯ @
30
31
32
33
34 ⑰ @GetMapping(@RequestMapping("/greeting"))
35 ⑱ @ManagedOperation
36 ⑲ public String greeting() {
37    counter.increment();
38    return "Hallo";
39
40 ⑳ @
41
42
43
44 } ⑴
```

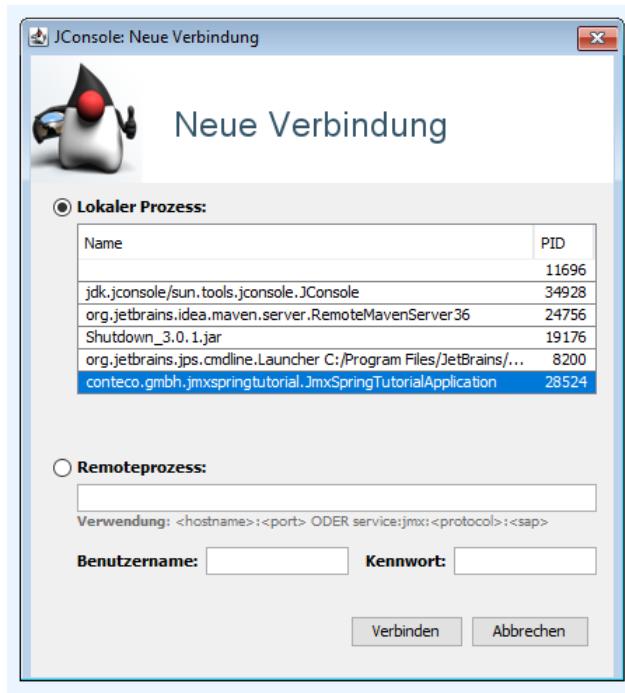
# Konfiguration von jmx

- Wir geben hier alle Endpunkte frei (was in der Produktion nicht geschehen würde).

```
application.properties
1 #jmx
2 management.endpoints.jmx.domain=superapp
3 management.jmx.metrics.export.enabled=true
4 management.endpoints.jmx.exposure.include=*
5
6 #actuator
7 management.endpoints.web.exposure.include=*
```

# jconsole

- Über die jconsole kann der Code ausgeführt und überwacht werden.



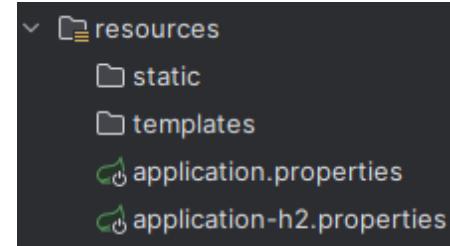
PS C:\Users\Vikto\IdeaProjects\jmx-spring-tutorial> **jconsole**

The screenshot shows the Java Monitoring and Management Console (JConsole) interface. At the top, it displays the command PS C:\Users\Vikto\IdeaProjects\jmx-spring-tutorial> followed by the application name **jconsole**. The main window title is 'Java-Monitoring- und Managementkonsole - pid: 28524 conteco.gmbh.jmxspringtutorial.JmxSpringTutorialApplication'. The interface has several tabs: Überblick, Arbeitsspeicher, Threads, Klassen, VM-Übersicht, and MBeans. The 'MBeans' tab is currently selected. On the left, a tree view shows the structure of MBeans, including 'JMImplementation', 'MyMBeans', and 'testBean'. Under 'testBean', there are nodes for 'Attribute', 'Vorgänge', and 'Benachrichtigung'. On the right, a table titled 'Attributwerte' (Attribute Values) lists a single entry: 'Counter' with a value of '0.0'. At the bottom right of the main window, there is a button labeled 'Aktualisieren' (Update).

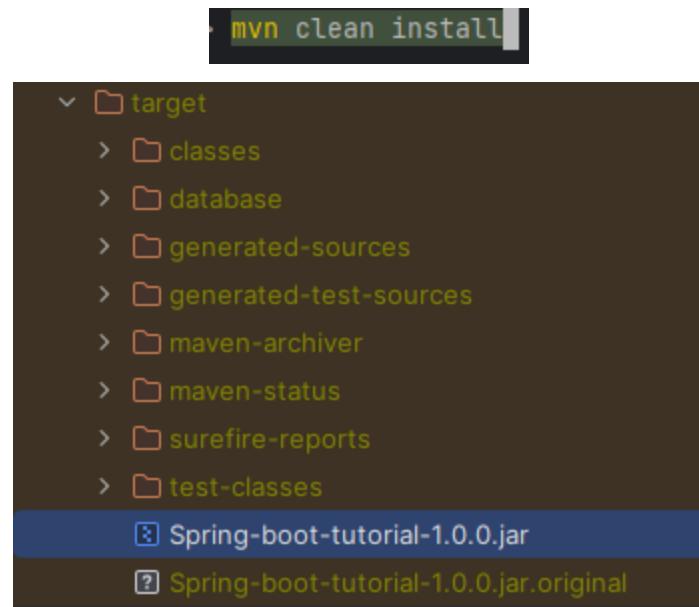
# @Profile

- Mit @Profile kann definiert werden, bei welchen Profilen die Bean nutzbar ist.
- Die Profile können als Array angegeben werden oder mit „!“ (nicht), „||“ (oder) und „&&“ (und) verknüpft werden.
- Profil aktiveren (eine von vielen Möglichkeiten): füge in application.properties hinzu:  
`spring.profiles.active=dev,test`
- „default“ ist das Profil, dass aktiv ist, wenn kein anderes Profil aktiv ist
- Kann an @Component und an @Bean gesetzt werden.

# Einstellungen in Profildatei



- Mit application-{profile}.properties, können Properties gesetzt werden, die für ausgewählte Profile gelten.
- Jar mit Profil starten:



```
\target> java -jar .\Spring-boot-tutorial-1.0.0.jar --spring.profiles.active=h2
```

# J2EE, JEE, Jakarta EE

- Bezeichnung für Plattform zur Entwicklung von Enterprise-Anwendungen.
- Diese umfasst viele verschiedene APIs.
- Namenswechsel im Laufe der Zeit:
  - **J2EE** (Java 2 Enterprise Edition)
  - **JEE** (Java Enterprise Edition) wurde vom Java Community Process (JCP) definiert.
  - **Jakarta EE**. Verantwortung für JEE ging von Oracle an die Eclipse Foundation über. Aus rechtlichen Gründen, musste der Name „Java“ in „Jakarta“ geändert werden.

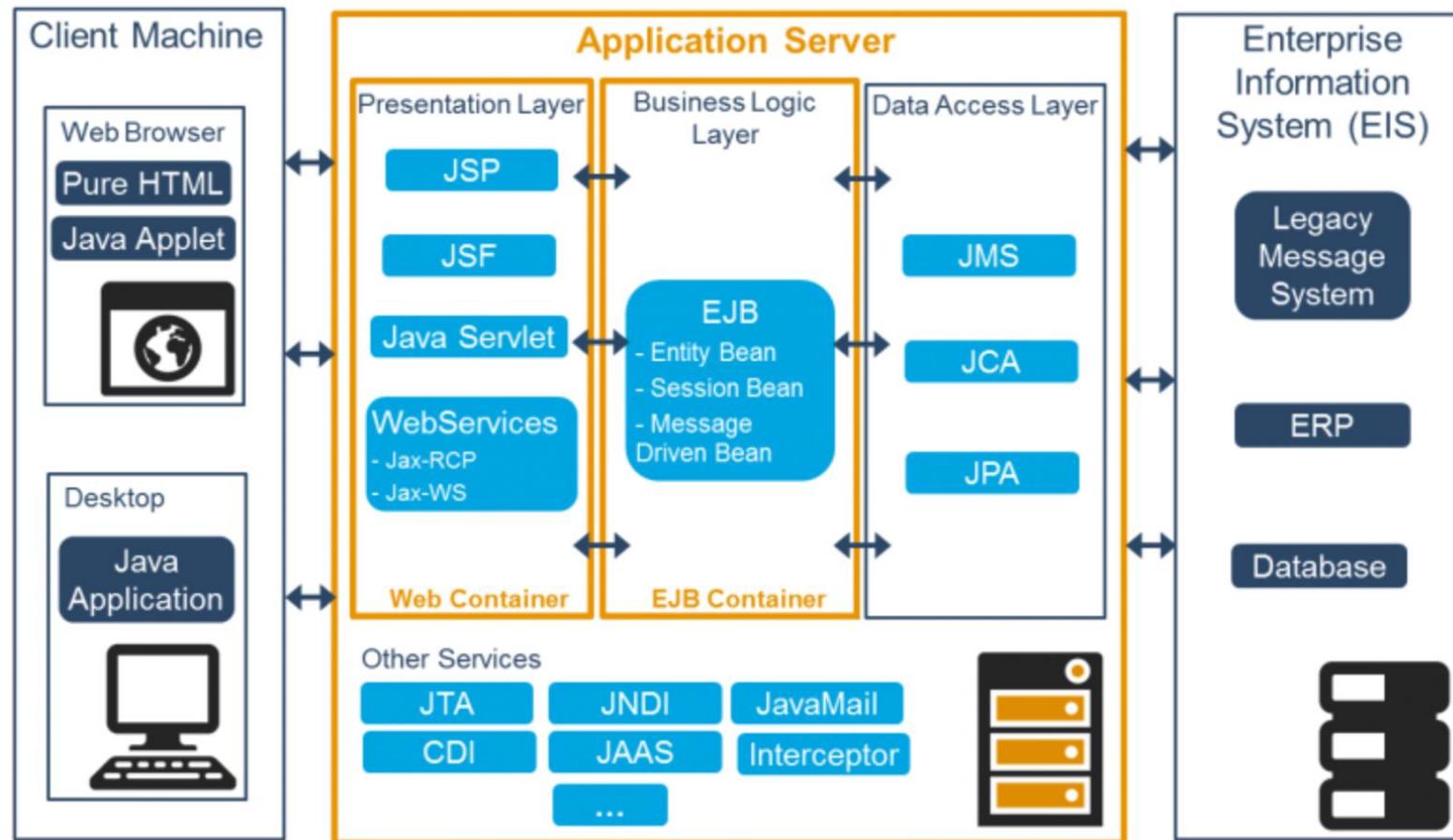
Version	Ausführlicher Name	Veröffentlichungsdatum der Final Release
1.0	Java 2 Platform Enterprise Edition, v 1.0	Dezember 1999
1.2	Java 2 Platform Enterprise Edition, v 1.2	2000
1.2.1	Java 2 Platform Enterprise Edition, v 1.2.1	23. Mai 2000
1.3	Java 2 Platform Enterprise Edition, v 1.3	24. September 2001
1.4	Java 2 Platform Enterprise Edition, v 1.4	24. November 2003
5	Java Platform, Enterprise Edition, v 5	11. Mai 2006
6	Java Platform, Enterprise Edition, v 6	10. Dezember 2009
7	Java Platform, Enterprise Edition, v 7	12. Mai 2013
8	Java Platform, Enterprise Edition, v 8	18. September 2017

Danach folgten die Versionen:

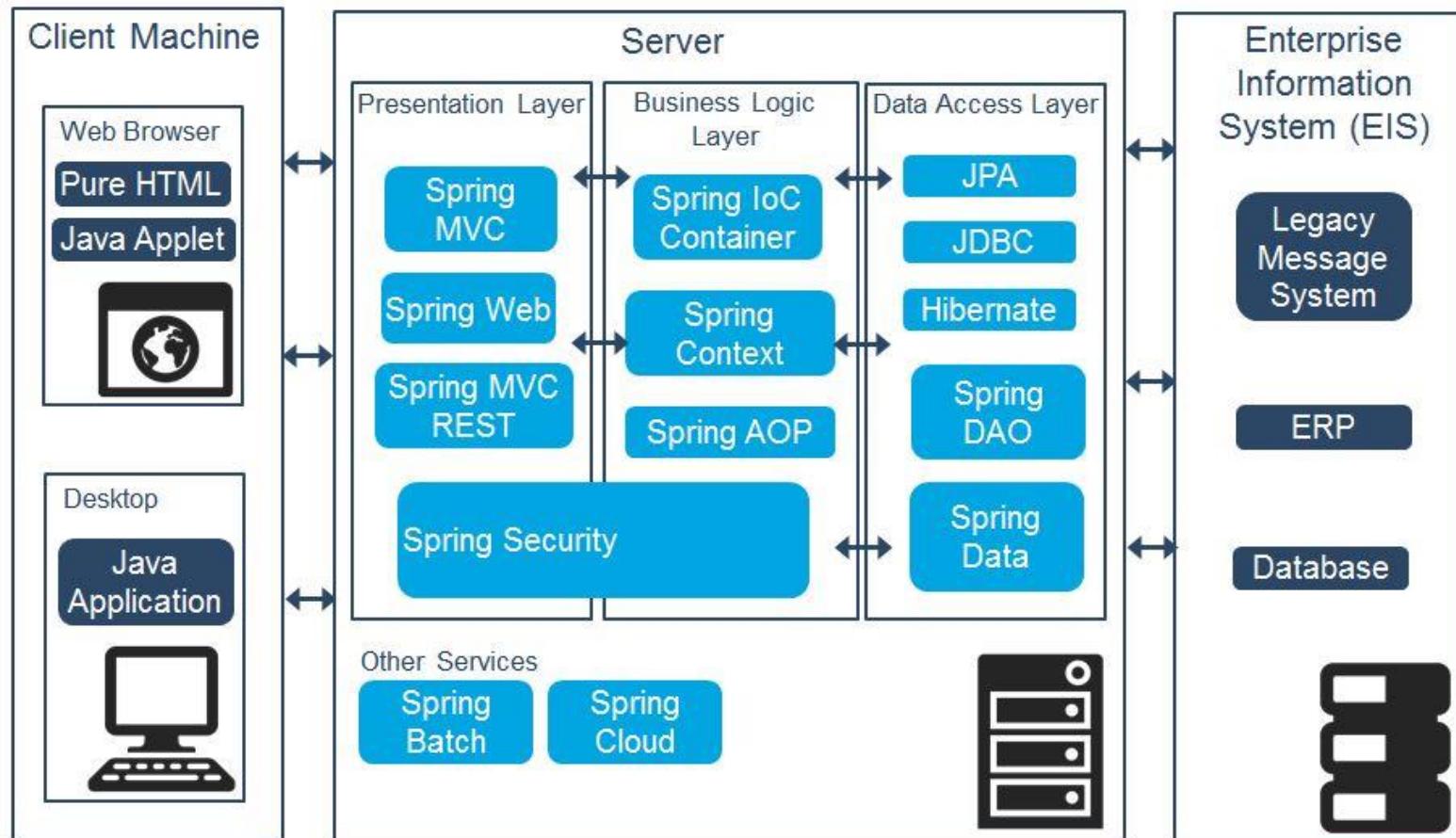
- Jakarta EE 8 (2019), vollkompatible Version zu Java Platform, Enterprise Edition, v 8
- Jakarta EE 9 (2020)
- Jakarta EE 9.1 (2021)
- Jakarta EE 10 (2022)

Jakarta EE 11 (Planned for July 2024) + Support JDK 21

# JEE/Jakarta EE - Komponenten



# Spring Komponenten



# Spring JPA

- Java basiert auf Klassen, Datenbanken auf Tabellen.
- Wir lassen Spring die Übersetzung dieser beiden Datenstrukturen übernehmen.
- Nutzt Hibernate als Vendor. Andere können konfiguriert werden (eclipse-link, OpenJPA, DataNucleus,...)
- JPA (Java Persistence API) ist eine Spezifikation für das Persistieren von Java-Objekten in relationale Datenbanken.

# Pures JDBC

- Ohne ein Framework ist die Arbeit mit JDBC aufwendig und fehleranfällig.
- Viel Boilerplate Code (Connections öffnen und schließen,...)

```
39     String DB_URL = "jdbc:h2:file:./database/h2db;AUTO_SERVER=true";
40     String USER = "sa";
41     String PASS = "";
42     String QUERY = "SELECT id, dep_name, address, code FROM dep";
43
44     try(Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);
45         Statement stmt = conn.createStatement();
46         ResultSet rs = stmt.executeQuery(QUERY)) {
47
48         // Extract data from result set
49         while (rs.next()) {
50             // Retrieve by column name
51             System.out.print("ID: " + rs.getInt(columnLabel: "id"));
52             System.out.print(", NAME: " + rs.getString(columnLabel: "dep_name"));
53             System.out.print(", ADDRESS: " + rs.getString(columnLabel: "address"));
54             System.out.println(", CODE: " + rs.getString(columnLabel: "code"));
55         }
56     } catch (SQLException e) {
57         e.printStackTrace();
58     }
59 }
```

# Wer macht was?

Action	Spring	You
Define connection parameters.		X
Open the connection.	X	
Specify the SQL statement.		X
Declare parameters and provide parameter values		X
Prepare and run the statement.	X	
Set up the loop to iterate through the results (if any).	X	
Do the work for each iteration.		X
Process any exception.	X	
Handle transactions.	X	
Close the connection, the statement, and the resultset.	X	

# Neues Projekt erstellen

- Erstelle ein neues Projekt mit [start.spring.io](https://start.spring.io) den Abhängigkeiten:
  - Spring Data JPA
  - Lombok
  - H2
  - Spring Web

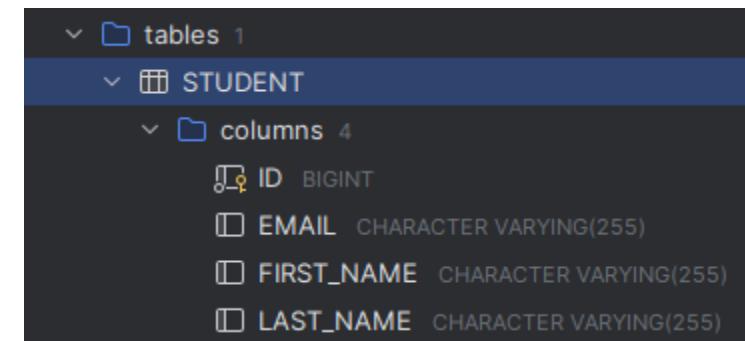
# Properties für DB

```
application.properties ×  
1 spring.jpa.database-platform=org.hibernate.dialect.H2Dialect  
2 spring.datasource.url=jdbc:h2:file:./database/h2db;AUTO_SERVER=true  
3 spring.datasource.username=sa  
4 spring.datasource.password=  
5 spring.datasource.driver-class-name=org.h2.Driver  
6 spring.h2.console.enabled=true  
7 spring.jpa.hibernate.ddl-auto=create-drop  
8 spring.jpa.show-sql=true  
9 spring.jpa.properties.hibernate.format_sql=true
```

# Eine erste Klasse

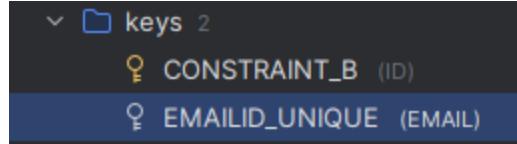
Annotation	Bedeutung
@Entity	Zu dieser Klasse gehört eine Tabelle
@Table	Genauere Definitionen über die Tabelle.
@Data @AllArgsConstructor @NoArgsConstructor @Builder	Lombok Annotationen für kompaktere Java Klassen.
@Id	Dieses Feld ist der Primary Key.
@GeneratedValue	Der Inhalt dieses Feldes wird automatisch generiert.
@Column	Genauere Informationen zur Spalte.

