Contents

[1. JPA 2](#_Toc39654208)

[2. Dependency Injection general 2](#_Toc39654209)

[3. Dependency Injection using Spring 3](#_Toc39654210)

[a. Value 3](#_Toc39654211)

[4. JUnit basics 4](#_Toc39654212)

[5. Mockito basics 4](#_Toc39654213)

[6. Reflection basics 4](#_Toc39654214)

[7. Design Patterns 5](#_Toc39654215)

[a. Builder 5](#_Toc39654216)

[b. Proxy 5](#_Toc39654217)

[c. Singleton 5](#_Toc39654218)

[d. Factory 5](#_Toc39654219)

[e. Observer 5](#_Toc39654220)

[8. Http/Rest basics 5](#_Toc39654221)

[9. Spring Web/MVC 6](#_Toc39654222)

[a. DispatcherServlet 6](#_Toc39654223)

[b. Request mapping 6](#_Toc39654224)

[c. Exception handling 7](#_Toc39654225)

[d. Security 7](#_Toc39654226)

[10. Spring Data 7](#_Toc39654227)

[a. Transactions 8](#_Toc39654228)

[b. Inject EntityManager 8](#_Toc39654229)

[11. AOP 8](#_Toc39654230)

[a. AspectJ Annotations 8](#_Toc39654231)

[12. Logging 9](#_Toc39654232)

[13. Architecture 11](#_Toc39654233)

[a. API 11](#_Toc39654234)

[b. DTO 11](#_Toc39654235)

[c. MVC 11](#_Toc39654236)

[d. Layers 11](#_Toc39654237)

# JPA

* 1. Relationships

**@OneToMany:**

Zum Mappen einer 1:n Beziehung (1 Kunde hat mehrere Bestellungen aber eine Bestellung gehört nur zu 1 Kunden)

**@ManyToMany:**

Zum Mappen einer m:n Beziehung (1 Lehrer hat mehrere Schüler und 1 Schüler hat mehrere Lehrer)

**@JoinColumn:**   
erzeugt eine Fremdschlüsselspälte (bei @ManyToOne)

**@JoinTable:**

Erzeugt eine Fremdschlüsseltabelle (bei @ManyToMany)

# Dependency Injection general

Abhängigkeiten werden von außen bereit gestellt.

3 Arten von DI:

* DI über den Konstruktor:

+ gut bei Pflichtfelder (mandatory)

+ keine Nullpointer checks

- schlecht bei optionalen Feldern

- bei vielen Parametern wird diese Methode sehr unübersichtlich

* DI über Setter:

+ gut bei optionalen Feldern

- schlecht bei Pflichtfelder (mandatory)

- Nullpointer checks müssen gemacht weren

* DI bei Feldern:

+ weniger Code

- Nullpointer checks

- man braucht DI Framework (Spring)

# Dependency Injection using Spring

**@Component:**   
 Klasse die sich im Spring Context befindet

**@Qualifier:**   
bei mehreren Spring-Komponenten werden damit eine konkrete Komponente ausgewählt die injected werden soll

**@Primary:**  
Annotation für Komponente, es wird immer diese ausgeführt, wird von @Qualifier überschrieben

**@Autowired:**  
 hier wird eine Dependency injected

**@Configuration:**  
 Annotation für Klasse in der programmatisch Beans erzeugt werden

**@Bean:**   
Methode, die ein Spring-Bean erzeugt, in Klassen die mit @Configuration annotiert werden

**@Lazy:**   
 bestimmt, dass die Beans lazy (erst wenn sie gebraucht werden) erzeugt werden

**@Scope:**  
bestimmt, ob die Beans als Singleton implementiert werden oder nicht, Standard: Singleton, kann man auf prototype ändern, dann wird für jede injection eine neue Bean erzeugt