```
9  @Entity
10 @Table(name = "student")
11 @Data
12 @AllArgsConstructor
13 @NoArgsConstructor
14 @Builder
15 public class Student {
16   @Id
17   @GeneratedValue(strategy = GenerationType.AUTO)
18   @Column(name = "id", nullable = false)
19   private Long id;
20
21   private String firstName;
22   private String lastName;
23   private String email;
24 }
```



# @UniqueConstraints

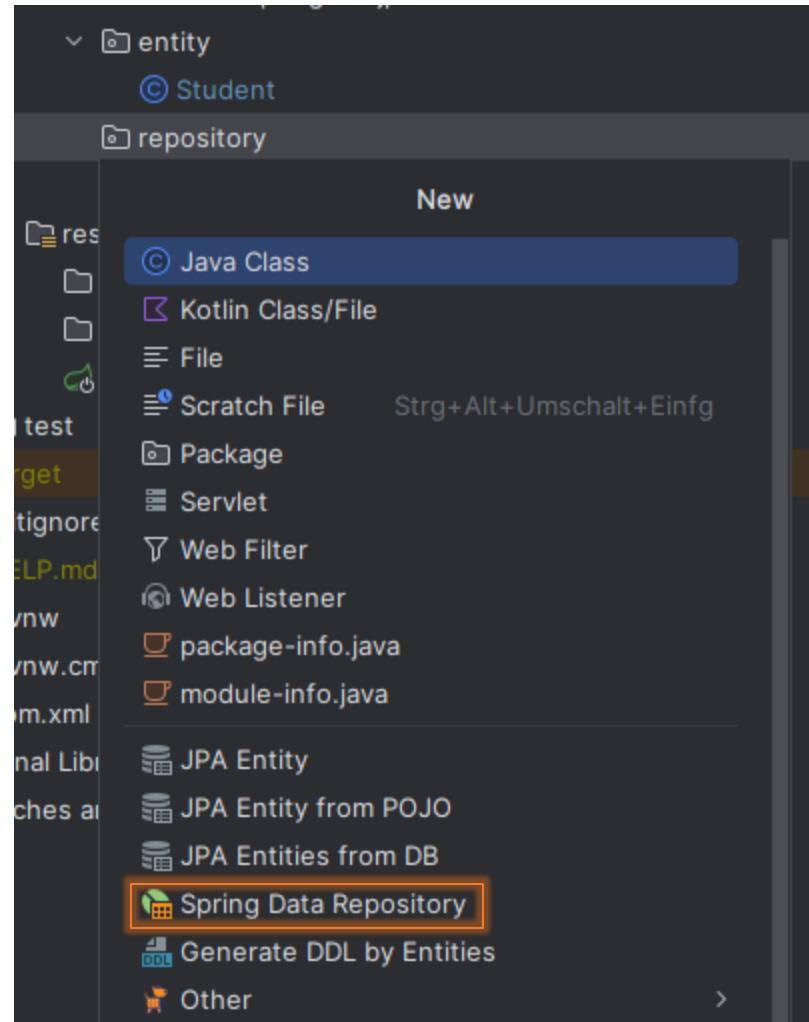
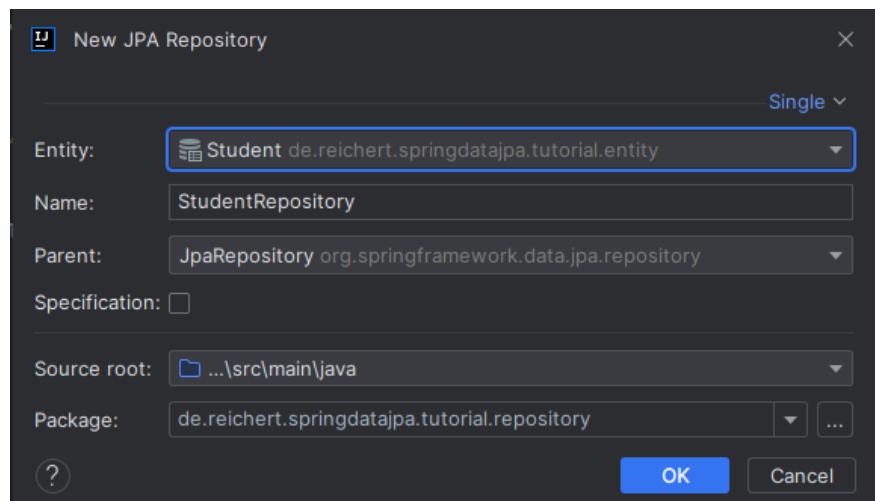
- Mit `@UniqueConstraints` können wir eben solche einfügen.



```
10  @Data
11  @AllArgsConstructor
12  @NoArgsConstructor
13  @Builder
14  @Entity
15  @Table(
16      name = "student",
17      uniqueConstraints = @UniqueConstraint(
18          name = "emailid_unique",
19          columnNames = "email")
20  )
21  public class Student {
22      @Id
23      @GeneratedValue(strategy = GenerationType.AUTO)
24      @Column(name = "id", nullable = false)
25      private Long id;
26
27      @Column(name = "first_name", nullable = false)
28      private String firstName;
29
30      @Column(name = "last_name", nullable = false)
31      private String lastName;
32
33      @Column(name = "email", nullable = false)
34      private String email;
35  }
```

# Repository für Student

- Wir erstellen wieder ein Repository für die Student Entity.



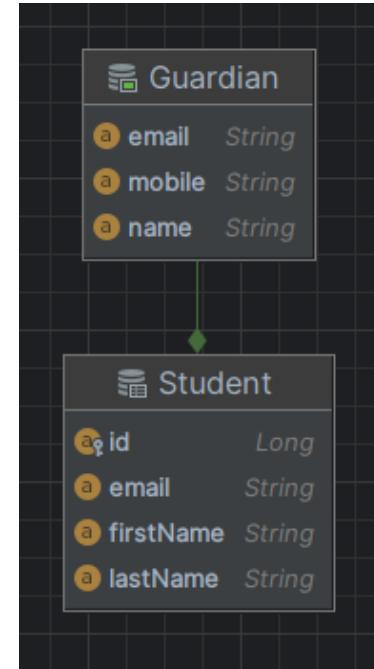
```
6  public interface StudentRepository extends JpaRepository<Student, Long> {  
7 }
```

# Daten Speichern

```
10 ④ @SpringBootApplication
11 ► public class SpringDataJpaTutorialApplication {
12
13 ►     public static void main(String[] args) {
14         SpringApplication.run(SpringDataJpaTutorialApplication.class, args);
15     }
16
17 ④ @Bean
18 ④ public CommandLineRunner app(StudentRepository studentRepository) {
19     return (args -> {
20         Student student = Student.builder()
21             .email("swantje@web.de")
22             .firstName("Swantje")
23             .lastName("Maja")
24             .build();
25
26         studentRepository.save(student);
27     } );
28 }
29 }
```

# @Embeddable & @Embedded

```
13  @Data
14  @AllArgsConstructor
15  @NoArgsConstructor
16  @Builder
17  @Embeddable
18  @AttributeOverrides({
19      @AttributeOverride(name = "name", column = @Column(name = "guardian_email")),
20      @AttributeOverride(name = "email", column = @Column(name = "guardian_email")),
21      @AttributeOverride(name = "mobile", column = @Column(name = "guardian_mobile"))
22  })
23  public class Guardian {
24      private String name;
25      private String email;
26      private String mobile;
27  }
```



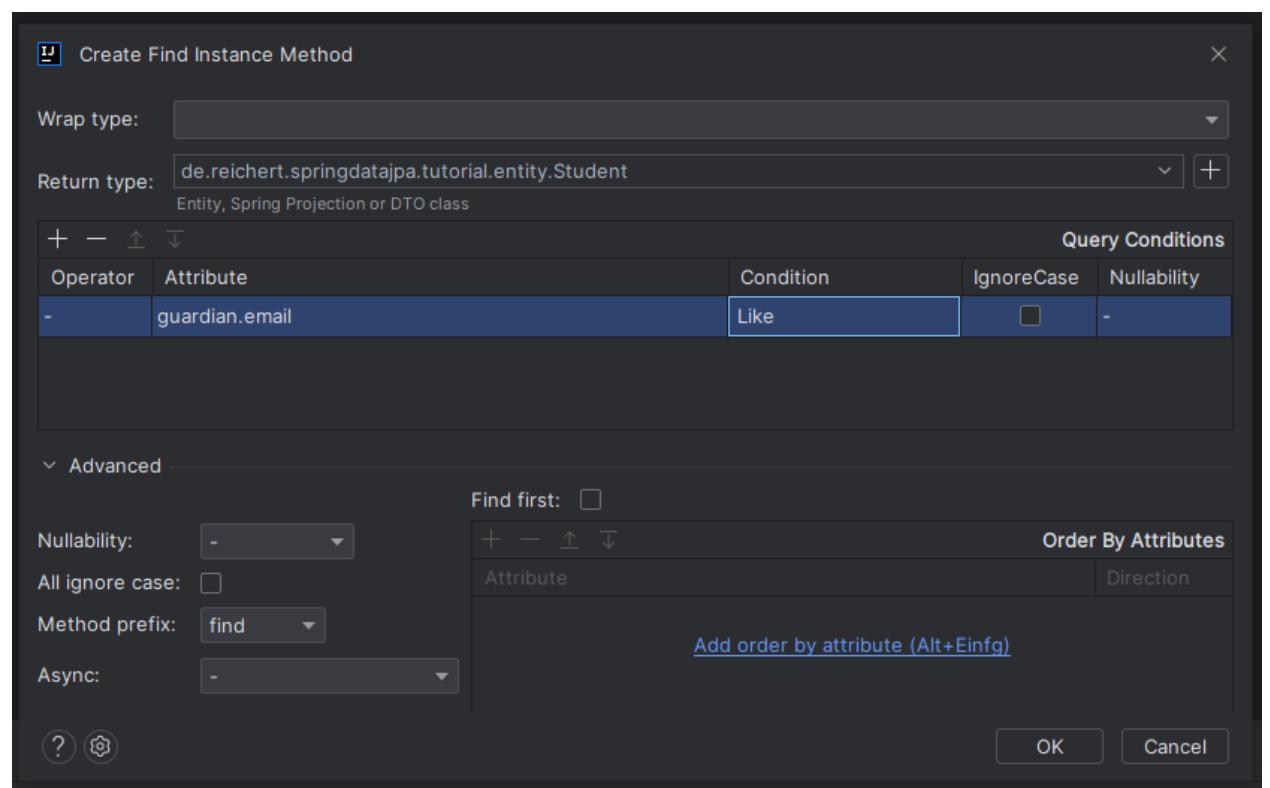
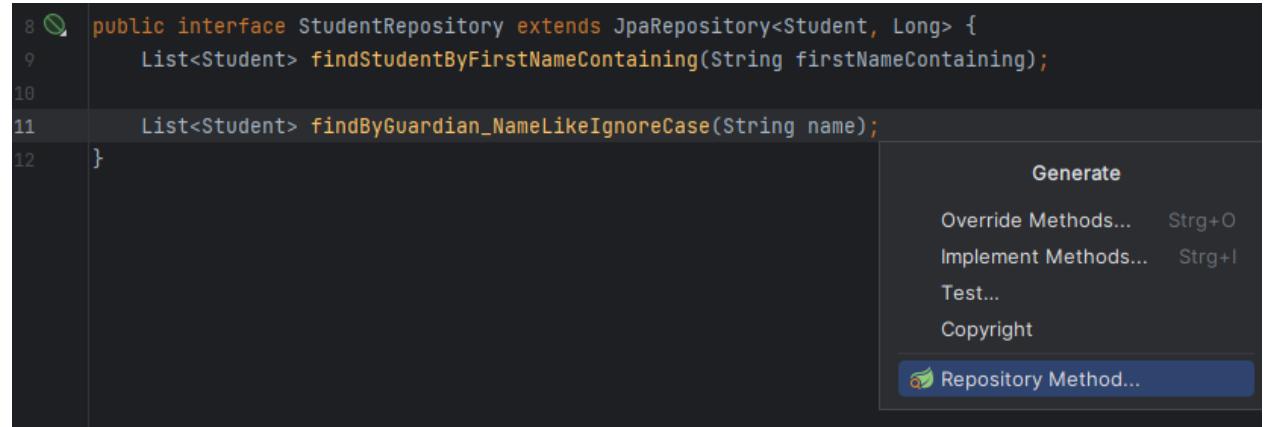
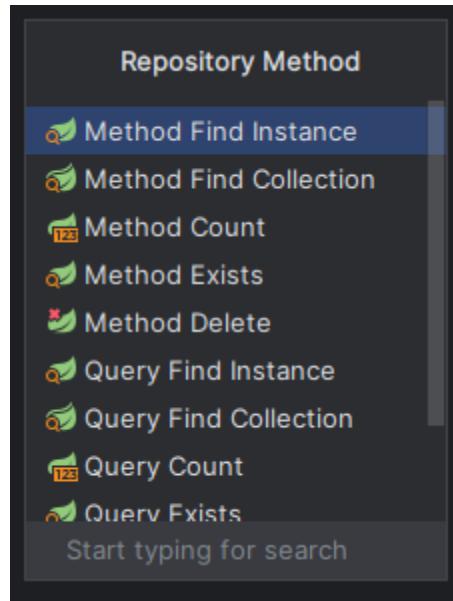
```
21  public class Student {
22      @Embedded
23      private Guardian guardian;
```

STUDENT

ID	EMAIL	FIRST_NAME	GUARDIAN_EMAIL	GUARDIAN_MOBILE	GUARDIAN_NAME	LAST_NAME
1	swantje@web.de	Swantje	karl@guardian	000-1234	Karl	Maja
2	viktor@web.de	Viktor	karl@guardian	000-1234	Karl	Reichert

# Query erstellen

- Mit Alt+Einfg öffnet sich in IntelliJ ein Kontextmenü zum Erstellen von Repository Methoden.



# JPQL Query

- JPQL basiert auf den Klassen, nicht auf den Tabellennamen!
- Hier ist ein Fehler versteckt ;)

```
15     //JPQL
16     @Query("select s from Student s where upper(s.email) like upper(?1)")
17     Optional<Student> findByEmailLikeIgnoreCase(String email);
18
19     @Query("select s from Student s where upper(s.guardian.email) like upper(?1)")
20     Optional<Student> findByGuardian_EmailLikeIgnoreCase(String email);
21
22     @Query("select s.firstName from Student s where s.id = ?1")
23     Optional<String> getNameById(Long id);
24
```

# Native Query & Parameter

- Achtung, bei Datenbankwechsel können Native Queries ggf. nicht mehr funktionieren.
- Der @Param ist ggf. optional.

```
26     // Native
27     @Query(value = "SELECT * FROM student s WHERE s.first_name = :name", nativeQuery = true)
28     List<Student> getAllByName(@Param("name") String name);
```

# Modifying Query

- `@Transactional` sorgt dafür, dass bei der Ausführung mehrerer Queries alle erfolgreich sein müssen. Ist dies nicht der Fall wird ein Rollback durchgeführt.
- `@Modifying` zeigt an, dass die Methode Änderungen an bestehenden Zeilen in der Datenbank vornimmt.
- `@Modifying` ist für `DELETE`, `UPDATE`, `INSERT` und `DDL`-Statements notwendig.

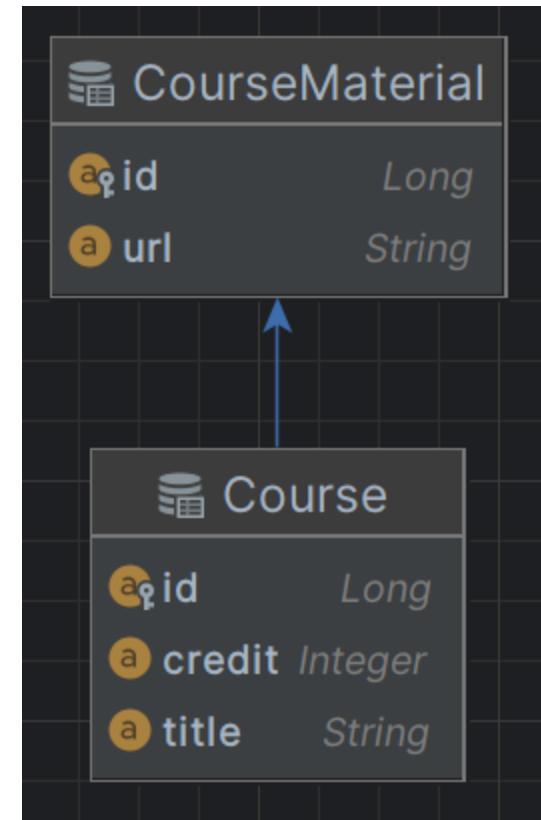
```
32      @Transactional
33      @Modifying
34      @Query("update Student s set s.email = :email where s.firstName = :firstName")
35      int updateEmailByName(@Param("email") String email, @Param("firstName") String firstName);
```

# Unidirektionale OneToOne Verknüpfung

- Nutze die @OneToOne Annotation.

```
9  @Data  
10 @AllArgsConstructor  
11 @NoArgsConstructor  
12 @Builder  
13 @Entity  
14 @Table(name = "course")  
15 public class Course {  
16     @Id  
17     @GeneratedValue(strategy = GenerationType.AUTO)  
18     @Column(name = "id", nullable = false)  
19     private Long id;  
20     private String title;  
21     private Integer credit;  
22  
23     @OneToOne  
24     private CourseMaterial courseMaterial;  
25 }
```

```
9  @Data  
10 @AllArgsConstructor  
11 @NoArgsConstructor  
12 @Builder  
13 @Entity  
14 @Table(name = "course_material")  
15 public class CourseMaterial {  
16     @Id  
17     @GeneratedValue(strategy = GenerationType.AUTO)  
18     @Column(name = "id", nullable = false)  
19     private Long id;  
20     private String url;  
21 }
```



# Unidirektionale OneToOne Verknüpfung

- Derzeit müssen wir die Objekte noch nacheinander erstellen und in der richtigen Reihenfolge speichern.

```
CourseMaterial courseMaterial = CourseMaterial.builder()
    .url("ana1.de")
    .build();

Course course = Course.builder()
    .courseMaterial(courseMaterial)
    .credit(20)
    .title("Analysis 1")
    .build();

courseMaterialRepository.save(courseMaterial);
courseRepository.save(course);
```

The screenshot shows a database interface with two tables:

- COURSE\_MATERIAL**: Contains one row with ID 1, URL "ana1.de".
- COURSE**: Contains one row with ID 1, CREDIT 20, TITLE "Analysis 1", and COURSE\_MATERIAL\_ID 1.

# CascadeType

- Damit die nicht gesicherten Objekte automatisch gespeichert/gelöscht usw. werden, kann mit „cascade“ eingestellt werden, welche Befehle weitergegeben werden.
- Es kann auch die @Cascade Annotation genutzt werden.

```
@OneToOne(cascade = CascadeType.PERSIST)
private CourseMaterial courseMaterial;
```

```
CourseMaterial courseMaterial = CourseMaterial.builder()
    .url("ana1.de")
    .build();

Course course = Course.builder()
    .courseMaterial(courseMaterial)
    .credit(20)
    .title("Analysis 1")
    .build();

// courseMaterialRepository.save(courseMaterial);
courseRepository.save(course);
```

# Bidirektionale OneToOne

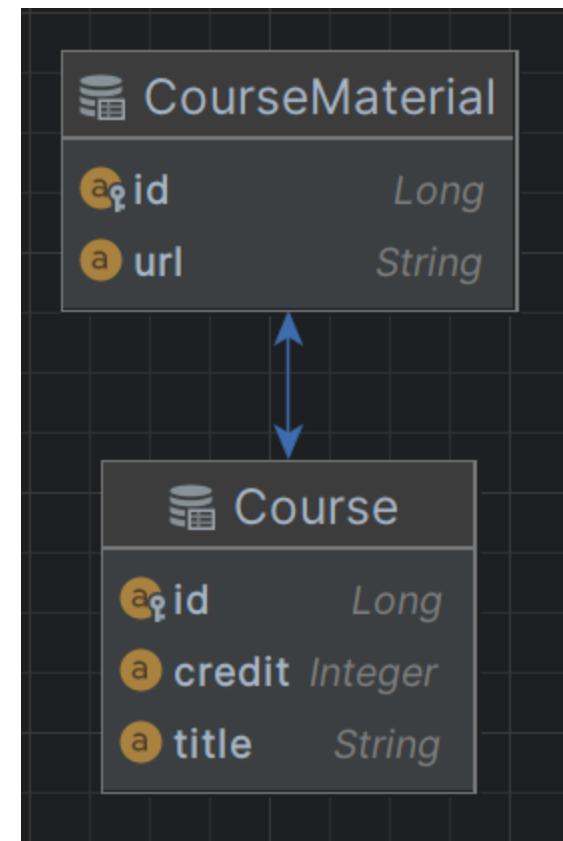
- Die mappedBy Option von @OneToOne gibt an, dass die Zuordnung in einer anderen Spalte stattfindet.
- Gebe den Namen des Feldes an, nicht den Tabellennamen!

Course.java

```
@OneToOne(cascade = CascadeType.PERSIST)  
private CourseMaterial courseMaterial;
```

CourseMaterial.java

```
22 @OneToOne(mappedBy = "courseMaterial")  
23 private Course course;
```



# OneToMany Unidirektional

- Nutze @OneToMany Annotation.
- Wenn auf @JoinColumn verzichtet wird, wird eine extra Tabelle für den Join erstellt.
- Der Parameter „name“ sagt den Spaltennamen an.
- Der Parameter „referencedColumnName“ sagt den im Ziel referenzierten Spaltennamen an. Wenn nicht gesetzt, wird der Primary Key verwendet.

```
27     @OneToMany(cascade = CascadeType.ALL)
28     @JoinColumn(
29         name = "taughtBy",
30         referencedColumnName = "id"
31     )
32     private Set<Course> taughtCourses = new LinkedHashSet<>();
```

The screenshot shows a database interface with two tables:

**COURSE** table:

ID	CREDIT	TITLE	COURSE_MATERIAL_ID	TAUGHT_BY
1	1	20	Analysis	1

**TEACHER** table:

ID	FIRST_NAME	LAST_NAME
1	Oli	Kohl

# ManyToOne

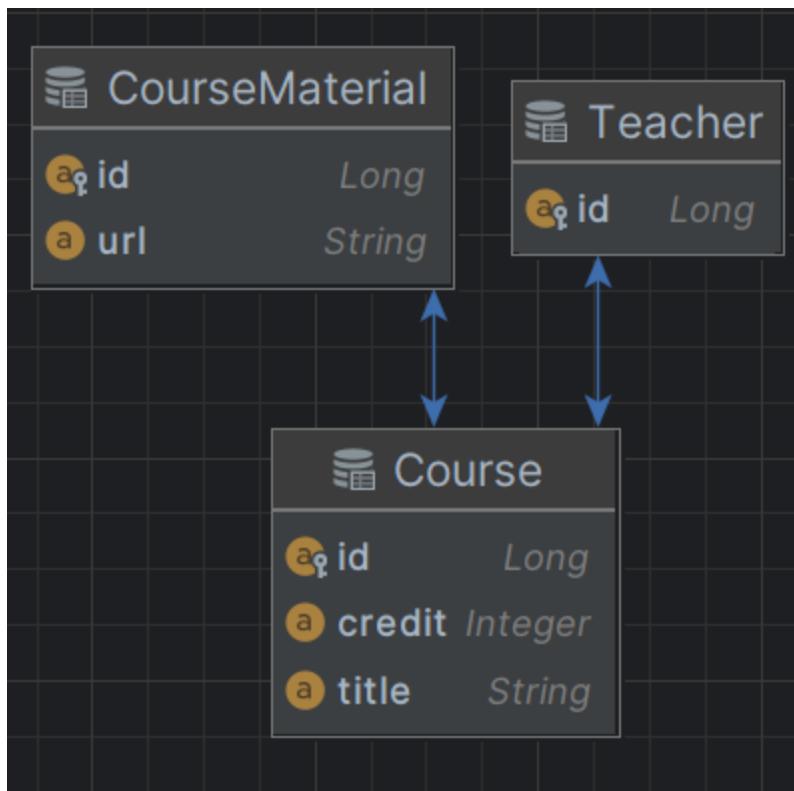
- Alternativ kann @ManyToOne verwendet werden.
- Dies ist empfohlen.
- Es wird hier keine Tabelle Jointabelle erstellt.

Course.java

```
28     @ManyToOne(cascade = CascadeType.ALL)  
29     private Teacher taughtBy;
```

# ManyToOne bidirektional

- Füge wieder das mappedBy Attribut hinzu.



```
Course.java
28 @ManyToOne(cascade = CascadeType.ALL)
29 private Teacher taughtBy;
```

```
Teacher.java
27 @OneToMany(cascade = CascadeType.ALL, mappedBy = "taughtBy")
28 private Set<Course> taughtCourses = new LinkedHashSet<>();
```

# Fake Daten erzeugen

- <https://mvnrepository.com/artifact/com.github.javafaker/javafaker/1.0.2>

```
45      <!-- https://mvnrepository.com/artifact/com.github.javafaker/javafaker -->
46      <dependency>
47          <groupId>com.github.javafaker</groupId>
48          <artifactId>javafaker</artifactId>
49          <version>1.0.2</version>
50      </dependency>
```

```
7  @Configuration
8  public class fakerConfig {
9      @Bean
10     public Faker faker() {
11         return new Faker();
12     }
13 }
```

```
12  @Component
13  public class FakeMachine {
14      @Autowired
15      private Faker faker;
16
17      public Student fakeStudent() {
18          String firstName = faker.name().firstName();
19          String lastName = faker.name().lastName();
20          String email = firstName + "." + lastName + "@" + faker.internet().domainName();
21          return Student.builder().lastName(lastName).firstName(firstName).email(email).build();
22      }
23
24      public Iterable<Student> fakeStudents(int n) {
25          return IntStream.range(0, n).mapToObj(i -> fakeStudent()).collect(Collectors.toList());
26      }
27  }
```

# Paging and Sorting

```
Pageable firstTen = PageRequest.of( page: 0, size: 10);
Pageable thirdTen = PageRequest.of( page: 2, size: 10);

studentRepository.findAll(firstTen).getContent().forEach(System.out::println);
System.out.println();
studentRepository.findAll(thirdTen).getContent().forEach(System.out::println);
```

```
Hibernate:
select
    s1_0.id,
    s1_0.email,
    s1_0.first_name,
    s1_0.guardian_email,
    s1_0.guardian_mobile,
    s1_0.guardian_name,
    s1_0.last_name
from
    student s1_0 offset ? rows fetch first ? rows only
```

```
studentRepository.saveAll(fakeMachine.fakeStudents( n: 10000));
Pageable firstTen = PageRequest.of(
    page: 0,
    size: 10,
    Sort.by( ...properties: "firstName").descending()
        .and(Sort.by( ...properties: "lastName")));
studentRepository.findAll(firstTen).getContent().forEach(System.out::println);
```

```
select
    s1_0.id,
    s1_0.email,
    s1_0.first_name,
    s1_0.guardian_email,
    s1_0.guardian_mobile,
    s1_0.guardian_name,
    s1_0.last_name
from
    student s1_0
order by
    s1_0.first_name desc,
    s1_0.last_name asc offset ? rows fetch first ? rows only
```

# Query Methode

- Es lassen sich auch Query Methoden erstellen:

## StudentRepository

```
Page<Student> findByFirstNameContainsIgnoreCaseOrderByFirstNameAsc(String firstName, Pageable pageable);
```

```
studentRepository.findByFirstNameContainsIgnoreCaseOrderByFirstNameAsc(  
    firstName: "Vi",  
    PageRequest.of( page: 0, size: 20))  
.getContent().forEach(System.out::println);
```

```
Student(guardian=null, id=3653, firstName=Alvin, lastName=Bins, email=Alvin.Bins@barrows.com)  
Student(guardian=null, id=3538, firstName=Alvin, lastName=Erdman, email=Alvin.Erdman@dach.name)  
Student(guardian=null, id=458, firstName=Alvina, lastName=Hessel, email=Alvina.Hessel@quigley.net)  
Student(guardian=null, id=4381, firstName=Calvin, lastName=Gutmann, email=Calvin.Gutmann@rolfson.com)  
Student(guardian=null, id=3489, firstName=Calvin, lastName=Wiegand, email=Calvin.Wiegand@bahringer.name)  
Student(guardian=null, id=2453, firstName=Calvin, lastName=Davis, email=Calvin.Davis@hodkiewicz.net)  
Student(guardian=null, id=4259, firstName=David, lastName=Romaguera, email=David.Romaguera@macejkovic.org)  
Student(guardian=null, id=3044, firstName=Davida, lastName=Herman, email=Davida.Herman@blanda.io)  
Student(guardian=null, id=201, firstName=Davina, lastName=Wilkinson, email=Davina.Wilkinson@homenick.io)  
Student(guardian=null, id=6513, firstName=Davis, lastName=Howe, email=Davis.Howe@hessel.com)  
Student(guardian=null, id=1197, firstName=Davis, lastName=Erdman, email=Davis.Erdman@jast.co)  
Student(guardian=null, id=4823, firstName=Devin, lastName=Feest, email=Devin.Feast@runolfsson.org)  
Student(guardian=null, id=4848, firstName=Devin, lastName=Kertzmann, email=Devin.Kertzmann@brown.io)  
Student(guardian=null, id=5311, firstName=Devin, lastName=Breitenberg, email=Devin.Breitenberg@pfeffer.org)  
Student(guardian=null, id=3911, firstName=Devin, lastName=Heller, email=Devin.Heller@rice.biz)  
Student(guardian=null, id=5889, firstName=Devin, lastName=Goodwin, email=Devin.Goodwin@runte.com)  
Student(guardian=null, id=3618, firstName=Devin, lastName=Renner, email=Devin.Renner@oconnor.name)  
Student(guardian=null, id=2527, firstName=Devin, lastName=Dickens, email=Devin.Dickens@gutmann.co)  
Student(guardian=null, id=7075, firstName=Devin, lastName=Lang, email=Devin.Lang@kutch.io)  
Student(guardian=null, id=836, firstName=Devin, lastName=Kunde, email=Devin.Kunde@bechtelar.info)
```

# ManyToMany

- Hier muss eine JoinTable definiert werden, die in der Datenbank

```
34     @ManyToMany  
35     @JoinTable(name = "course_participants",  
36         joinColumns = @JoinColumn(name = "course_id"),  
37         inverseJoinColumns = @JoinColumn(name = "participants_id"))  
38     private Set<Student> participants = new LinkedHashSet<>();
```

COURSE_PARTICIPANTS		
WHERE		ORDER BY
COURSE_ID	PARTICIPANTS_ID	
1	1	721
2	1	891
3	1	1520
4	1	1618
5	1	2424
6	1	2573
7	1	3019
8	1	4147
9	1	4536
10	1	6729
11	1	7234
12	1	7441
13	1	7586
14	1	7684
15	1	8045
16	1	8235
17	1	8288
18	1	8509
19	1	9275
20	1	9982

# Transaktionen

- Transaktionen managen die Änderungen, die man an einem System vornimmt (Datenbanken, ...). Dabei werden die ACID Charakteristika eingehalten:
  - Atomicity (Alles oder nichts wird commitet)
  - Consistency (System bleibt in einem konsistenten Zustand)
  - Isolation (Änderungen einer Transaktion sind vor dem Commit nicht für andere Transaktionen sichtbar)
  - Durability (stellt sicher, dass commitete Änderungen auch persistent sind.)
- JDBC stellt dieses Transaktionsmanagement zur Verfügung.
- Spring stellt bequeme Wege bereit, diese zu verwenden.

# Transaktionen ohne Spring

- Start: Connection aufstellen & auto-commit deaktivieren.
- Commit: Wenn alle Operationen getan sind, mit commit() persistieren.
- Bei Fehler: Einen Rollback mit rollback() ausführen
- Theoretisch nachvollziehbar, praktisch schwer zu implementieren.

```
import java.sql.Connection;

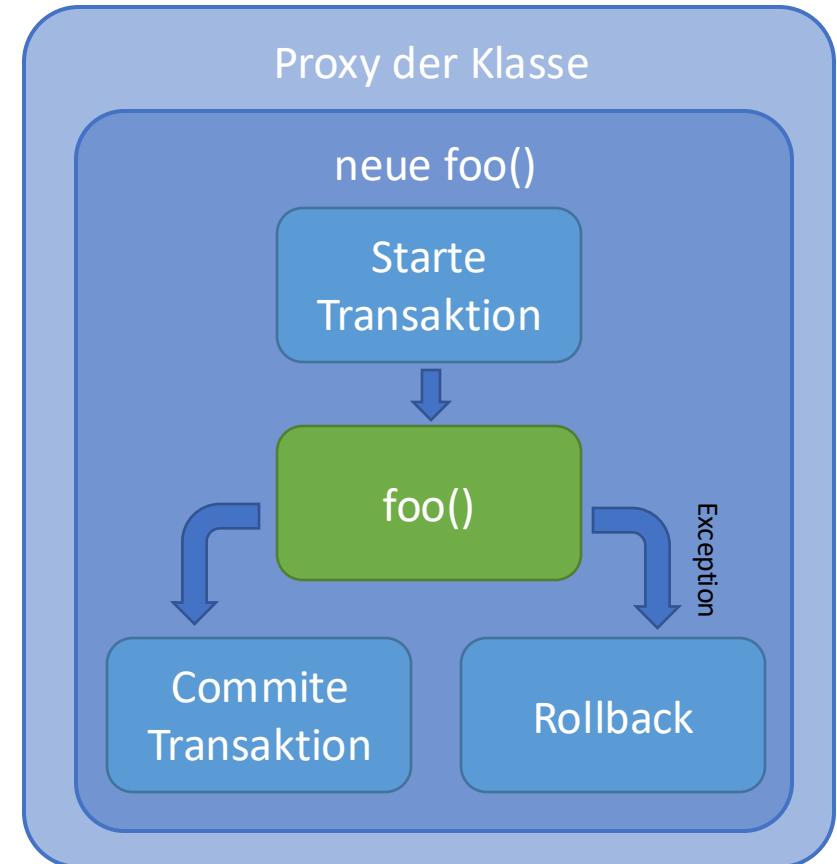
Connection connection = dataSource.getConnection(); // (1)

try (connection) {
    connection.setAutoCommit(false); // (2)
    // execute some SQL statements...
    connection.commit(); // (3)

} catch (SQLException e) {
    connection.rollback(); // (4)
}
```

# Spring Transaktionmanagement

- Wenn kein Spring Boot verwendet wird, muss die Annotation `@EnableTransactionManagement` gesetzt werden, um Transaktionen zu ermöglichen.
- Spring erstellt für die Methoden, die mit `@Transactional` annotiert ein Proxy. Kann an Klasse oder an Methode gehängt werden.
- Wichtigste Attribute von `@Transactional`:
  - `readOnly`
  - `rollbackFor`
  - `noRollbackFor`
  - `propagation`



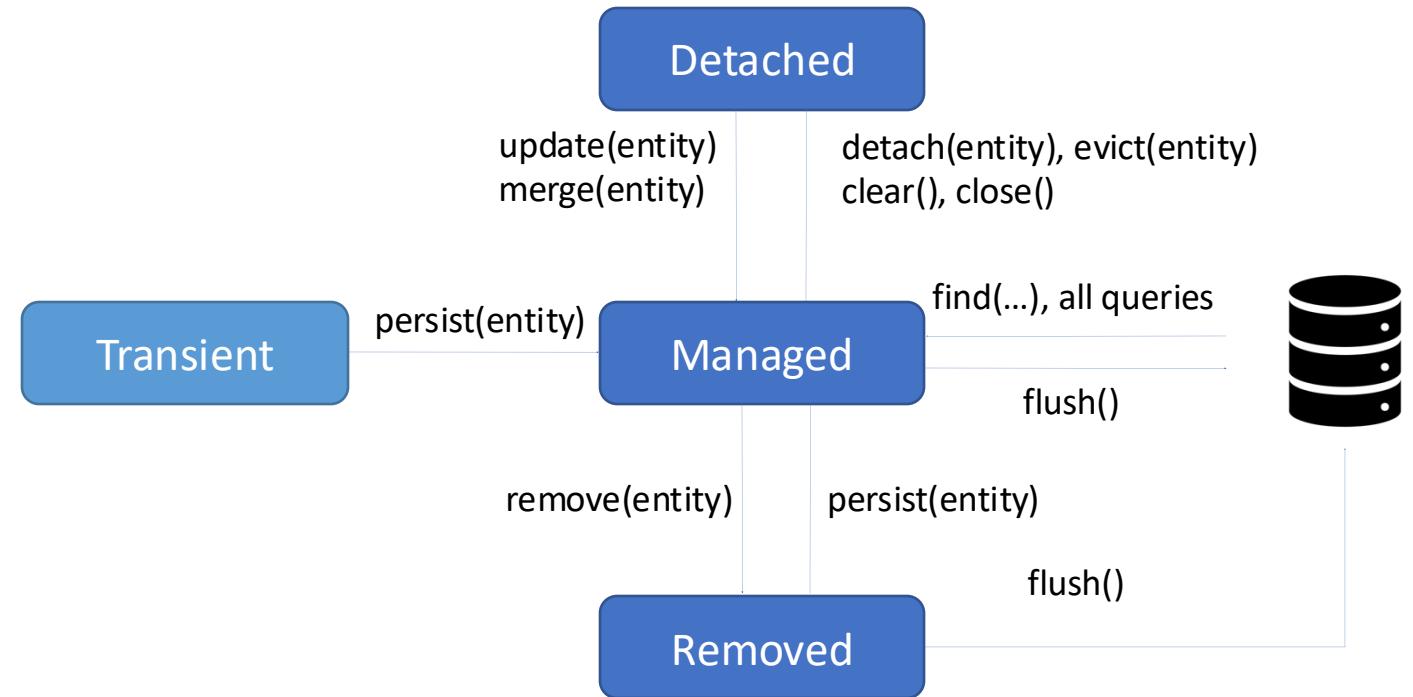
# Propagationtypes

Type	Bedeutung
Required (default)	Füge dich einer bestehenden Transaktion hinzu oder erstelle eine neue.
Supports	Füge dich einer bestehenden Transaktion hinzu. Ansonsten wird die Methode ohne Transaktionskontext durchgeführt.
Mandatory	Füge dich einer bestehenden Transaktion hinzu. Ansonsten werfe eine Exception.
Never	Werfe eine Exception, wenn diese Methode im Rahmen einer Transaktion aufgerufen wird.
Not_supported	Pausiere die momentane Transaktion und führe die Methode ohne Transaktionskontext aus.
Requires_new	Starte immer eine neue Transaktion. Pausiere ggf. die momentane.
Nested	Kann innerhalb einer Transaktion aufgerufen werden. Dann wird ein Savepoint gesetzt, zu dem im Fall eines Rollbacks zurückgesprungen wird.

# Entity Lifecycle Model

- Ein Persistence Context verwaltet Entity-Instanzen und ihre Lifecycles.

State	Bedeutung.
Transient	Neues Objekt, das nicht überwacht ist. (POJO)
Managed	Objekt stellt Inhalte der Datenbank dar. Wird von Persistence Context überwacht.
Detached	Objekt wird nicht mehr von Persistence Context überwacht.
Removed	Objekt ist zum Löschen markiert.

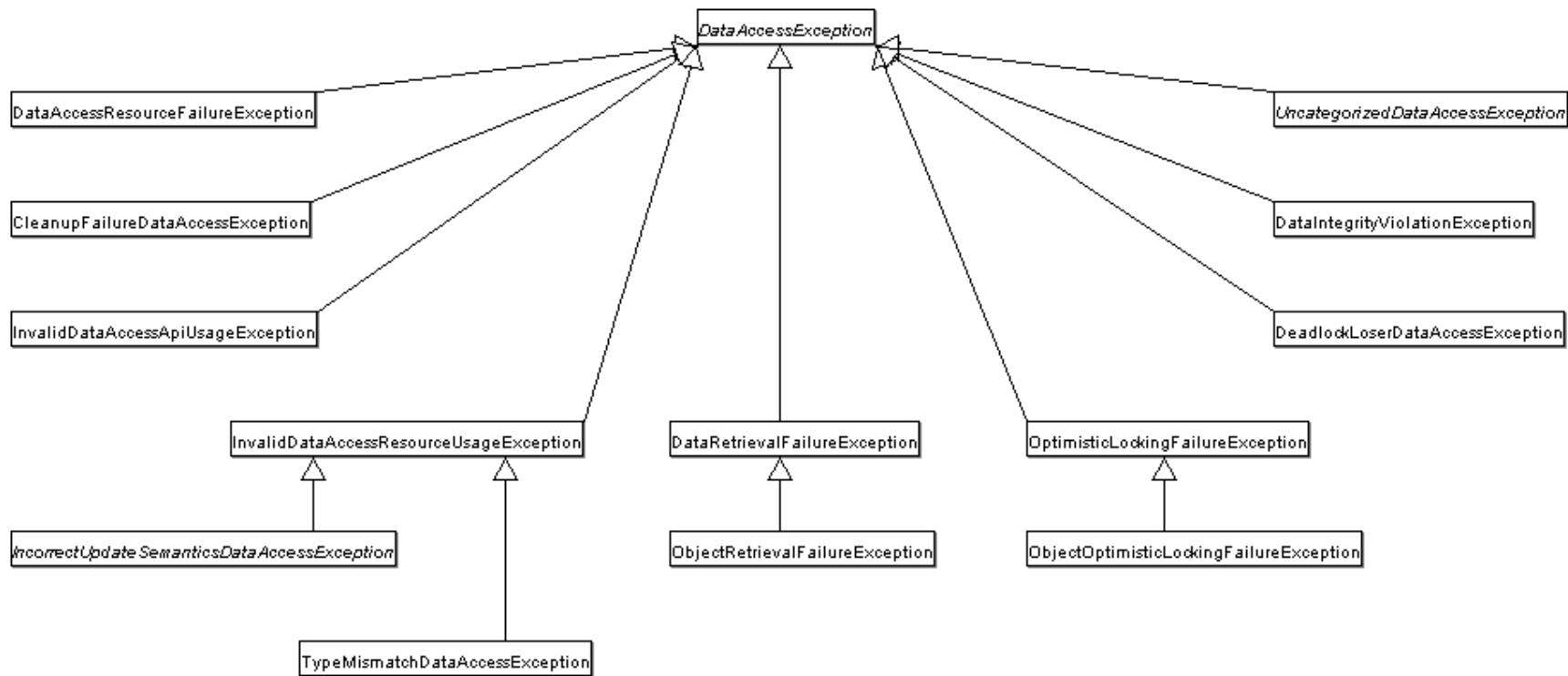


# EntityManager, Datasource programmatisch erstellen

```
12 @Configuration
13 public class DevDatabaseConfig {
14     @Bean
15     public DataSource dataSource() {
16         JdbcDataSource dataSource = new JdbcDataSource();
17         dataSource.setURL("jdbc:h2:file:./databases/devH2/devH2Database;AUTO_SERVER=TRUE");
18         dataSource.setUser("sa");
19         dataSource.setPassword("");
20         return dataSource;
21     }
22
23     @Bean
24     public LocalContainerEntityManagerFactoryBean entityManagerFactory(DataSource dataSource) {
25         LocalContainerEntityManagerFactoryBean entityManagerFactory = new LocalContainerEntityManagerFactoryBean();
26         entityManagerFactory.setDataSource(dataSource);
27         entityManagerFactory.setPackagesToScan("com.example.training.model");
28         entityManagerFactory.setJpaVendorAdapter(new HibernateJpaVendorAdapter());
29         entityManagerFactory.setJpaProperties(jpaProperties());
30         return entityManagerFactory;
31     }
32
33     private Properties jpaProperties() {
34         Properties properties = new Properties();
35         properties.setProperty("hibernate.hbm2ddl.auto", "create-drop");
36         properties.setProperty("hibernate.dialect", "org.hibernate.dialect.H2Dialect");
37         properties.setProperty("hibernate.show_sql", "true");
38         return properties;
39     }
40 }
41 }
```

# Exceptions

- Spring übersetzt Exceptions (z.B. von Hibernate, JPA, JDBC,...) in DataAccessExceptions (Wrapper).



# Eigenes DAO erstellen

- JPARepository ist ein DAO (Data Access Object).
- Es ist auch möglich eigene DAOs zu definieren.
- Je nach verwendeter Technologie werden andere Objekte zum Datenbankzugriff genutzt.