**@ComponentScan:**  
 gibt an in welchen Package noch nach Beans gesucht werden soll

**@PropertySource:**  
 gibt die Property an, aus der man Werte lesen kann

**@Value:**  
 damit werden Werte aus einer Property gelesen  
 @Value("${value}")   
 String value;

## Value

## Methode:

## Systemproperties können einfach mit @Value(„${name}“) ausgelesen werden

ODER

## Methode

Alle Systemproperties injecten:  
 @Autowired  
 @Qualifier(„systemPorperties“)  
 Properties properties;

Darauf zugreifen:  
 Properties.getProperty(“Property Name”);

* + 1. SpEl (Spring Expression Language)

**Systemproperties mit SpEl holen:**

@Value(“#{sytemProperties[‘user.name’]}

# JUnit basics

# Reflection basics

Reflection:

mit Reflection kann man zur Laufzeit das Programm überprüfen und analysieren  
 Reflection ist für die Implementierung von dynamischen Verhalten verantwortlich  
mit Hilfe von Reflections kann man u. A. Methoden, Felder und Annotationen auslesen

**Annotationen:**  
 Metainformationen für Compiler/Framework

**Frameworks erstellen: 🡪 Annotationen erstellen**

public @interface Table {  
 String value();  
}

**@Retention(RetentionType.{CLASS,RUNTIME, SOURCE})**

um zu bestimmen wann die Annotation verfügbar ist

**@Target(ElementType.{FIELD, METHOD, TYPE, ...})**

um zu bestimmen bei welchen Elementen die Annotation verfügbar ist

**Value setzen:**

@Table("Test")

Die Annotation wird vor dem erstellten Target gesetzt und somit gemarked.

**Werte holen:**

mit Reflection

# Design Patterns

## Builder

Wird für immutable Objekte und bei vielen optionalen & Pflichtfelder verwendet  
der Konstruktor des Objekts ist privat und eine Instanz wird über ein **public static class Builder** erzeugt. In diesen Builder werden alle Pflichtfelder im Konstruktor der Builder-Klasse angegeben und die optionalen Felder werden mit Setter angegeben.

## Proxy

Fügt Funktionalitäten hinzu und arbeitet mit konkreten Implementierungen  
ist ein Platzhalter für ein Objekt  
anstatt das richtige Objekt aufzurufen, zuft man den Proxy auf, der eine Implementierung des Objekts aufweist und zusätzliche Funktionen

## Singleton

Dient dazu, dass immer nur eine Instanz von einem Objekt erstellt werden kann.   
Es gibt Eager-Singleton & Lazy-Singleton.

## Factory

Dient zur Trennung zwischen der Erstellung & der Verwendung eines Objektes  
in einer Factory-Klasse wird entschieden welches Objekt erstellt wird  
in dieser Factory-Klasse gibt es nur 1 privaten Konstruktor und die Methode, in der entschieden wir welches Objekt erstellt wird

## Observer

Dient dazu Änderungen an einem Objekt an andere Objekte, die diese Änderung interssiert, weiterzugeben  
um ständige Abfragen zu vermeiden  
in dem beobachtenden Objekt wird eine Observer-Liste angelegt, in der sich die verschiedenen Observer registrieren, wird das Objekt verändert, muss man die verschiedensten Observer aufrufen

# Http/Rest basics

Objekte werden mit JSON (Format für Datendarstellung) verschickt.

Json Bsp.:  
 {  
 „firstname“: „Max“,  
 „lastname“: „Mustermann“  
 }

Rest-Services sind stateless, d.h. jede Anfrage muss alle Daten beinhalten um die Aktion auszuführen, da sich historische Daten nicht gemerkt werden

**Standardoperation (CRUD-Operationen):**

* GET: User lesen
* POST: User erstellen
* PUT: User updaten (id muss mitgegeben werden)
* DELETE: User löschen

Resourcen die abgefragt werden, werden in der URL inkludiert (Bsp: GET/customer)

Man kann verschiedene Parameter mitgeben (Bsp: GET/customer?lastname=Mayer)

**Header:**Metainforamationen die Informationen enthalten, z.B. welches Format benötigt wird, welche Http-Operation verwendet wurde, die URL, ...)