```
@Repository  
public class JpaMovieFinder implements MovieFinder {  
  
    @PersistenceContext  
    private EntityManager entityManager;  
  
    // ...  
}
```

```
@Repository  
public class HibernateMovieFinder implements MovieFinder {  
  
    private SessionFactory sessionFactory;  
  
    @Autowired  
    public void setSessionFactory(SessionFactory sessionFactory) {  
        this.sessionFactory = sessionFactory;  
    }  
  
    // ...  
}
```

```
@Repository  
public class JdbcMovieFinder implements MovieFinder {  
  
    private JdbcTemplate jdbcTemplate;  
  
    @Autowired  
    public void init(DataSource dataSource) {  
        this.jdbcTemplate = new JdbcTemplate(dataSource);  
    }  
  
    // ...  
}
```

# schema.sql und data.sql verwenden.

- Tabellen können auch über sql-Files geladen werden.
- Erstelle die Entities dann entsprechend.

## application.properties

```
10 # Keine Tabellengenerierung aufgrund von Annotationen
11 spring.jpa.hibernate.ddl-auto=none
12
13 # Tabellenschemas
14 spring.sql.init.schemaLocations=classpath:schema.sql
15
16 # Zeilen
17 spring.sql.init.dataLocations=classpath:data.sql
18
19 # Nutze die sql Dateien!
20 spring.sql.init.mode=always
```

## resources/schema.sql

```
1 CREATE TABLE IF NOT EXISTS users (
2     id INTEGER PRIMARY KEY,
3     name VARCHAR(255),
4     email VARCHAR(255)
5 );
6
7 CREATE TABLE IF NOT EXISTS orders (
8     id INTEGER PRIMARY KEY,
9     user_id INTEGER,
10    total_price DECIMAL(10,2),
11    FOREIGN KEY (user_id) REFERENCES users(id)
12 );
```

## resources/data.sql

```
1 INSERT INTO users (id, name, email) VALUES (1, 'Alice', 'alice@example.com');
2 INSERT INTO users (id, name, email) VALUES (2, 'Bob', 'bob@example.com');
3
4 INSERT INTO orders (id, user_id, total_price) VALUES (1, 1, 99.99);
5 INSERT INTO orders (id, user_id, total_price) VALUES (2, 1, 199.99);
6 INSERT INTO orders (id, user_id, total_price) VALUES (3, 2, 299.99);
```

# Tabelle für spezielles Schema

- `@Table(schema = „schema“)` setzt das Schema für die Tabelle fest oder `@Table(„schema.tablename“)`
- Bei H2 Datenbanken werden Schemata nicht automatisch erstellt. Sie müssen also vorher angelegt werden.

# Kommunikation über Rest

- Wir können über RestTemplate andere Rest-Schnittstellen anzusprechen.

```
49 ⚡  <dependency>
50
51     <groupId>org.springframework.boot</groupId>
52     <artifactId>spring-boot-starter-web</artifactId>
53   </dependency>
```

```
25 ← @Bean
26 ↗ @
27     public RestTemplate restTemplate(RestTemplateBuilder builder) {
28         return builder.rootUri("http://localhost:8080").build();
29     }
30 ← @Bean
31 ↗ @ CommandLineRunner consumeRest(RestTemplate restTemplate) {
32     return args -> {
33         String answer =
34             restTemplate.getForObject(url: "/Maria", String.class);
35         System.out.println(answer);
36     };
37 }
```

# RMI (Remote Method Invocation)

- Nicht mehr verfügbar in Spring Boot ab Version 3!
- RMI dient dazu Methoden von Java Applicationen aufzurufen, die sich auf einer anderen JVM befinden.
- Für die Clientapplication liegen die Objekte dann wie gewöhnliche Beans vor.

# RMI Vorgehen

- Definiere die Interfaces
- Implementiere den Server
- Implementiere den Client.

# Definiere das Interface

- Das Interface kann in einem eigenen Projekt definiert werden. Dann muss das pom.xml wie links angepasst werden.
- Konsole: mvn clean install

```
1 package gmbh.conteco.colculator;  
2  
3 public interface Calculator {  
4     int add(int a, int b);  
5 }
```

```
32     <build>  
33         <plugins>  
34             <plugin>  
35                 <groupId>org.springframework.boot</groupId>  
36                 <artifactId>spring-boot-maven-plugin</artifactId>  
37             </plugin>  
38         </plugins>  
39     </build>  
40  
41     <build>  
42         <plugins>  
43             <plugin>  
44                 <groupId>org.springframework.boot</groupId>  
45                 <artifactId>spring-boot-maven-plugin</artifactId>  
46             </plugin>  
47         </plugins>  
48     </build>  
49  
50 </project>
```

# Implementiere das Interface auf dem Server

- Importiere das Interface als Dependency.
- spring-boot-starter-web wird benötigt.
- Erstelle eine Implementierung.

```
20  ↗ <dependency>
21          <groupId>org.springframework.boot</groupId>
22          <artifactId>spring-boot-starter-web</artifactId>
23      </dependency>
```

```
31          <dependency>
32              <groupId>gmbh.conteco</groupId>
33              <artifactId>CalculatorInterface</artifactId>
34              <version>0.0.1-SNAPSHOT</version>
35          </dependency>
36      </dependencies>
```

```
6  🌐 @Component
7  🔍 public class CalculatorImpl implements Calculator {
8
9  ↗   @Override
10     public int add(int a, int b) {
11         return a+b;
12     }
13 }
```

# Implementiere den Server

- In einer Configuration Klasse wird ein RmiServiceExporter erstellt.
- Das Interface wird implementiert und über einen port bereitgestellt.

```
3 import gmbh.conteco.calculator.Calculator;
4 import org.springframework.stereotype.Component;
5
6 @Component
7 public class CalculatorImpl implements Calculator {
8     @Override
9     public int add(int a, int b) { return a+b; }
12 }
```

```
application.properties
1 rmi.service.name=calcservice
2 rmi.service.port=1000
```

```
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
```

```
@Configuration
public class RmiServerConfiguration {

    @Value("${rmi.service.name}")
    public String serviceName;

    @Value("${rmi.service.port}")
    public int port;

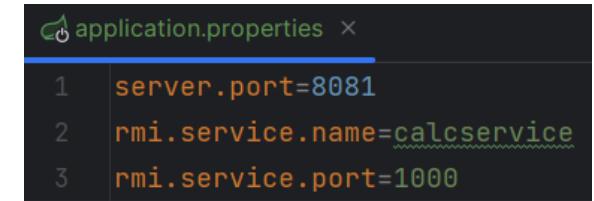
    @Bean
    public RmiServiceExporter registerRmiServiceExporter(
        Calculator calculator) {
        RmiServiceExporter exporter = new RmiServiceExporter();
        exporter.setServiceName(serviceName);
        exporter.setServiceInterface(Calculator.class);
        exporter.setService(calculator);
        exporter.setRegistryPort(port);

        return exporter;
    }
}
```

# Implementiere den Client

- Importiere das Interface als Dependency.
- spring-boot-starter-web wird benötigt.
- Implementiere eine RmiProxyFactoryBean.

```
12 ❸ @SpringBootApplication
13 ► public class RmiClientApplication {
14 ❹     @Bean
15         RmiProxyFactoryBean rmiProxy(
16             @Value("${rmi.service.name}") String serviceName,
17             @Value("${rmi.service.port}") int servicePort
18         ) {
19             RmiProxyFactoryBean bean = new RmiProxyFactoryBean();
20             bean.setServiceInterface(Calculator.class);
21             bean.setServiceUrl("rmi://localhost:"+servicePort+"/"+serviceName);
22             return bean;
23         }
24
25 ►     public static void main(String[] args) {
26         ConfigurableApplicationContext context =
27             SpringApplication.run(RmiClientApplication.class, args);
28         Calculator calculator = context.getBean(Calculator.class);
29
30         System.out.println(calculator.add(a: 1, b: 2));
31     }
32 }
```



```
server.port=8081
rmi.service.name=calcservice
rmi.service.port=1000
```

# http Remoting

- Eine Alternative zu RMI bildet das http Remoting von Spring.
- Das vorherige Beispiel kann übernommen werden. Die RMI-Spezifischen Klassen werden durch folgende ersetzt...
  - HttpInvokerProxyFactory beim Client
  - HttpInvokerServiceExporter beim Server

# http Remoting Serverseite

- Der Name der Bean bestimmt, wie auf den Service zugegriffen wird.

```
8  @Configuration
9
10 @Bean(name = "/calculator")
11     HttpInvokerServiceExporter calculatorService(
12         Calculator calculator
13     ) {
14         HttpInvokerServiceExporter exporter =
15             new HttpInvokerServiceExporter();
16
17         exporter.setService(calculator);
18         exporter.setServiceInterface(Calculator.class);
19         return exporter;
20     }
21 }
```

# http Remoting Client

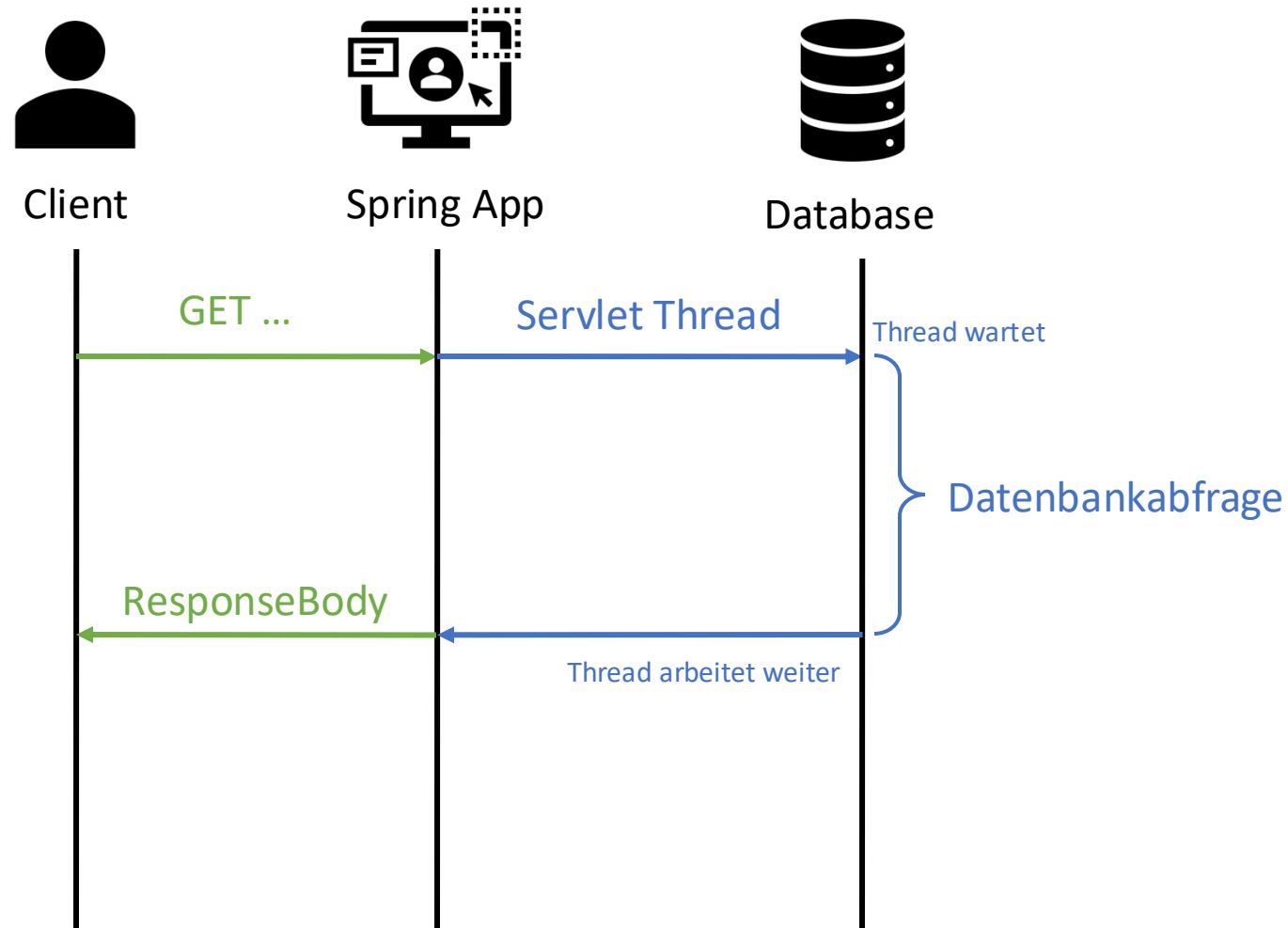
- Verwende eine `HttpInvokerProxyFactoryBean` auf Seite des Clients.
- Es wird keine besondere rmi-url verwendet.

```
28 @Bean
29     public HttpInvokerProxyFactoryBean invoker() {
30         HttpInvokerProxyFactoryBean invoker =
31             new HttpInvokerProxyFactoryBean();
32         invoker.setServiceUrl("http://localhost:8080/calculator");
33         invoker.setServiceInterface(Calculator.class);
34         return invoker;
35     }
```

# Threads bei Spring MVC

- Bei Spring MVC wird für jede Anfrage ein eigener Servlet Thread erstellt.
- Es stehen eine maximale Anzahl an Servlet Threads zur Verfügung, die mit [server.tomcat.threads.max](#) angepasst werden können.
- Der Servlet Threads ist blockiert z.B. bei Datenbankabfragen, bis eine Antwort erhalten wurde und steht so nicht für andere Anfragen zur Verfügung.
- Skalierung ist beschränkt durch die Anzahl verfügbarer Servlet Threads.

# Threads in Spring MVC



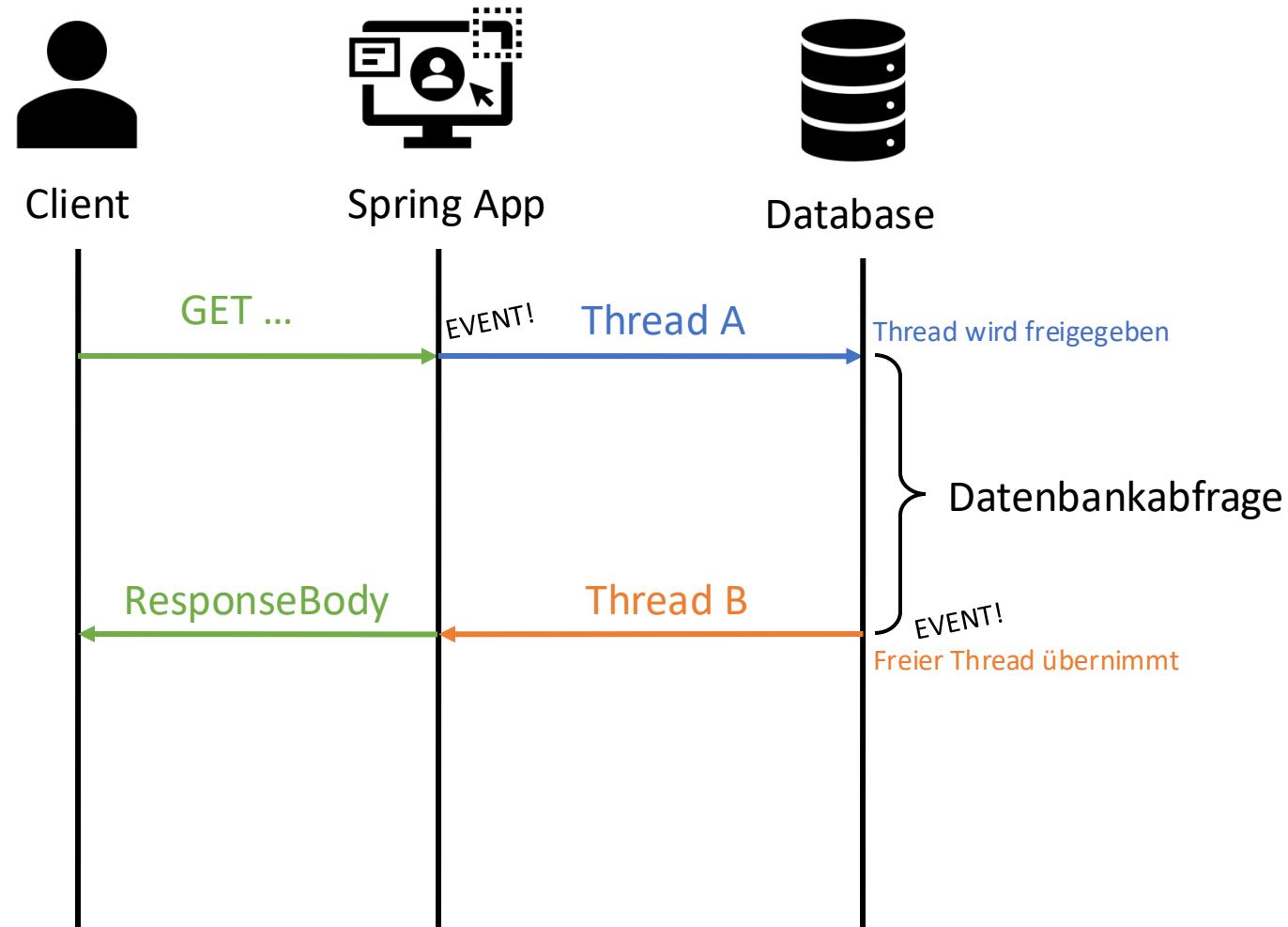
# Threads in Spring MVC

- Wenn mehr Anfragen als Threads zur Verfügung stehen, kommt es zu Wartezeiten bei den Clients.
- Das Handling von 200+ Threads ist keine optimale Auslastung der CPU.
- Optimal wäre ein Thread pro Kern.

# Let's think reactive!

- Bei einer reaktiven Kommunikation wird nicht blockierend gearbeitet.
- Bei einer Anfrage wartet der Client nicht bis er eine Antwort erhält, sondern er erhält von der Applikation ein Mono- oder Flux-Objekt, indem sich später die Antwort finden lassen wird.
- Der Thread kann so vorzeitig freigegeben werden, und steht so für andere Anfragen zur Verfügung.
- Wenn tiefer liegende Systeme Daten bereit stellen (z.B. hat die Datenbank alle Daten gefetcht) löst dies ein Event aus, dass von einem Thread bearbeitet wird.
- Man spricht hier vom **Event-Loop**.

# Threads in Spring Flux



# Beispiel laden

- Zunächst erstellen wir ein neues Projekt mit den Dependencies:
  - H2 Database
  - Lombok
  - Spring Reactive Web
  - Spring Data R2DBC
- Wir fügen `@EnableWebFlux` unserer Main-Klasse hinzu.

```
19 @SpringBootApplication
20 @EnableWebFlux
21 public class WebfluxSpringTutorialApplication {
22
23     public static void main(String[] args) {
24         SpringApplication.run(WebfluxSpringTutorialApplication.class, args);
25     }
26 }
```

# Faker einbinden

- Um zufällige Daten zu erzeugen verwenden wir Faker.
- Wir stellen Faker über eine Bean bereit.

```
27 @Bean  
28     public Faker faker() {  
29         return new Faker();  
30     }
```

```
54 <dependency>  
55     <groupId>com.github.javafaker</groupId>  
56     <artifactId>javafaker</artifactId>  
57     <version>1.0.2</version>  
58 </dependency>
```

# Beispiel 1

- Wir erstellen einen Controller, der zufällige Namen generiert und diese zurückgibt.

```
14  @RestController
15  @RequiredArgsConstructor
16  public class ReactiveNameController {
17      public final Faker faker;
18
19      @GetMapping("/{names}")
20      public Flux<String> getNames(
21          @RequestParam(defaultValue = "10") int amount
22      ) {
23          List<String> names = Stream
24              .generate(() -> faker.name().fullName() + " ")
25              .limit(amount)
26              .toList();
27
28          return Flux.fromIterable(names)
29              .delayElements(Duration.ofSeconds(1));
30      }
31  }
```

# Beispiel 2 - Controller + Entity

- Wir erstellen nun einen Controller um User abzufragen und zu generieren.
- Hier werden nun Mono und Flux zurückgeben.

```
9  @Getter
10 @NoArgsConstructor
11 @AllArgsConstructor
12 @Builder
13 @Table("USERS")
14 public class User {
15     @Id
16     public Long id;
17     public String firstName;
18     public String lastName;
19 }
```

```
11 @RestController
12 @RequiredArgsConstructor
13 public class UserController {
14     private final UserService userService;
15
16     @GetMapping("/users")
17     @ResponseStatus(HttpStatus.OK)
18     public Flux<User> getAllUsers() {
19         return userService.getAllUsers();
20     }
21
22     @PostMapping("/users/new")
23     @ResponseStatus(HttpStatus.CREATED)
24     public Flux<User> createNewUsers(
25         @RequestParam(defaultValue = "10") int amount
26     ) {
27         return userService.createNewUsers(amount);
28     }
29
30     @GetMapping("/users/{id}")
31     @ResponseStatus(HttpStatus.OK)
32     public Mono<User> getUserId(@PathVariable Long id){
33         return userService.getUserId(id);
34     }
35
36     @GetMapping("/users/")
37     @ResponseStatus(HttpStatus.OK)
38     public Flux<User> getUserName(@RequestParam String name){
39         return userService.getUserName(name);
40     }
41 }
```

# Beispiel 2 - Service

- Der Service ist für die Erstellung von Namen zuständig und die Kommunikation mit dem Repository.
- Die Rückgabe der Methoden sind Mono und Flux

```
15 @Service
16 @RequiredArgsConstructor
17 public class UserService {
18     private final UserRepository userRepository;
19     private final Faker faker;
20
21     private User createNewRandomUser() {
22         return User.builder()
23             .firstName(faker.name().firstName())
24             .lastName(faker.name().lastName())
25             .build();
26     }
27
28     public Flux<User> getAllUsers() {
29         return userRepository.findAll();
30     }
31
32     public Flux<User> createNewUsers(int amount) {
33         return userRepository.saveAll(
34             Stream.generate(this::createNewRandomUser)
35                 .limit(amount)
36                 .toList()
37         );
38     }
39
40     public Mono<User> getUserById(Long id) {
41         return userRepository.findById(id);
42     }
43
44     public Flux<User> getUserByName(String name) {
45         return userRepository.getUserByName(name);
46     }
47 }
```

# Beispiel 2 - Repository

- Wir leiten nun von R2dbcRepository ab.
- Die Methoden geben Mono oder Flux zurück.

```
8  public interface UserRepository extends R2dbcRepository<User, Long> {  
9      @Query("SELECT * FROM USERS WHERE USERS.first_name = ?1 OR USERS.last_name = ?1")  
10     Flux<User> getUserByName(String name);  
11 }
```

# Datenbank anlegen und füllen

- Die Datenbankstruktur wird nicht automatisch generiert, sondern muss bereitgestellt werden.
- Jetzt kann der Service gestartet und genutzt werden.
- Beachte, dass POST-Requests nicht vom Browser aus abgesetzt werden können.

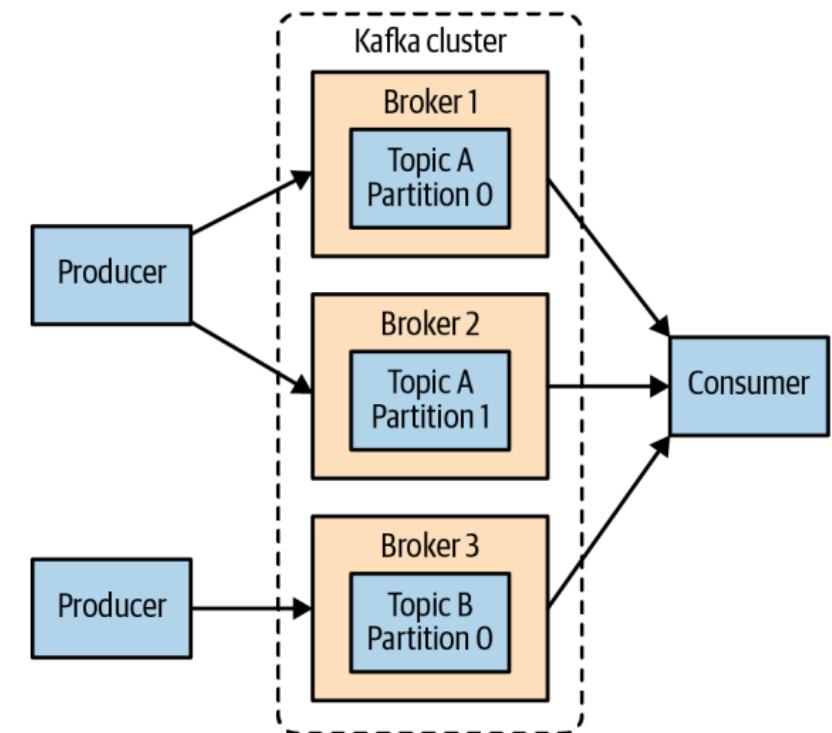
```
schema.sql ×  
1 DROP TABLE IF EXISTS "USERS";  
2  
3 CREATE TABLE "USERS"  
4 (  
5     id          BIGINT auto_increment PRIMARY KEY,  
6     first_name  VARCHAR(255),  
7     last_name   VARCHAR(255)  
8 )
```

```
data.sql ×  
1 INSERT INTO "USERS" (first_name, last_name) VALUES ('Viktor', 'Reichert')
```

```
32 @Bean  
33 public ConnectionFactoryInitializer databaseInitializer(  
34     ConnectionFactory connectionFactory  
35 ) {  
36     ConnectionFactoryInitializer initializer = new ConnectionFactoryInitializer();  
37     initializer.setConnectionFactory(connectionFactory);  
38  
39     CompositeDatabasePopulator populator = new CompositeDatabasePopulator();  
40     populator.addPopulators(  
41         new ResourceDatabasePopulator(new ClassPathResource("schema.sql")),  
42         new ResourceDatabasePopulator(new ClassPathResource("data.sql"))  
43     );  
44     initializer.setDatabasePopulator(populator);  
45  
46     return initializer;  
47 }
```

# Kommunikation mit Kafka

- Spring stellt ein [Modul](#) für die Erstellung von Kafka-Anwendungen bereit.
- Kafka ist ein Messaging System, dass die Nachrichtenübertragung vieler kleiner Nachrichten in verteilten Systemen übernimmt.
- Wir nutzen die Confluent Cloud, als Kafka Cluster in der Schulung.



# Properties

```
1 # Required connection configs for Kafka producer, consumer, and admin
2 spring.kafka.properties.bootstrap.servers=pkc-75m1o.europe-west3.gcp.confluent.cloud:9092
3 spring.kafka.properties.sasl.mechanism=PLAIN
4 spring.kafka.properties.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule \
5   required username='V3BKKSUQJVSMRAK' password='Wu4nplDbQaXDuGJAlINBmZ3ez7M0c657aLnTXAPWdJiFbxеutсJHKMLEaF5Y/WgR';
6 spring.kafka.properties.security.protocol=SASL_SSL
7 spring.kafka.properties.ssl.endpoint.identification.algorithm=https
8 spring.kafka.properties.request.timeout.ms=20000
9
10 spring.kafka.consumer.auto-offset-reset=earliest
11
12 # producer configuration
13 spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.StringSerializer
14 spring.kafka.producer.value-serializer=org.apache.kafka.common.serialization.StringSerializer
15 # consumer configuration
16 spring.kafka.consumer.key-deserializer=org.apache.kafka.common.serialization.StringDeserializer
17 spring.kafka.consumer.value-deserializer=org.apache.kafka.common.serialization.StringDeserializer
```

# Listener und Producer erstellen

- Listener können mit der Annotation `@KafkaListener` erstellt werden.
- Mit einem `KafkaTemplate<K,V>` (Key-type K und dem Value-type V) werden Nachrichten gesendet.

```
27      @KafkaListener(id = "myId", topics = "topic1")
28  ↗      public void listen(String value) { System.out.println(value); }
```

```
32 ⏪      @Bean
33 ⏴      public CommandLineRunner runner(KafkaTemplate<String, String> template) {
34      ↵          return args -> {
35              IntStream.range(0, 100)
36                  .forEach(i -> template.send(topic: "topic1", String.valueOf(i)));
37          };
38      }
39  }
```

# Was ist Batch Processing?

- Verarbeiten einer endlichen Menge von Daten, ohne Unterbrechung oder Interaktion (eines Nutzers).
- Use Case:
  - Big Data Processing
  - Trainieren von Modellen in Data Science
  - ETL (extract, transform, load) (Daten von A nach B transportieren)
  - Reporting (pro Nacht, Monat)
  - Nicht-Interaktive Verarbeitung

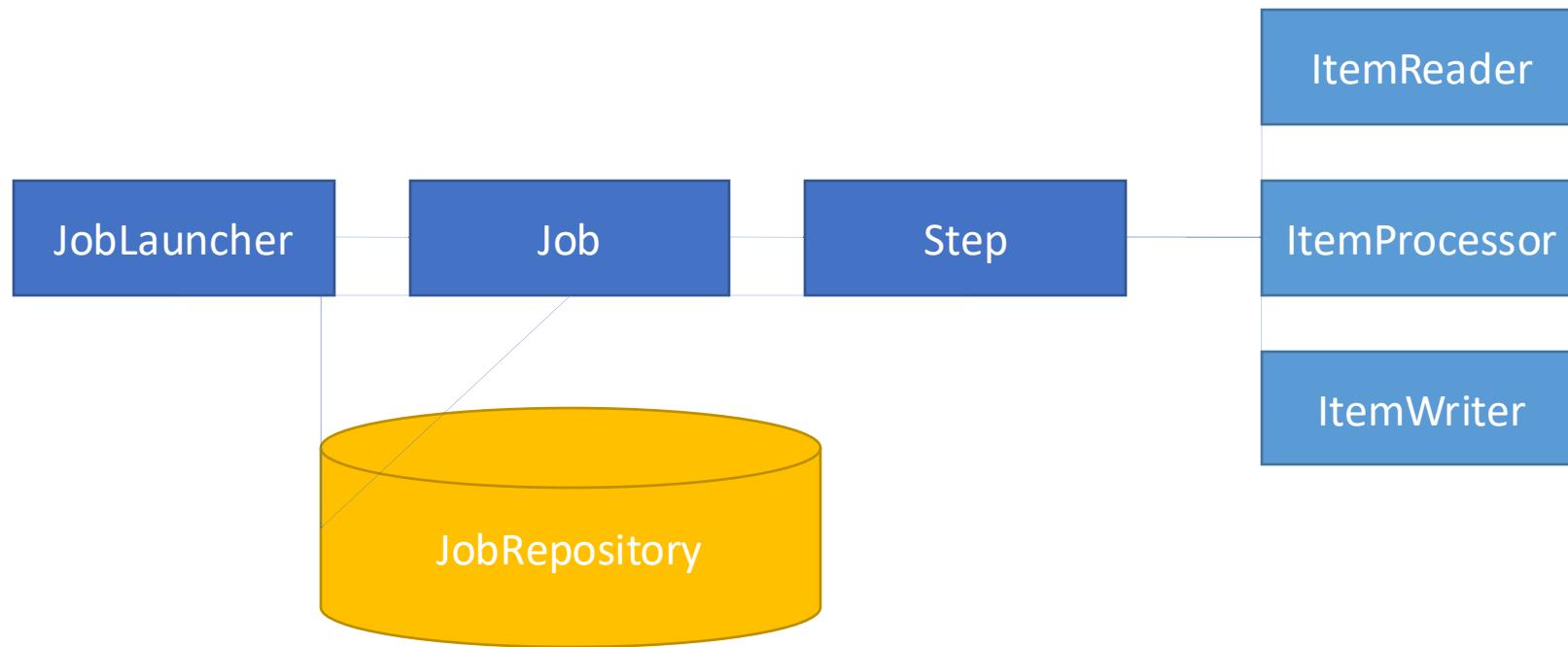
# Spring Batch

- Spring Batch ist DAS Framework für die Programmierung von Batchprozessen in der JVM.
- Bildete eine der Grundlagen für [JSR 352](#), welches einen standardisierten Ablauf von Batchprozessen in der JVM definiert. (Spring Batch setzt diesen auch um)

# Features von Spring Batch

- Job Flow State Machine
- Transaction Handling
- Declarative I/O (Viele Reader/Writer/Processoren sind vorbereitet)
- Robust Error Handling
- Optionen zur Skalierung (innerhalb einer JVM und über viele)
- In der Wirtschaft etabliert ;)

# Grundstrukturen



# Konfiguration der h2 Datenbank

- Mit `spring.batch.jdbc.initialize-schema=always` wird sichergestellt, dass das Datenbankschema von Spring Batch erstellt wird.

```
1 spring.datasource.url=jdbc:h2:file:./databases/jobRepo/jobRepo;AUTO_SERVER=true
2 spring.datasource.username=sa
3 spring.datasource.password=
4 spring.datasource.driverClassName=org.h2.Driver
5 spring.jpa.hibernate.ddl-auto=create-drop
6
7 spring.batch.jdbc.initialize-schema=always
```

# Erstes Beispiel