**HTTP-Statuscodes:**

* 200: OK (GET, PUT)
* 201: Created (POST)
* 204: No Content (DELETE)
* 400: Bad Request
* 403: Forbidden
* 404: Not Found
* 418: I’m a Tea pot 🡪 Easter Egg
* 500: Internal Server Error

# Spring Web/MVC

## DispatcherServlet

Der Dispatcher Servlet behandelt die Anfragen und weist diese dem richtigen Controller zu.

## Request mapping

**@RestController:**   
sorgt dafür, dass diese Klasse als Rest-Endpunkt angelegt wird, Klassen mit dieser Annotation sind implizit mit @Component annotiert

**@GetMapping:**   
Methoden, der @RestController Klasse, um bestimmte Methode an eine URL zu binden die man mit GET bekommt, kurzschreibweie für: @Requestmapping(path=“/hello“, method=RequestMethod.GET), man kann als Parameter consumes und produces hinzufügen (geben den MimeType (XML, JSON,...) den sie bekommen und den sie zurück schicken)

**@Postmapping:**   
Methoden mit dieser Annotation machen ein POST-Request, kurzschreibweise für @Requestmapping(method=RequestMethod.POST)

**@Putmapping:**Methoden mit dieser Annotation machen ein PUT-Request, kurzschreibweise für @Requestmapping(method=RequestMethod.PUT)

**@DeleteMapping:**   
Methoden mit dieser Annotation machen ein DELETE-Request, kurzschreibweise für @Requestmapping(method=RequestMethod.DELETE)

**@RequestMapping:**   
kann man URL angeben mit der alle Methoden der Klasse verfügbar sind

**Unterschied @RestController & @Controller**@Restcontroller generiert automatisch bei jeder Methode die Annotation @ResponseBody hinzu und @Controller nicht.

**@RequestParam:**   
bei der URL kann man mit „?name=Timon“ den Wert der Variable ändern, der mit @RequestParam(value=“name“) im Parameter der Methode angegeben ist  
  
Bei mehreren argumenten mit & verknüpft: „?firstname=Timon&lastname=Gnadenberger“

**@PathVariable:**mit dieser Annotation holt man den Wert, der im Pfad übergeben wird

**@RequestBody:**   
Der übergeben Wert wird im Body des HTTP-Requests mitgegeben

**ResponseEntity:**Response zurückschicken um den Statuscode zu ändern

## Exception handling

Annotationen:

@ControllerAdvice, darunter die Klasse muss ResponseEntityExceptionHandler extenden   
Wird bei allen Controllern hinzugefügt. Man kann zudem noch Spezifische Exception in den einzelnen Controller Klassen einbauen.

@ExceptionHandler(“Exception Klassenname.class”)

Z.b: **public** ResponseEntity<Object> handleException(Exception e, RuntimeException ex, WebRequest webRequest)

Dabei ist sehr wichtig, die Exceptions in den Parametern der Methode müssen zu der Exception passen die im ExceptionHandler definiert ist.

## Security

@EnableWebSecurity 🡪 Erweitert eine @Configurable Annotation und sorgt dafür, dass diese mit der Spring Security Konfiguration erweitert wird. FRAGEN WIR MORGEN NOCH EINMAL!

# Spring Data

Spring Modul, das den Umgang mit der DB vereinfacht

wenn man mit Repositories arbeitet müssen alle interfaces die Annotation @Repository haben und mindestens das CrudRepository implementieren

**CrudRepository:**

C – Create  
 R – Read  
 U – Update  
 D - Delete

damit kann man Objekte lesen, erzeugen, updaten und löschen

**in Repositorys:**

* mann kann Queries definieren die keine Implementierung haben, die müssen findBy... heißen, diese Magic funktioniert mit DynamicProxies

List<Movie> findByTitle(String title);