```
19  @Configuration
20  public class JobConfiguration {
21
22      @Autowired
23      private JobRepository jobRepository;
24
25      @Autowired
26      private PlatformTransactionManager transactionManager;
27
28      @Bean
29      Job firstJob() {
30          Tasklet tasklet = (contribution, chunkContext) -> {
31              System.out.println("Hallo Welt");
32              return RepeatStatus.FINISHED;
33          };
34
35          Step step = new StepBuilder( name: "Hello_World_StepBuilder", jobRepository)
36              .tasklet(tasklet, transactionManager).build();
37
38          return new JobBuilder( name: "Hello_World_JobBuilder", jobRepository)
39              .start(step).build();
40
41      }
42  }
```

The code snippet shows a Java configuration class for a Spring Batch job. It defines a job named 'firstJob' which contains a single step. The step uses a tasklet that prints 'Hallo Welt' to the console and returns a finished status. The step is built using a StepBuilder with the name 'Hello\_World\_StepBuilder'. Finally, the job is built using a JobBuilder with the name 'Hello\_World\_JobBuilder'.

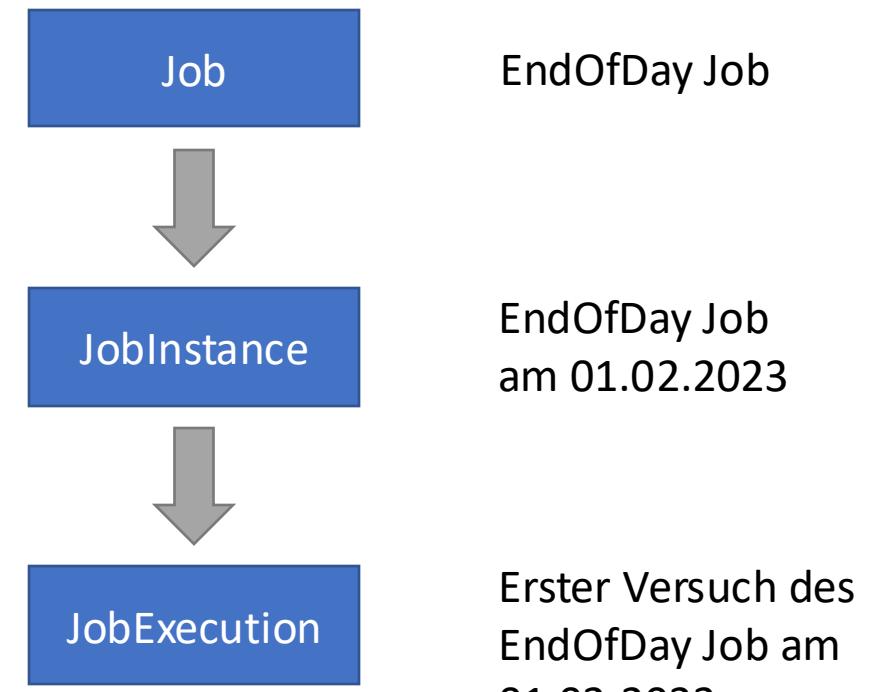
# JobLauncher einfügen

9 | `spring.batch.job.enabled=false`

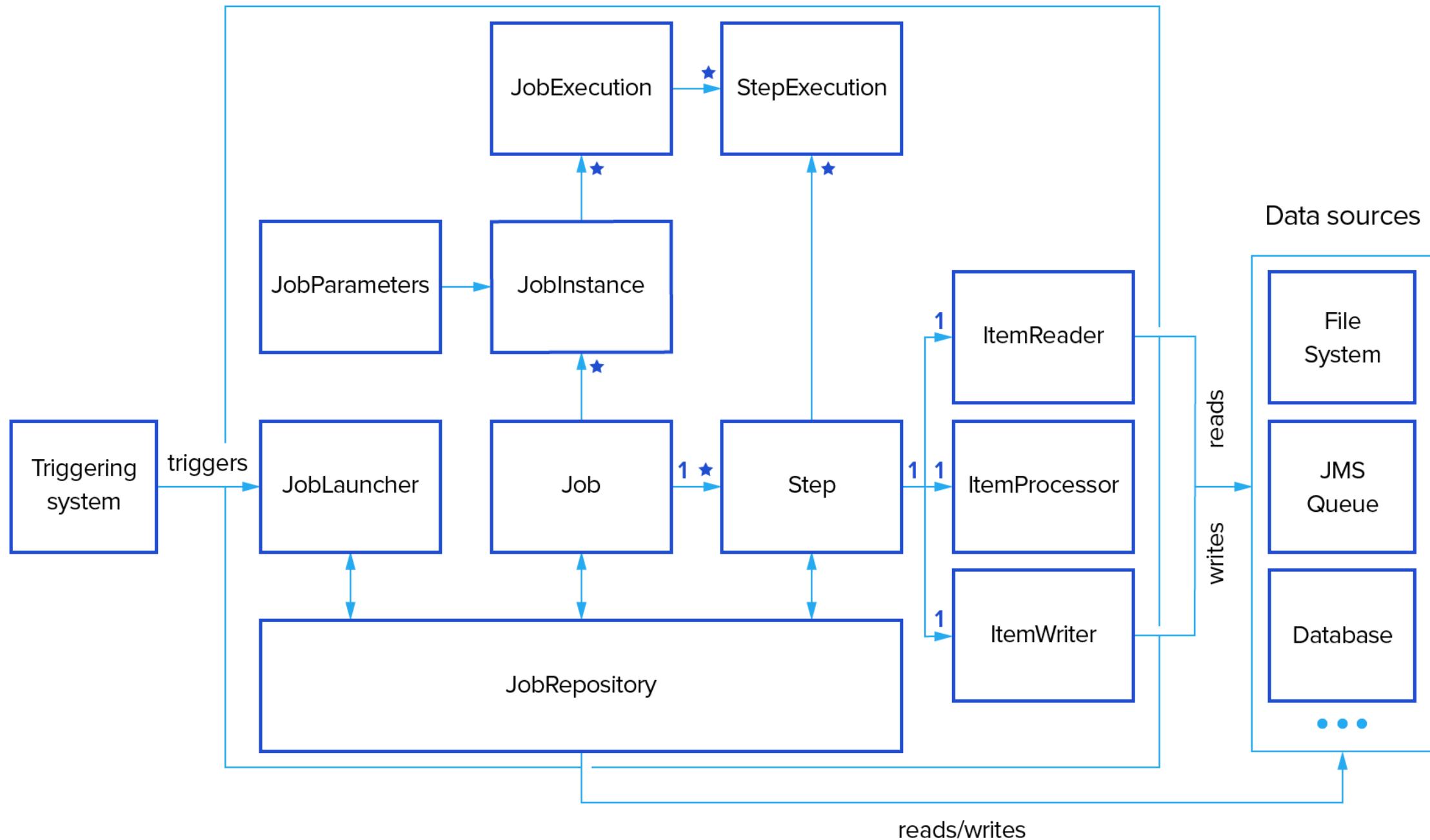
```
15 @SpringBootApplication
16 public class SpringBatchTutorialApplication {
17
18     public static void main(String[] args) throws JobInstanceAlreadyCompleteException,
19             JobExecutionAlreadyRunningException, JobParametersInvalidException, JobRestartException {
20
21         ConfigurableApplicationContext context
22             = SpringApplication.run(SpringBatchTutorialApplication.class, args);
23
24         JobParameters jobParameters
25             = new JobParametersBuilder()
26                 .addLong( key: "startAt", System.currentTimeMillis())
27                 .toJobParameters();
28
29         Job job = context.getBean(Job.class);
30         JobLauncher jobLauncher = context.getBean(JobLauncher.class);
31
32         jobLauncher.run(job, jobParameters);
33     }
34 }
```

# Job, JobInstance, JobExecution

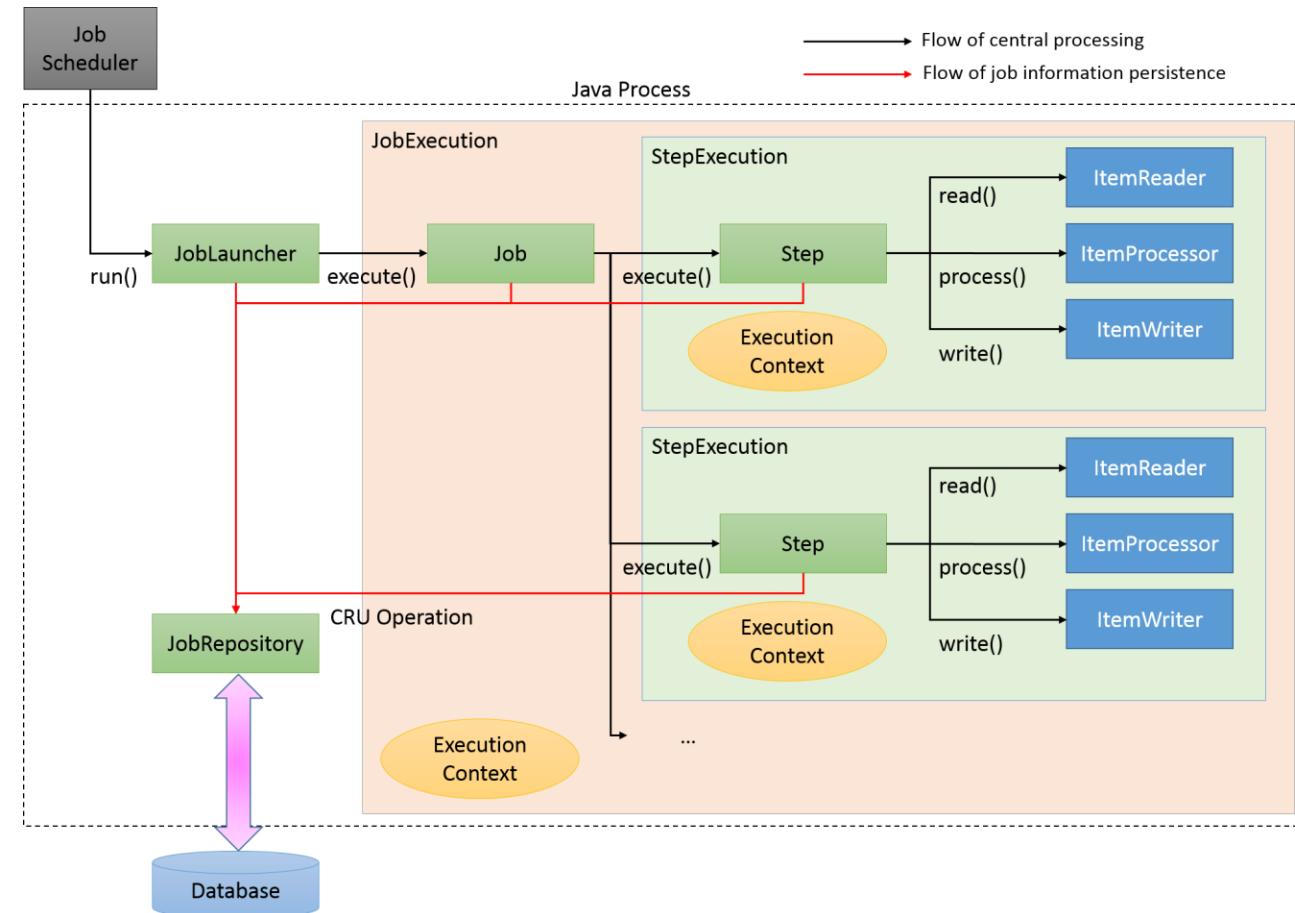
- Ein Job ist ein Flow von Steps, die durchlaufen werden.
- Beispiel: Ein EndOfDay Job wird definiert und soll täglich ausgeführt werden.
- Jeden Tag wird eine JobInstance von EndOfDay Job erstellt.
- Wenn die JobInstance ausgeführt wird, wird ein JobExecution erstellt.
- Wenn die JobExecution nicht erfolgreich abgeschlossen ist, können weitere zur selben JobInstance erstellt werden, bis einer erfolgreich war.
- Wenn ein JobExecution erfolgreich ist, kann die dazugehörige JobInstance nicht erneut gestartet werden.



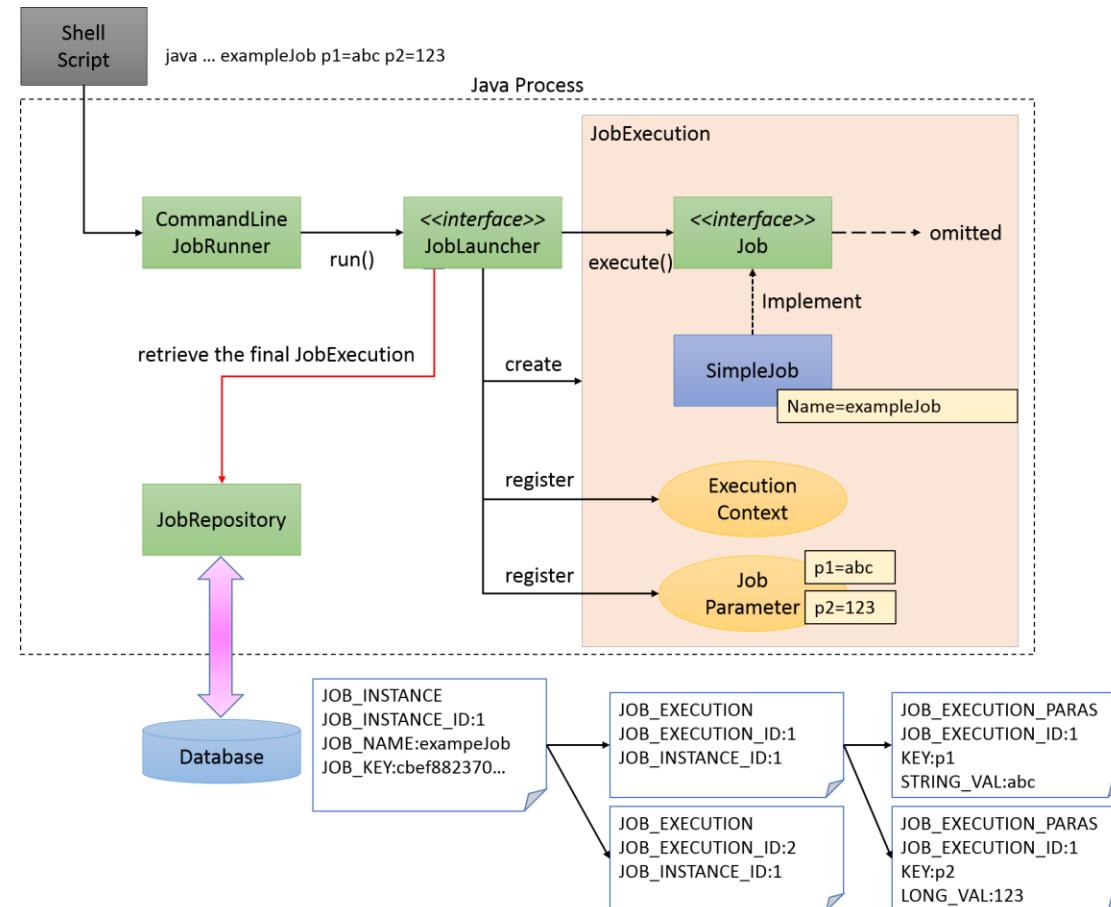
# Spring Batch



# Ablauf Jobausführung

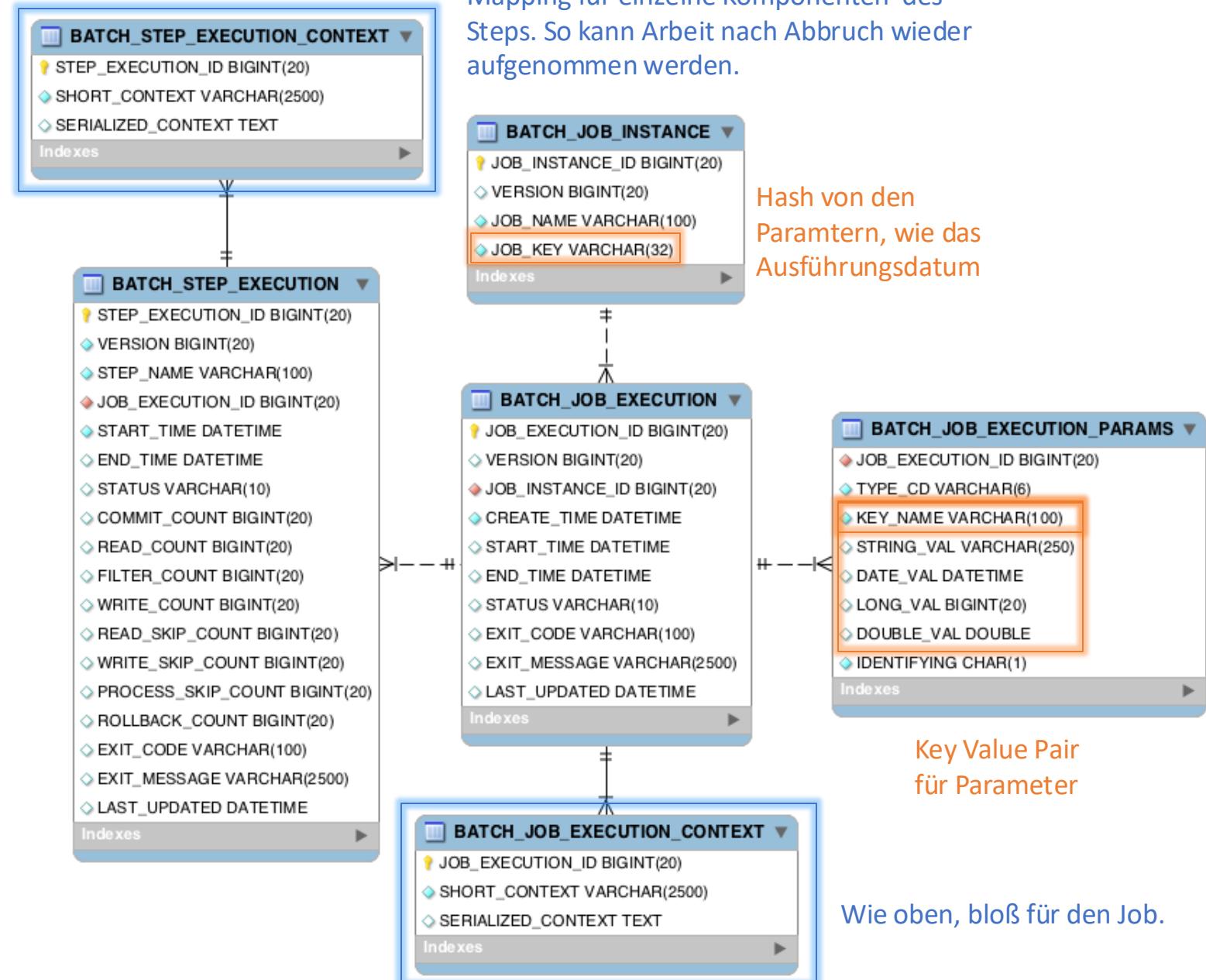


# Datenbankbefüllung



# JobRepository

- Beim JobRepository liegt dieses Schema vor.
- Hier werden Informationen über die ausgeführten Jobs und Steps gespeichert.



# Status und Exit Code

- Das Feld „STATUS“ in `BATCH_JOB_EXECUTION` wird mit dem vordefinierten Enum aus `BatchStatus` gefüllt
- `EXIT_CODE` und `EXIT_MESSAGE` können vom Nutzer konfiguriert werden.
- Default: Exceptions werden in `EXIT_MESSAGE` geschrieben.

```
Enumeration representing the status of an execution.  
Author: Lucas Ward, Dave Syer, Michael Minella, Mahmoud Ben Hassine  
  
public enum BatchStatus {  
  
    >     /** The order of the status values is significant because it can be  
        |     The batch job has successfully completed its execution.  
        |     COMPLETED,  
        |     Status of a batch job prior to its execution.  
        |     STARTING,  
        |     Status of a batch job that is running.  
        |     STARTED,  
        |     Status of batch job waiting for a step to complete before stopping the batch job.  
        |     STOPPING,  
        |     Status of a batch job that has been stopped by request.  
        |     STOPPED,  
        |     Status of a batch job that has failed during its execution.  
        |     FAILED,  
        |     Status of a batch job that did not stop properly and can not be restarted.  
        |     ABANDONED,  
        |     Status of a batch job that is in an uncertain state.  
        UNKNOWN;
```

# SoutStep erstellen

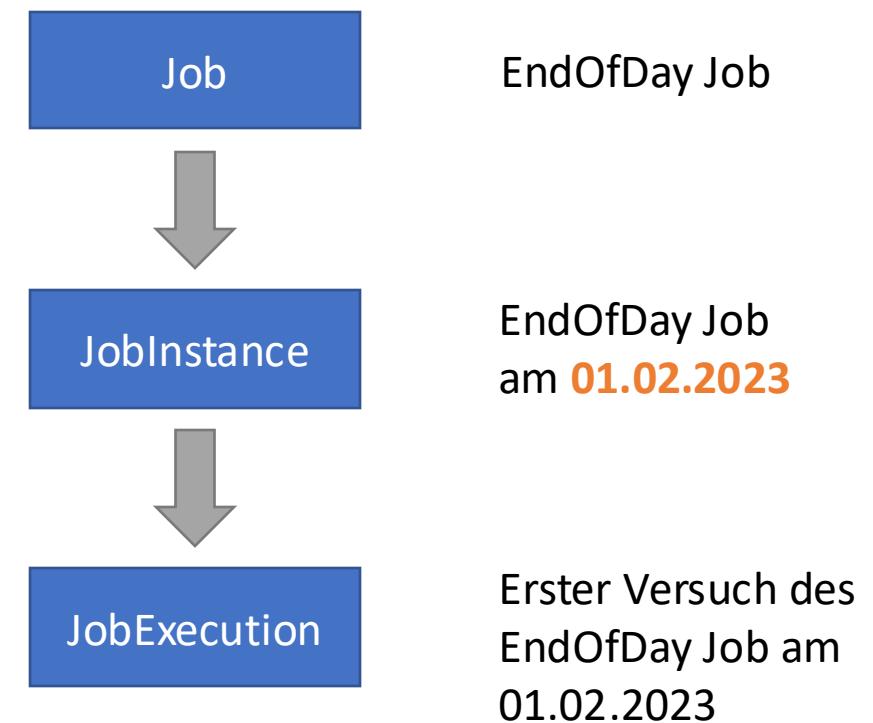
- Wir bauen uns eine Klasse, um im Rahmen der Schulung leicht Tasklets zu erstellen.

```
28     @Autowired
29     private SoutBuilder soutBuilder;
30
31     @Bean
32     Job firstJob() {
33         return soutBuilder
34             .setJobName("MeinJobName")
35             .setStepName("MeinStepName")
36             .setMessage("Hey, du da!")
37             .getJob();
38 }
```

```
17     @Component
18     @Setter
19     @Accessors(chain = true)
20     public class SoutBuilder {
21
22         @Autowired
23         private JobRepository jobRepository;
24         @Autowired
25         private PlatformTransactionManager transactionManager;
26
27         @Value("Hello World")
28         private String message;
29         @Value("soutStep")
30         private String stepName;
31         @Value("soutJob")
32         private String jobName;
33
34         public Tasklet getTasklet() {
35             String message = this.message;
36             return (contribution, chunkContext) -> {
37                 System.out.println(message);
38                 return RepeatStatus.FINISHED;
39             };
40         }
41
42         public TaskletStep getTaskletStep() {
43             String stepName = this.stepName;
44             return new StepBuilder(stepName, jobRepository)
45                 .tasklet(getTasklet(), transactionManager).build();
46         }
47
48         public Job getJob() {
49             String jobName = this.jobName;
50             return new JobBuilder(jobName, jobRepository)
51                 .start(getTaskletStep()).build();
52         }
53     }
```

# Job Parameter

- Über Parameter werden Jobinstanzen eindeutig gemacht.
- Die Parameter werden gehasht als Key gespeichert.
- Default: Sobald eine Jobinstanz eine erfolgreiche JobExecution aufweist, kann diese keine neue JobExecution mehr starten.



# No more JobBuilderFactory

- JobBuilderFactory wurde mit 5.0.0 auf deprecated gesetzt.
- Es war eigentlich nur ein Jobbuilder, bei dem das Jobrepository automatisch gesetzt wurde (was man nicht immer möchte).
- Zukünfig soll daher einfach Jobbuilder verwendet werden.

```
// Sample with v4
@Configuration
@EnableBatchProcessing
public class MyJobConfig {

    @Autowired
    private JobBuilderFactory jobBuilderFactory;

    @Bean
    public Job myJob(Step step) {
        return this.jobBuilderFactory.get("myJob")
            .start(step)
            .build();
    }

}
```

```
// Sample with v5
@Configuration
@EnableBatchProcessing
public class MyJobConfig {

    @Bean
    public Job myJob(JobRepository jobRepository, Step step) {
        return new JobBuilder("myJob", jobRepository)
            .start(step)
            .build();
    }

}
```

# Steps hintereinander ausführen

```
45 @Bean
46     @Primary
47     Job chainedSteps() {
48         TaskletStep step1 = soutBuilder.setMessage("Message 1").setStepName("Step 1").getTaskletStep();
49         TaskletStep step2 = soutBuilder.setMessage("Message 2").setStepName("Step 2").getTaskletStep();
50         TaskletStep step3 = soutBuilder.setMessage("Message 3").setStepName("Step 3").getTaskletStep();
51
52         return new JobBuilder(name: "chainedJob", jobRepository)
53             .start(step1)
54             .next(step2)
55             .next(step3)
56             .build();
57     }
58 }
```

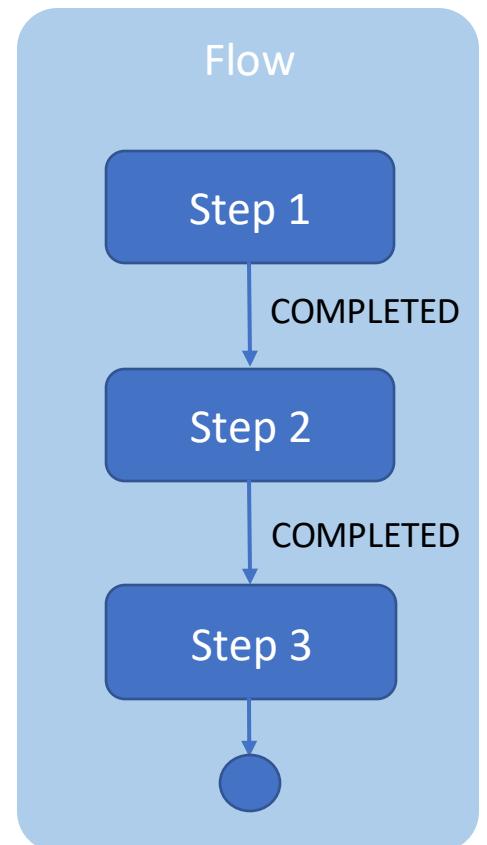
Step 1

Step 2

Step 3

# Implizit Flow erzeugen

```
59  @Bean
60  @Primary
61  Job chainedStepsWithStatus() {
62      TaskletStep step1 = soutBuilder.setMessage("Message 1").setStepName("Step 1").getTaskletStep();
63      TaskletStep step2 = soutBuilder.setMessage("Message 2").setStepName("Step 2").getTaskletStep();
64      TaskletStep step3 = soutBuilder.setMessage("Message 3").setStepName("Step 3").getTaskletStep();
65
66      return new JobBuilder(name: "chainedJob", jobRepository)
67          .start(step1) SimpleJobBuilder
68          .on(pattern: "COMPLETED").to(step2) FlowBuilder<FlowJobBuilder>
69          .from(step2).on(pattern: "COMPLETED").to(step3)
70          .end() FlowJobBuilder
71          .build();
72 }
```



# Flow

- Flows sind Zusammenfassung von Steps
- Flows bestehen aus Steps und Flows.
- Stelle mit einem FlowBuilder<Flow> einen Flow zusammen.
- Ein Flow muss mit end() explizit beendet werden.

# Abschlusspunkte

Methode	Parameter	Bedeutung	Job kann mit selben Parametern erneut gestartet werden?
end	-	Job endet erfolgreich.	Nein
stop	-	Job stoppt und wird bei Neustart wieder von Beginn aufgenommen.	Ja
fail	-	Job endet als erfolglos.	Ja
stopAndRestart	Flow Step JobExecutionDecider	Job stoppt und wir an der vorgegebene Stelle wieder aufgenommen.	Ja

```
return new JobBuilder( name: "chainedJob", jobRepository)
    .start(step1) SimpleJobBuilder
    .on( pattern: "COMPLETED").to(step2) FlowBuilder<FlowJobBuilder>
    .from(step2).on( pattern: "COMPLETED").stopAndRestart(step3)
    .end() FlowJobBuilder
    .build();
```

# Flow→Step & Step→Flow

- Von einem Flow aus kann der nächste Step direkt angegeben werden.
- Von einem Step aus, kann nur mit on() zum nächsten Stepp gelangt werden.

```
79     Flow createSmallFlow() {
80         TaskletStep step1 = soutBuilder.setMessage("Message 1").setStepName("Step 1").getTaskletStep();
81         TaskletStep step2 = soutBuilder.setMessage("Message 2").setStepName("Step 2").getTaskletStep();
82         TaskletStep step3 = soutBuilder.setMessage("Message 3").setStepName("Step 3").getTaskletStep();
83         return new FlowBuilder<Flow>( name: "3 Part Flow").start(step1).next(step2).next(step3).end();
84     }
85
86     @Bean
87     //@Primary
88     Job flowFirstJob() {
89         TaskletStep step0 = soutBuilder.setMessage("Message 0").setStepName("Step 0").getTaskletStep();
90         Flow flow = createSmallFlow();
91         return new JobBuilder( name: "flowFirstJob", jobRepository)
92             .start(flow).next(step0)
93             .end().build();
94     }
95
96     @Bean
97     @Primary
98     Job flowLastJob() {
99         TaskletStep step0 = soutBuilder.setMessage("Message 0").setStepName("Step 0").getTaskletStep();
100        Flow flow = createSmallFlow();
101        return new JobBuilder( name: "flowFirstJob", jobRepository)
102            .start(step0).on( pattern: "COMPLETED").to(flow)
103            .end().build();
104    }
```

# Parallelisierung von Flows

- Flows können mit split() parallelisiert werden.

```
107     @Bean
108     @Primary
109     Job parallelFlows() {
110         TaskletStep stepA1 = soutBuilder.setMessage("Message A1").setStepName("Step A1").getTaskletStep();
111         TaskletStep stepA2 = soutBuilder.setMessage("Message A2").setStepName("Step A2").getTaskletStep();
112         TaskletStep stepA3 = soutBuilder.setMessage("Message A3").setStepName("Step A3").getTaskletStep();
113         Flow flowA = new FlowBuilder<Flow>( name: "3 Part A Flow").start(stepA1).next(stepA2).next(stepA3).end();
114
115
116         TaskletStep stepB1 = soutBuilder.setMessage("Message B1").setStepName("Step B1").getTaskletStep();
117         TaskletStep stepB2 = soutBuilder.setMessage("Message B2").setStepName("Step B2").getTaskletStep();
118         TaskletStep stepB3 = soutBuilder.setMessage("Message B3").setStepName("Step B3").getTaskletStep();
119         Flow flowB = new FlowBuilder<Flow>( name: "3 Part B Flow").start(stepB1).next(stepB2).next(stepB3).end();
120
121         return new JobBuilder( name: "parallelFlows", jobRepository)
122             .start(flowA).split(new SimpleAsyncTaskExecutor()).add(flowB).end().build();
123     }
124 }
```

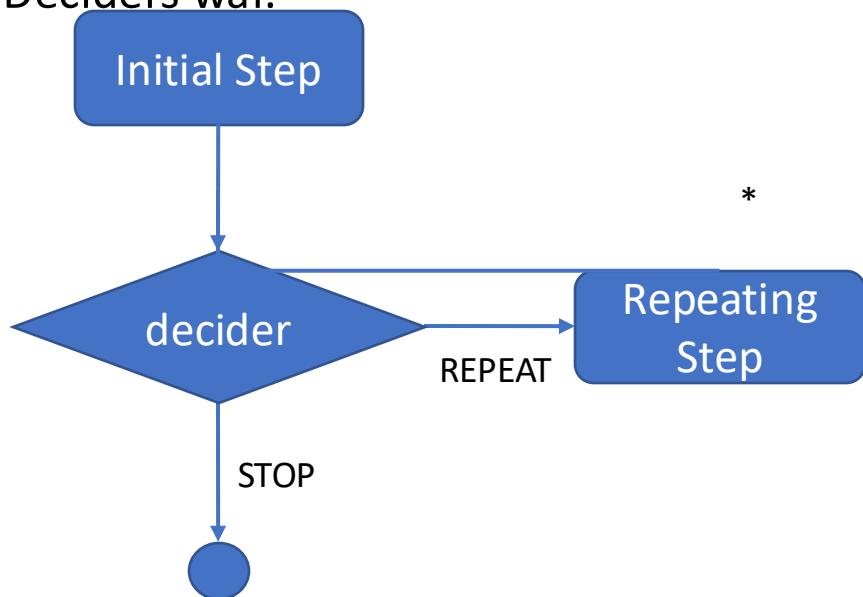
# Parallelisierung von Flows

- Tasklet anpassen, um Thread zu sehen:

```
33     private Boolean showThread = false;
34
35     public Tasklet getTasklet() {
36         String message = this.message;
37         return (contribution, chunkContext) -> {
38             System.out.println(message +
39                 (showThread ? " "+Thread.currentThread().getName() : ""));
40             return RepeatStatus.FINISHED;
41         };
42     }
```

# Entscheidungen treffen

- Mit einem JobExecutionDecider kann eine Auswahl getroffen werden, welcher Flow/Step als nächstes durchgeführt werden soll.
- Decider sind kein voller Step. Es wird nicht gespeichert, was das Ergebnis eines Deciders war.



```
9  public class RepeatDecider implements JobExecutionDecider {  
10     public static final String REPEAT = "REPEAT";  
11     public static final String STOP = "STOP";  
12     private int count = 0;  
13     private final int max;  
14  
15     public RepeatDecider(int max) {  
16         this.max = max;  
17     }  
18  
19     @Override  
20     public FlowExecutionStatus decide(JobExecution jobExecution, StepExecution stepExecution) {  
21         return new FlowExecutionStatus(count++ < max ? REPEAT : STOP);  
22     }  
23 }
```

```
125 @Bean  
126 @Primary  
127 Job deciderJob() {  
128     TaskletStep step1 = soutBuilder.setMessage("Initial Step").setStepName("Step I").getTaskletStep();  
129     TaskletStep step2 = soutBuilder.setMessage("Repeating Step").setStepName("Step R").getTaskletStep();  
130  
131     RepeatDecider repeatDecider = new RepeatDecider( max: 3);  
132  
133     return new JobBuilder( name: "deciderJob", jobRepository)  
134         .start(step1) SimpleJobBuilder  
135         .next(repeatDecider) JobFlowBuilder  
136         .from(repeatDecider).on(RepeatDecider.REPEAT).to(step2) FlowBuilder<FlowJobBuilder>  
137         .from(repeatDecider).on(RepeatDecider.STOP).end()  
138         .from(step2).on( pattern: "*").to(repeatDecider).end() FlowJobBuilder  
139     }  
140 }
```

# Nested Job

- Wir können Jobs in einen Step verpacken.
- Schlägt der innere Job fehl, wird dies auch für den äußeren gespeichert.

```
146 @Bean
147 @Primary
148 Job nestedJob(JobLauncher jobLauncher) {
149     TaskletStep step0 = soutBuilder.setMessage("First Step").setStepName("Step F").getTaskletStep();
150     TaskletStep step1 = soutBuilder.setMessage("Nested Step").setStepName("Step N").getTaskletStep();
151     TaskletStep step2 = soutBuilder.setMessage("Last Step").setStepName("Step L").getTaskletStep();
152
153     Job nestedJob = new JobBuilder( name: "Nested Job", jobRepository).start(step1).build();
154
155     Step nestedJobStep = new JobStepBuilder(new StepBuilder( name: "nestedJobStep", jobRepository))
156         .job(nestedJob)
157         .launcher(jobLauncher)
158         .repository(jobRepository)
159         .build();
160
161     return new JobBuilder( name: "outer Job", jobRepository)
162         .start(step0).next(nestedJobStep).next(step2).build();
163 }
164 }
```

# Listeners

- JobexecutionListener
- StepexecutionListener
- ChunkListener
- ItemReadListener
- ItemProcessListener
- ItemWriteListener
- Listener können über Interfaces implementiert werden oder über Annotationen.

# Listener implementieren

```
165 @Bean
166 @Primary
167 Job jobWithListener() {
168
169     JobExecutionListener listener = new JobExecutionListener() {
170         @Override
171         public void beforeJob(JobExecution jobExecution) {
172             System.out.println("LET'S DO IT, "
173                         + jobExecution.getJobInstance().getJobName());
174         }
175
176         @Override
177         public void afterJob(JobExecution jobExecution) {
178             System.out.println("YOU'VE DONE IT, "
179                         + jobExecution.getJobInstance().getJobName());
180         }
181     };
182
183     Flow flow = createSmallFlow();
184
185     return new JobBuilder( name: "very verbose Job", jobRepository)
186         .listener(listener) JobBuilder
187         .start(flow) JobFlowBuilder
188         .end().build();
189 }
```

# Annotationen verwenden

```
7  public class VerboseJobListener {
8      @BeforeJob
9      public void beforeJob(JobExecution jobExecution) {
10          System.out.println("LET'S DO IT, "
11                  + jobExecution.getJobInstance().getJobName());
12      }
13
14      @AfterJob
15      public void afterJob(JobExecution jobExecution) {
16          System.out.println("YOU'VE DONE IT, "
17                  + jobExecution.getJobInstance().getJobName());
18      }
19  }
```

```
190 @
191
192
193
194
195
196
197
198
    @Bean
    @Primary
    Job jobWithListener2() {
        return new JobBuilder( name: "very verbose Job", jobRepository)
            .listener(new VerboseJobListener()) JobBuilder
            .start(createSmallFlow()) JobFlowBuilder
            .end().build();
    }
}
```

```
9  public class MyChunkListener {  
10     @BeforeChunk  
11     @ public void beforeChunnk(ChunkContext chunkContext) {  
12         System.out.println(">> Before Chunk. Is Complete:" + chunkContext.isComplete());  
13     }  
14  
15     @AfterChunk  
16     @ public void afterChung(ChunkContext chunkContext) {  
17         System.out.println("<< After Chunk. Is Complete:" + chunkContext.isComplete());  
18     }  
19 }
```

```
6  public class MyJobListener implements JobExecutionListener {  
7      @Override  
8      @ public void beforeJob(JobExecution jobExecution) {  
9          System.out.println("Job " + jobExecution.getJobInstance().getJobName() + " started.");  
10     }  
11  
12     @Override  
13     @ public void afterJob(JobExecution jobExecution) {  
14         System.out.println("Job " + jobExecution.getJobInstance().getJobName() + " ended.");  
15     }  
16 }
```

```
31     ▼  public Job createJob(List<String> list) {  
32         ListItemReader<String> reader = new ListItemReader<>(list);  
33         ItemWriter<String> writer = chunk -> chunk.forEach(System.out::println);  
34  
35         TaskletStep step = new StepBuilder( name: "ReadAndWriteStep", jobRepository)  
36             .<String, String>chunk( chunkSize: 2, transactionManager) SimpleStepBuilder<String, String>  
37             .faultTolerant() FaultTolerantStepBuilder<String, String>  
38             .listener(new MyChunkListener())  
39             .reader(reader) SimpleStepBuilder<String, String>  
40             .writer(writer)  
41             .build();  
42  
43  
44         return new JobBuilder( name: "IterableToSout", jobRepository)  
45             .start(step)  
46             .listener(new MyJobListener())  
47             .build();  
48     }
```

faultTolerant() ist nötig,  
wenn der Listener auf  
Annotationen basiert.

```
110 ↵ @  
111 ⇢ @  
112     public Job listenerJob(ListToSoutJobFactory listToSoutJobFactory) {  
113         return listToSoutJobFactory.createJob(List.of("a", "b", "c", "d", "e"));  
114     }
```

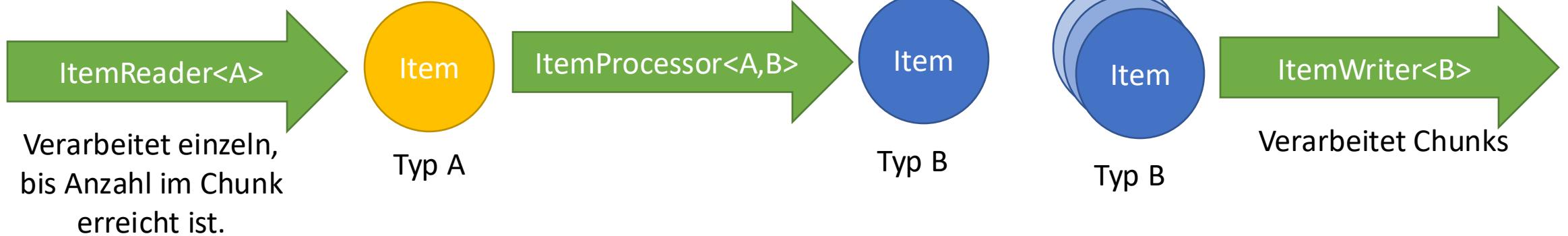
# ItemReader, ItemProcessor und ItemWriter

- Jeder Step kann genau einen Reader, Processor und Writer haben.

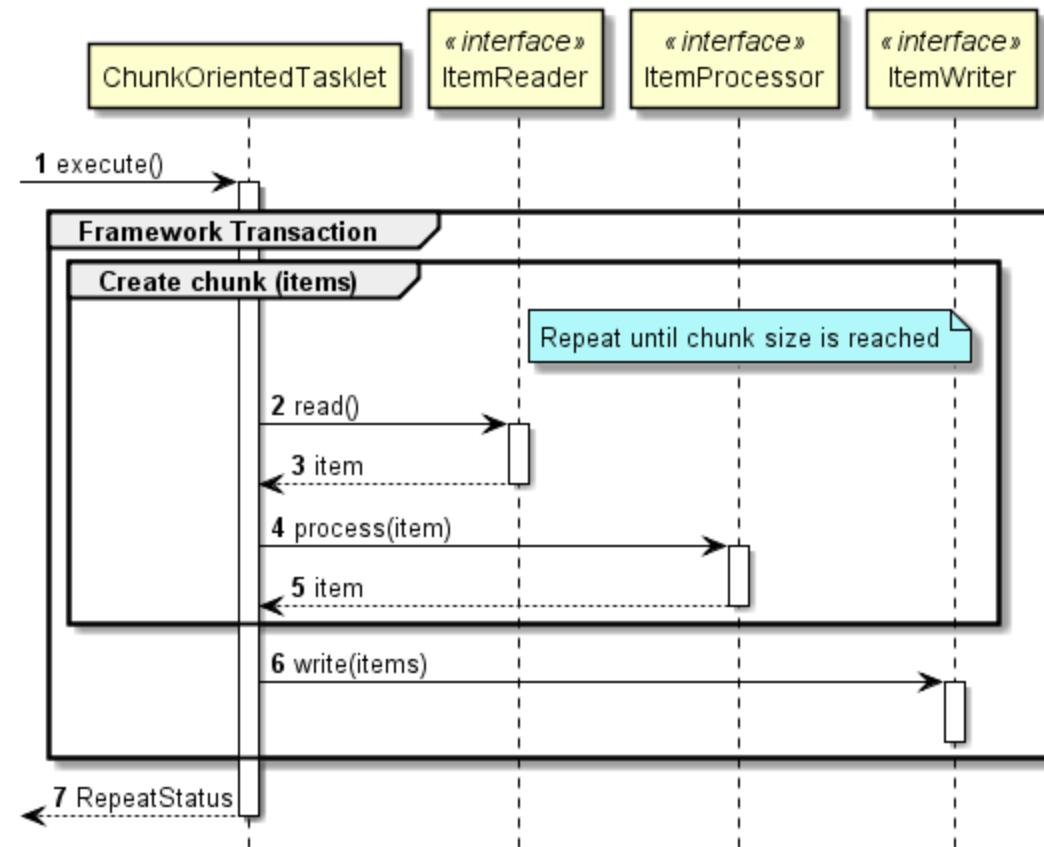
```
10  @FunctionalInterface  
11  public interface ItemReader<T> {  
12      @Nullable  
13  T read() throws Exception, Un  
14 }
```

```
11  @FunctionalInterface  
12  public interface ItemProcessor<I, O> {  
13      @Nullable  
14  O process(@NonNull I var1) throws Exception;  
15 }
```

```
10  @FunctionalInterface  
11  public interface ItemWriter<T> {  
12  void write(@NonNull Chunk<? extends T> var1) throws Exception;  
13 }
```



# Ablauf



# Beispiel für ItemReader

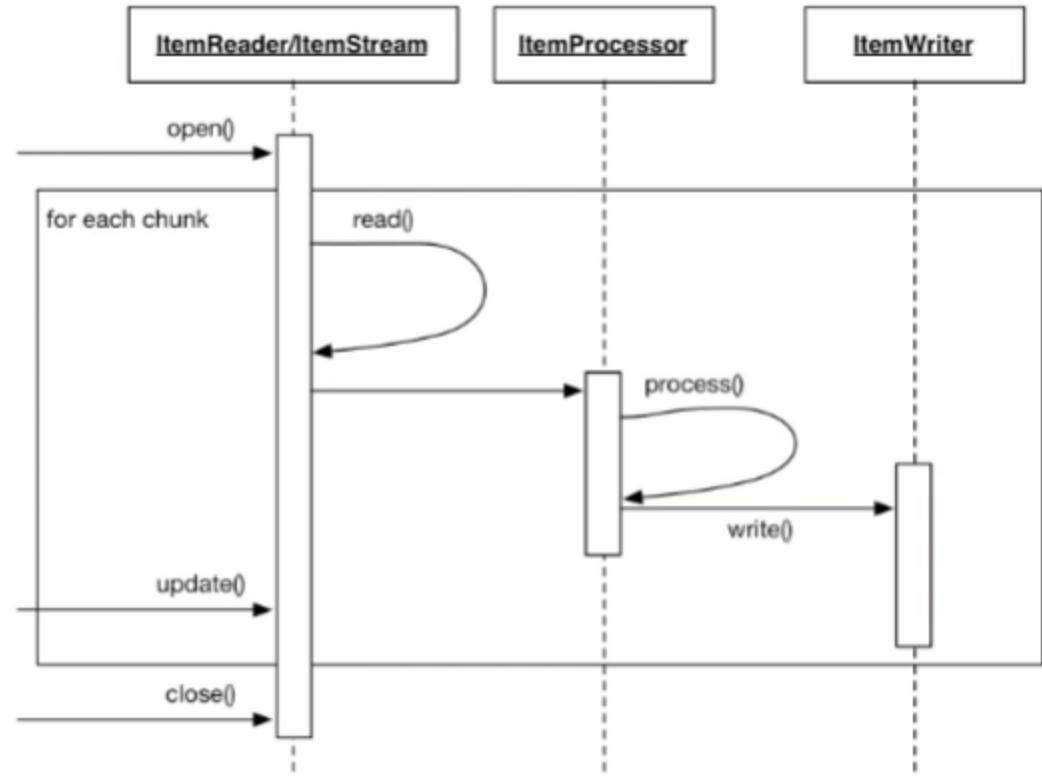
```
8  import java.util.Iterator;
9
10 public class IterableReader<T> implements ItemReader<T> {
11     private final Iterator<T> iterator;
12
13     public IterableReader(Iterator<T> iterator) {
14         this.iterator = iterator;
15     }
16
17     @Override
18     public IterableReader(Iterable<T> iterable) {
19         this.iterator = iterable.iterator();
20     }
21
22     public T read() throws Exception,
23             UnexpectedInputException, ParseException, NonTransientResourceException {
24         return iterator.hasNext() ? iterator.next() : null;
25     }
26 }
```

# Reader Writer Processor Example

```
203     @Bean
204     @Primary
205     Job iterableReaderJob() {
206         List<String> list = List.of("a", "b", "c", "d", "e", "f", "g");
207
208         TaskletStep listReadAndSoutStep = new StepBuilder( name: "ListReadAndSoutStep", jobRepository)
209             .<String, String>chunk( chunkSize: 3, transactionManager)
210             .reader(new IterableReader<String>(list))
211             .processor(String::toUpperCase)
212             .writer(chunk -> chunk.forEach(System.out::println))
213             .build();
214
215         return new JobBuilder( name: "ListReadAndSoutJob", jobRepository)
216             .start(listReadAndSoutStep)
217             .build();
218     }
219 }
```

# ItemStream

- Im ExecutionContext sind Informationen über den Step in Form einer Map gespeichert.
- Update wird nach jedem erfolgreichen write() eines Chunks durchgeführt.
- Prüfe immer, ob Reader oder Writer statefull sind!



Informationen für den Step

BATCH\_STEP\_EXECUTION\_CONTEXT

STEP_EXECUTION_ID BIGINT(20)
SHORT_CONTEXT VARCHAR(2500)
SERIALIZED_CONTEXT TEXT

Indexes

Informationen für den Job

BATCH\_JOB\_EXECUTION\_CONTEXT

JOB_EXECUTION_ID BIGINT(20)
SHORT_CONTEXT VARCHAR(2500)
SERIALIZED_CONTEXT TEXT

Indexes

```
6 package org.springframework.batch.item;
7
8 public interface ItemStream {
9     default void open(ExecutionContext executionContext) throws ItemStreamException {
10 }
11
12     default void update(ExecutionContext executionContext) throws ItemStreamException {
13 }
14
15     default void close() throws ItemStreamException {
16 }
17 }
```

```
7  public class StateFullListReader<T> implements ItemStreamReader<T> {
8      private static String CURRENT_INDEX_KEY = "currentIndex";
9
10     private List<T> list;
11     private int currentIndex;
12     private long delayTimeInMillis;
13
14     private boolean restarted = false;
15
16     public StateFullListReader(List<T> list, long delayTimeInMillis) {
17         this.list = list;
18         this.delayTimeInMillis = delayTimeInMillis;
19     }
20
21     @Override
22     public T read() throws Exception, UnexpectedInputException, ParseException, NonTransientResourceException {
23         Thread.sleep(delayTimeInMillis);
24         if(currentIndex == 7 && !restarted) throw new RuntimeException("TOO MUCH!");
25         if (currentIndex < list.size()) return list.get(currentIndex++);
26         return null;
27     }
28
29     @Override
30     public void open(ExecutionContext executionContext) throws ItemStreamException {
31         restarted = executionContext.containsKey(CURRENT_INDEX_KEY);
32         currentIndex = executionContext.getInt(CURRENT_INDEX_KEY, defaultInt: 0);
33         System.out.println("Opened at: " + currentIndex);
34     }
35
36     @Override
37     public void update(ExecutionContext executionContext) throws ItemStreamException {
38
39         executionContext.putInt(CURRENT_INDEX_KEY, currentIndex);
40         System.out.println("Saved at: " + currentIndex);
41
42     }
43
44 }
```

# Automatisches Jobstarten - @Scheduled

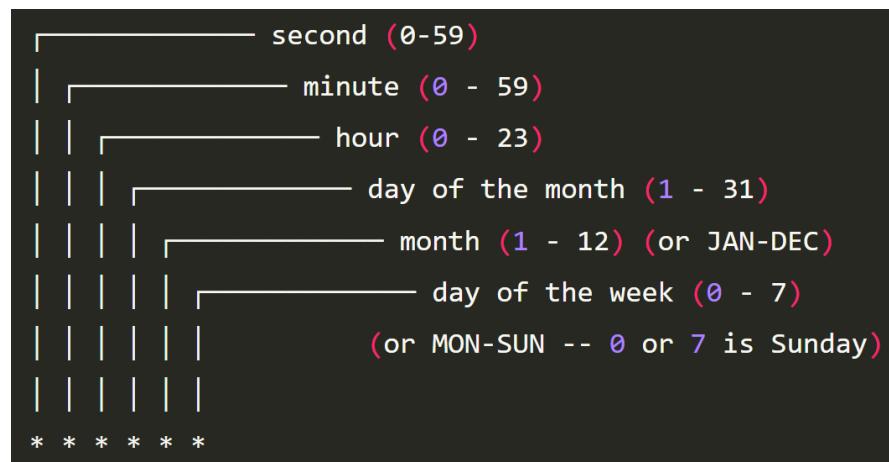
- Mit der @Scheduled Annotation können automatisch Funktionen gestartet werden.
- Mehrere @Scheduled können mit einem @Schedules zusammengefasst werden.
- Aktiviere mit @EnableScheduling

```
9  @SpringBootApplication
10 @EnableScheduling
11 public class JobSchedulingApplication {
12
13     public static void main(String[] args) {
14         SpringApplication.run(JobSchedulingApplication.class, args);
15     }
16 }
```

```
7  @Component
8  public class Counter {
9
10    private int i = 0;
11
12    @Scheduled(fixedRate = 1000)
13    public void count() {
14        System.out.println(i++);
15    }
16 }
```

# @Scheduled(cron=...)

- Zeigt an, zu welchen Zeiten ein Job ausgeführt wird.
- @Scheduled(cron = „0 10 15 \* \* \*“) Bedeutet, dass der Job jeden Tag um 15:10 Uhr durchgeführt wird.
- Alternativ können Macros verwendet werden: @yearly, @monthly, @weekly, @dayly, @hourly (= 0 0 \* \* \* \*)
- Was bedeutet @Scheduled(cron=„\* 10-15 20 \* \* \*“)



## @Scheduled(fixedDelay=...)

- Job wird gestartet, nachdem der letzte Job die angegebene Zeit beendet ist.
- Mit dem Parameter initialDelay kann eine initiale Wartezeit hinzugefügt werden.

# @Scheduled(fixedRate=...)

- Nächster Start ist unabhängig vom Ende des letzten.
- Methoden müssen zusätzlich mit @Async annotiert werden.
- Es muss @EnableAsync gesetzt werden.