* bei komplizierteren Queries wird die Annotation @Query verwendet, @Query("select m from Movie m where m.title like %:partOfTitle"),

dabei muss im Parameter der Wert für die Query mit @Param("partOfTitle") String title) übergeben werden

@Query("select m from Movie m where m.title like %:title%")

List<Movie> findByPartOfTitle(@Param("title") String title);

## Transactions

Methoden die auf eine DB zugreifen, werden mit der Annotation @Transactional annotiert

## Inject EntityManager

@PersistenceContext  
EntityManager em;

# AOP

ist dafür da um Funktionalitäten die mehrere Methoden betreffen, zu den Methoden hinzuzufügen ohne sie überall implementieren zu müssen

**Cross-Cutting Concern**: Funktionalitäten, die nichts mit der Logik der Methode zu tun haben, die man aber trotzdem hinzufügen möchte

## AspectJ Annotations

**@Aspect** definiert ein Aspekt

**@Around**:  
 wird vor und nach allen Methoden aufgerufen

**@Before**:  
 wird vor allen Methoden aufgerufen

**@After**:   
 wird vor Returnwert aufgerufen

**@AfterReturning**:  
 wird nach den Returnwert aufgerufen

**@AfterThrowing**:  
 wird aufgerufen, nachdem eine Exception geworfen wurde

**ProceedingJoinPoint:**  
 Die Methode die man wrappen möchte, wird aufgerufen (around)

**JoinPoint:**   
 Ruft man nicht auf aber man kann Information davon bekommen (before & after)

# Logging

immer gegen API programmieren -> Logger kann man leichter austauschen

**Logger werden verwendet:**

* um die Ausgabe in einem gleichen Format auszugeben
* um die Ausgabe leichter lesbar zu machen
* da man Ausgaben mit Logger leichter in Files schreiben kann
* da man Ausgaben mit Logger leichter verschlüsseln kann

**verschiedene Ausgaben von Logger:**

**LOGGER.trace():**   
 detailreiche Infos über den Ablauf des Programs ausgeben

**LOGGER.debug():**  
 Debug-Infos ausgeben

**LOGGER.info():**  
 allgemeine Infos ausgeben, die das Programm betreffen

**LOGGER.warning():**   
 Fehler werden protokolliert, die nicht zu einem Abbruch führen, aber ein falsches Ergebnis liefern könnte

**LOGGER.error():**   
 Programmfehler werden protokolliert

**Man kann Logger auch Exceptions hinzufügen:**

Logger.error("Fehler", Exception);

**Logger Konfiguration in logback.xml:**

Es gibt einen root-Logger der angibt ab welchem Level Ausgabenerzeugt werden sollen

<root level="info">  
 <appender-ref ref=""/>  
 </root>

in appender-ref wird ein konfigurierter Appender eingefügt (ConsoleAppender, FileAppender, ...)

möchte man für bestimmte Packete/Klassen z.B. detailiertere Ausgaben muss ein logger in der logback.xml konfiguriert werden:

<logger name="Package/Class" level="debug" additivity=false>  
 <appender-ref ref=""/>  
 </logger>

additivity=false -> Logger wird nicht doppelt ausgegeben

# Architecture

## API

Application Programm Interfaces bietet Programmierschnittstellen zu anderen Programmen und somit wird das System zur Verfügung gestellt.

## DTO

Sind Objekte, die nur die Informationen beinhalten, die die nächsthöhere Schicht benötig

## MVC

Model View Controller 🡪 Muster zur Unterteilung einer Software in die drei Komponenten. Datenmodell (Modell), Präsentation (View), Programmsteuerung (Controller).

## Layers

Schichten warden verwendet um die Anwendung in verschiedene Teile zu unterteilen, die alle eine Aufgabe haben (z.B. DB-Zugriff, Business Logik, User Interface)  
Jede Schicht kann nur mit der darunterliegenden Schicht kommunizieren (dabei werden DTOs verwendet)  
Theoretisch sind verschiedenste Schichten austauschbar