```
● ⚡ @SpringBootApplication
  11   @EnableScheduling
  12   @EnableAsync
  13 ➤ public class JobSchedulingApplication {
```

```
19      @Scheduled(fixedRate = 1000)
20
21  ➤     @Async
22
23     public void multiple() throws InterruptedException {
24
25         int j = i++;
26
27         String name = Thread.currentThread().getName();
28         System.out.println("start:" + j + " Threadname:" + name);
29         Thread.currentThread().sleep( millis: 1500 );
30
31         System.out.println("end:" + j + " Threadname:" + name);
```

# CSV lesen

```
400 @Bean
401 @Primary
402 Job csvReadJob() {
403
404     FlatFileItemReader<Student> reader = new FlatFileItemReader<>();
405     reader.setResource(resource);
406     reader.setLinesToSkip(0);
407     reader.setLineMapper(new DefaultLineMapper<>(){
408         setLineTokenizer(new DelimitedLineTokenizer(){
409             setNames("id", "firstName", "lastName", "email");
410         });
411         setFieldSetMapper(new BeanWrapperFieldSetMapper<>(){
412             setTargetType(Student.class);
413         });
414     });
415
416
417
418     ItemWriter<Student> writer = chunk -> chunk.forEach(System.out::println);
419
420     TaskletStep writeStep = new StepBuilder("WriteStep", jobRepository)
421         .<Student, Student>chunk(chunkSize: 15, transactionManager)
422         .reader(reader)
423         .writer(writer)
424         .build();
425
426
427     return new JobBuilder("WriteRepoJob", jobRepository)
428         .start(writeStep).build();
429     }
430
431 }
```

# CSV schreiben

```
369     private Resource resource = new FileSystemResource( path: "output/outputData.csv");
370
371     @Bean
372     //  @Primary
373     @Job
374     Job csvWriteJob(StudentRepository studentRepository,
375                      FakeMachine fakeMachine) {
376
377         Iterable<Student> students = fakeMachine.fakeStudents( n: 100);
378
379         IterableReader<Student> reader = new IterableReader<>(students);
380
381         FlatFileItemWriter<Student> writer = new FlatFileItemWriter<>();
382         writer.setAppendAllowed(true);
383         writer.setResource((WritableResource) resource);
384         writer.setLineAggregator(new DelimitedLineAggregator<Student>() {{
385             setDelimiter(",");
386             setFieldExtractor(new BeanWrapperFieldExtractor<>() {{
387                 setNames(new String[]{"id", "firstName", "lastName", "email"});
388             }});
389         }});
390
391         TaskletStep writeStep = new StepBuilder( name: "WriteStep", jobRepository)
392             .<Student, Student>chunk( chunkSize: 15, transactionManager)
393             .reader(reader)
394             .writer(writer)
395             .build();
396
397         return new JobBuilder( name: "WriteRepoJob", jobRepository)
398             .start(writeStep).build();
399     }
```

# Mehr Reader, Writer, Processor

- <https://docs.spring.io/spring-batch/docs/current/reference/html/appendix.html#listOfReadersAndWriters>

# Monitoring



- Spring Batch liefert alle Metriken automatisch an [Micrometer](#).
- [Link](#)

Metric Name	Type	Description	Tags
spring.batch.job	TIMER	Duration of job execution	name, status
spring.batch.job.active	LONG_TASK_TIMER	Currently active jobs	name
spring.batch.step	TIMER	Duration of step execution	name, job.name, status
spring.batch.step.active	LONG_TASK_TIMER	Currently active step	name
spring.batch.item.read	TIMER	Duration of item reading	job.name, step.name, status
spring.batch.item.process	TIMER	Duration of item processing	job.name, step.name, status
spring.batch.chunk.write	TIMER	Duration of chunk writing	job.name, step.name, status

# Repository Reader

```
278 @Bean  
279 @Primary  
280 Job databaseReadJob(StudentRepository studentRepository) {  
281     RepositoryItemReader<Student> reader = new RepositoryItemReader<>();  
282     reader.setRepository(studentRepository);  
283     reader.setMethodName("findAll");  
284     reader.setSort(Map.of("id", Sort.Direction.ASC));  
285     reader.setPageSize(10);  
286  
287     TaskletStep step = new StepBuilder("printStudentsStep", jobRepository)  
288         .<Student, Student>chunk(chunkSize: 10, transactionManager)  
289         .reader(reader)  
290         .writer(chunk -> chunk.forEach(System.out::println)).build();  
291  
292     return new JobBuilder("printStudentsJob", jobRepository)  
293         .start(step).build();  
294 }
```

Pflichtfeld

```
27     FakeMachine fakeMachine = context.getBean(FakeMachine.class);  
28     StudentRepository studentRepository = context.getBean(StudentRepository.class);  
29     studentRepository.saveAll(fakeMachine.fakeStudents(n: 100));
```

# RepositoryWriter

```
296 @Bean  
297     @Primary  
298     Job DataBaseWriteJob(StudentRepository studentRepository,  
299                 FakeMachine fakeMachine) {  
300         Iterable<Student> students = fakeMachine.fakeStudents( n: 100);  
301  
302         IterableReader<Student> reader = new IterableReader<>(students);  
303  
304         RepositoryItemWriter<Student> writer = new RepositoryItemWriter<>();  
305         writer.setRepository(studentRepository); Pflichtfeld  
306  
307         TaskletStep writeStep = new StepBuilder( name: "WriteStep", jobRepository)  
308             .<Student, Student>chunk( chunkSize: 15, transactionManager)  
309             .reader(reader)  
310             .writer(writer)  
311             .build();  
312  
313         return new JobBuilder( name: "WriteRepoJob", jobRepository)  
314             .start(writeStep).build();  
315     }
```

# Items aussortieren mit ItemProcessor

```
@Bean
@Primary
Job databaseReadJob(StudentRepository studentRepository) {
    RepositoryItemReader<Student> reader = new RepositoryItemReader<>();
    reader.setRepository(studentRepository);
    reader.setMethodName("findAll");
    reader.setSort(Map.of( k1: "id", Sort.Direction.ASC));
    reader.setPageSize(10);

    ItemProcessor<Student, Student> processor = new ItemProcessor<>() {
        @Override
        public Student process(Student item) throws Exception {
            return item.getFirstName().length() % 2 == 0 ? item : null;
        }
    };

    TaskletStep step = new StepBuilder( name: "printStudentsStep", jobRepository)
        .<Student, Student>chunk( chunkSize: 10, transactionManager)
        .reader(reader)
        .processor(processor)
        .writer(chunk -> chunk.forEach(System.out::println)).build();

    return new JobBuilder( name: "printStudentsJob", jobRepository)
        .start(step).build();
}
```

Gebe das Item  
zurück oder null,  
um es auszufiltern.

# Validierung Items

- Ggf. möchte man eine Validierung von Items vornehmen und Exceptions werfen oder herausfiltern, wenn es Fehler gibt.

```
291      Validator<Student> validator = new Validator<>() {
292
293     @Override
294     public void validate(Student value) throws ValidationException {
295         if (value.getFirstName().length() > 4) throw new ValidationException("Name Too Long");
296     }
297 }
298
299
300     ValidatingItemProcessor<Student> processor = new ValidatingItemProcessor<>(validator);
301
302     processor.setFilter(true); true: kein Abbruch des Jobs.
303
304     TaskletStep step = new StepBuilder( name: "printStudentsStep", jobRepository)
305         .<Student, Student>chunk( chunkSize: 10, transactionManager)
306         .reader(reader)
307         .processor(processor)
308         .writer(chunk -> chunk.forEach(System.out::println)).build();
```

# CompositeItemProcessors

- Es können mehrere ItemProcessor hintereinander geschaltet werden.

```
299     ValidatingItemProcessor<Student> validatingItemProcessor = new ValidatingItemProcessor<>(validator);
300     validatingItemProcessor.setFilter(true);
301
302     ItemProcessor<Student, Student> nameOptimizer = item -> item.setFirstName(item.getFirstName() + "eth");
303
304     CompositeItemProcessor<Student, Student> compositeItemProcessor = new CompositeItemProcessor<>(
305         List.of(validatingItemProcessor, nameOptimizer));
306
307     TaskletStep step = new StepBuilder( name: "printStudentsStep", jobRepository)
308         .<Student, Student>chunk( chunkSize: 10, transactionManager)
309         .reader(reader)
310         .processor(compositeItemProcessor)
311         .writer(chunk -> chunk.forEach(System.out::println)).build();
```

# Restartability

- Wenn ein Job eine Exception wird, so bricht der Job mit „FAILED“ ab (default)
- Der Job kann dann erneut gestartet werden und macht an der alten Stelle weiter.

# Retry

- Wir können dem Step sagen, dass er Items erneut bearbeite, wenn bestimmte Exceptions geworfen werden.

```
287     ItemProcessor<Student, Student> processor = item -> {
288         String firstName = item.getFirstName();
289         if (firstName.length() < 6) {
290             System.out.println("Name Too Short, let's make it longer.");
291             item.setFirstName(firstName + "eth");
292             throw new CustomException();
293         }
294         return item;
295     };
296
297     TaskletStep step = new StepBuilder( name: "printStudentsStep", jobRepository)
298         .<Student, Student>chunk( chunkSize: 10, transactionManager) SimpleStepBuilder<Student, Student>
299         .reader(reader)
300         .processor(processor)
301         .writer(chunk -> chunk.forEach(System.out::println))
302         .faultTolerant() FaultTolerantStepBuilder<Student, Student>
303         .retry(CustomException.class)
304         .retryLimit(2)
305     .build();
```

# Skip

```
288     ItemProcessor<Student, Student> processor = item -> {
289         String firstName = item.getFirstName();
290         if (firstName.length() < 4) {
291             System.out.println("Name " + firstName + " too Short, SKIP!");
292             throw new CustomException();
293         }
294         return item;
295     };
296
297     ItemWriter<Student> writer = chunk -> {
298         System.out.println(chunk.getItems().size());
299         for (Student student : chunk) {
300             String firstName = student.getFirstName();
301             if (firstName.length() > 9) {
302                 System.out.println("Name " + firstName + " too long. SKIP!");
303                 throw new CustomException();
304             }
305             System.out.println("Name " + firstName + " is ok");
306         }
307     };
308
309     TaskletStep step = new StepBuilder( name: "printStudentsStep", jobRepository)
310         .<Student, Student>chunk( chunkSize: 10, transactionManager) SimpleStepBuilder<Student, Student>
311         .reader(reader)
312         .processor(processor)
313         .writer(writer)
314         .faultTolerant() FaultTolerantStepBuilder<Student, Student>
315         .skip(CustomException.class)
316         .skipLimit(10)
317         .build();
```

Wir überspringen Elemente mit Fehlern

# SkipListener

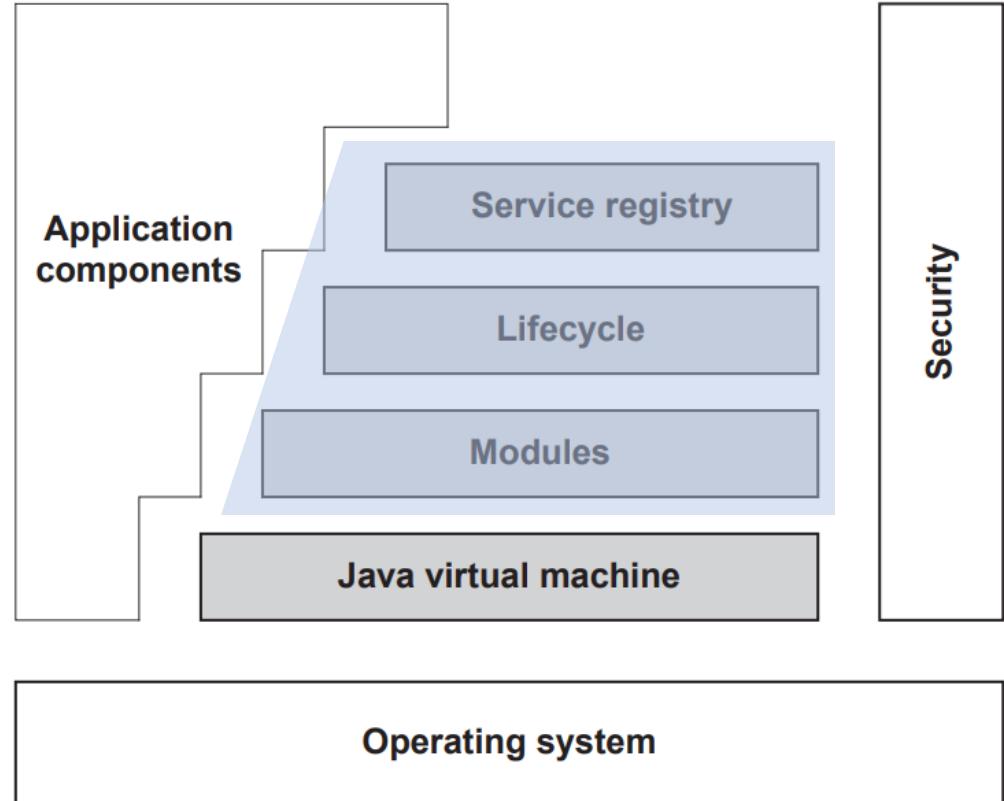
```
309
310
311 ⏹ SkipListener<Student, Student> skipListener = new SkipListener<>() {
312     @Override
313     public void onSkipInWrite(Student item, Throwable t) {
314         System.out.println("FirstName was too long. Item: " + item);
315     }
316 ⏹
317     @Override
318     public void onSkipInProcess(Student item, Throwable t) {
319         System.out.println("FirstName was too short. Item: " + item);
320     }
321
322     TaskletStep step = new StepBuilder( name: "printStudentsStep", jobRepository)
323         .<Student, Student>chunk( chunkSize: 10, transactionManager) SimpleStepBuilder<Student, Student>
324             .reader(reader)
325             .processor(processor)
326             .writer(writer)
327             .faultTolerant() FaultTolerantStepBuilder<Student, Student>
328             .skip(CustomException.class)
329             .skipLimit(10)
330             .listener(skipListener)
331             .build();
332 }
```

# Spring Dynamic Modules

- Spring DM ist eine Möglichkeit den OSGi Standard zu implementieren.
- Der OSGi-Standard bietet die Möglichkeit Java Applicationen in bundles aufzuteilen. Eines von vielen Features ist die Möglichkeit solche bundles zur Laufzeit der Application ohne Downtime auszutauschen.
- Der OSGi-Standard wird faktisch nicht mehr aktiv entwickelt und die Support von Spring DM wurde mit Spring 3.2.4 auch eingestellt.

# OSGi Layer

Layer	Bedeutung
Modules	Verwaltet Komponenten und Module in Form von Bundles.
Lifecycle	Verwaltet den Lifecycle der Bundles. (Installed, Active, ...) Ermöglicht es Bundles während der Laufzeit auszutauschen.
Service registry	Verwaltung der Kommunikation unter den Bundles.

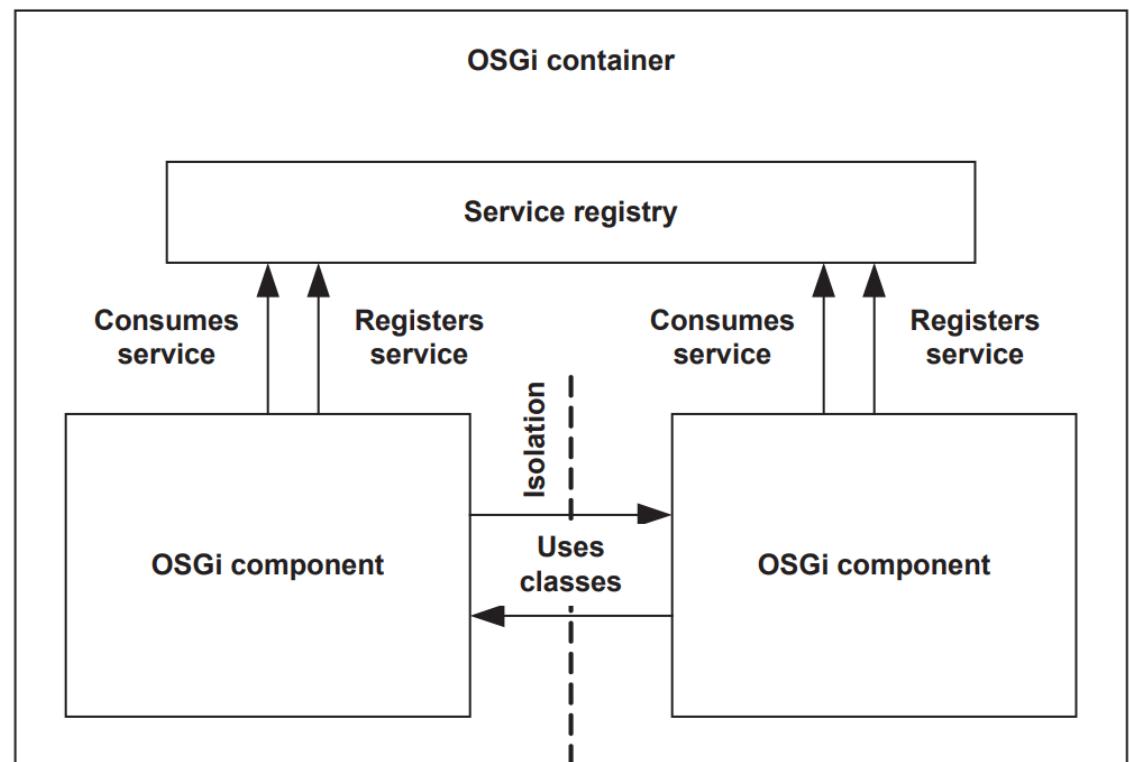


OSGI Bundle:  
JAR-File + Metadata  
(Sichtbarkeit, Identifikation,...)

Bundle-Name: Spring OSGi Bundle  
Bundle-Version: 1.0  
Bundle-SymbolicName: com.manning.spring.osgi.simple  
Export-Package: com.manning.spring.osgi.simple.service  
Import-Package: com.manning.spring.osgi.utils

# OSGi container

- OSGi liefert die Möglichkeit modulare Anwendungen zu entwickeln.
- Diese bestehen aus Komponenten (bundles)
- Die bundles kommunizieren Service basierend.



# Spring DM

- Spring DM bietet die Möglichkeit Spring-Applicationen zu entwickeln die sich leicht in einer OSGi Umgebung durchführen lassen.
- Spring stellte damals keine guten Möglichkeiten zur Modularität zur Verfügung.
  - jede Bean überall sichtbar.
  - Es gibt meist nur einen globalen Application Context.
  - Abhängigkeiten unter den Beans sind nicht zur Laufzeit änderbar.
- OSGi enthält keine Pattern oder Tools für das Design und die Implementierung von bundels. Spring DM hilft dabei
- Weiterhin hilft Spring DM beim einfachen Zugriff von Komponente zu Komponente, ohne die fehleranfällige Service Management API direkt zu verwenden.

Danke 

- Feedback zur Schulung geben:
- <https://www.provenexpert.com/oliver-kohl/a8hd/>



# Copyright und Impressum

© 2024 CONTECO e.U.

CONTECO e.U.  
Herzfeldergasse 30  
2351 Wiener Neudorf  
Österreich

<https://www.conteco.gmbh>

Tel AT: +43 660 3550300

Tel DE: +49 152 51352074

E-Mail: [training@conteco.gmbh](mailto:training@conteco.gmbh)

